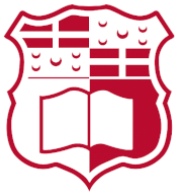


Static Checking of Concurrent Programs in Elixir Using Session Types

Gerard Tabone **Adrian Francalanza**

Technical Report
March 2022

Faculty of ICT

University of Malta

Abstract

Message-passing concurrency is becoming increasingly popular, and is prevalently used in several programming languages, such as Go, Scala, Rust and Elixir. In message-passing concurrency, processes interact with each other by exchanging messages. If these messages are sent incorrectly, certain behavioural issues, such as communication mismatches and deadlocks, may arise.

We employ session types, which are a form of behavioural types, to detect these kinds of communication errors at compile-time, in the context of the Elixir language. More concretely, session types are protocols used to define the whole message-based interaction between processes, by dictating the order and type of messages that can be sent and received. We use these protocols to annotate public functions in Elixir modules to formalise their expected behaviour in terms of the messages sent and received when executing the body of said functions. Then, we design a (session) type system that uses these protocols to statically analyse and determine whether the functions observe the session type specification. This type system is validated from a formal perspective, by showing that it follows the session fidelity property. Moreover, the type system is validated empirically, by being implemented as a tool, which automates typechecking for Elixir modules.

Contents

List of Figures

List of Definitions

1. Introduction

Concurrency [1] is ubiquitous in modern computing. Most systems, ranging from small embedded systems to large cloud servers, run on multiple processors working in parallel. Concurrency refers to units of computations, called processes, that can execute simultaneously with other processes. Concurrent processes typically interact with one another throughout their execution, and they do so via either message-passing or shared memory. In message-passing concurrency, isolated processes share data by exchanging messages. Since multiples processes cannot access common data, memory-based issues such as data races, cannot occur. This makes message-passing concurrency more structured than the shared-state counterpart, since the interactions and interferences can be more readily delineated [2]. For this reason, the message-passing paradigm is becoming increasingly popular, with a number of modern languages adopting it. Examples include Go [3], Scala [4], Rust [5], Swift [6] and Elixir [7], the last being our language focus.

Programming concurrent software, even when this is limited to message-passing, is inherently harder when compared to its sequential counterpart. Concurrent programs may compile successfully, but then they may behave in a different manner than intended, or even exhibit unexpected errors at runtime. These problems may be caused by a number of factors. For example, if a message is sent with an unusual form, it may cause a *communication mismatch* (*i.e.,* failing to handle an incoming

```
1  defmodule Auction do
2     @spec buyer(pid, number) :: atom
3     def buyer(auctioneer_pid, amount) do
4        send(auctioneer_pid, {:bid, amount})
5
6        receive do
7           {:sold}          -> :yay
8           {:higher, value} -> decide(auctioneer_pid, amount, value)
9        end
10    end
11
12    @spec decide(pid, number, number) :: atom
13    defp decide(auctioneer_pid, amount, value) do
14       if value < 100 do
15          send(auctioneer_pid, {:continue})
16          buyer(auctioneer_pid,  amount + 10)
17       else
18          send(auctioneer_pid, {:quit})
19          :ok
20       end
21    end
22    ...
23 end
```

Listing 1.1: Auction system written in Elixir

message or sending an unexpected message). Also, if messages are exchanged in an incorrect order, they may cause a *deadlock* (*i.e.,* multiple processes waiting forever for each other). A concurrent program may only execute *correctly*, if it is free from these kinds of behavioural issues.

For instance, consider an *auction* system, adapted from [8], whereby a buyer process bids on some item in an auction. The auction is moderated by an auctioneer, who may either accept the bid, declaring the item as sold, or else, inform the buyer of a new higher bid, allowing room for further bids. A sample `Auction` module, written in the Elixir language, is shown in Listing 1.1, which implements the buyer's side of the interaction. It offers one public function called `buyer` on lines 3–10, that takes two arguments: the *process identifier* (*pid*) of the interacting auctioneer, `auctioneer_pid`, and the starting bid value, `amount`.

The interaction starts with the buyer sending an initial bid to the auctioneer (line 4); this message contains the label `:bid` and the amount as a payload. Then,

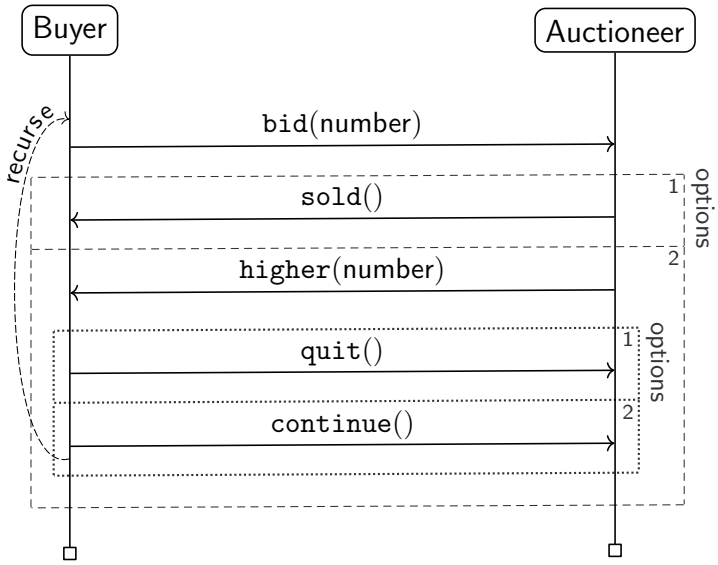


Figure 1.1: Auction protocol

the buyer waits for a reply. This happens in the **`receive`** construct (lines 6–9), which acts as a blocking mechanism and it continuously checks the process's mailbox until some message has been received. It only unblocks when a message containing the expected format is found. The buyer can accept two types of messages; a label `:sold`, showing that the bid was accepted, or a label `:higher` carrying a new bid `value`, showing the outbid value. The code then branches accordingly. In case of `:sold`, the function returns, terminating the interaction with notification `:yay`. In case of `:higher`, the remaining interaction is handled by the private function called `decide` (line 13), where the buyer has two choices: to either increase the bid or opt out of the auction. In line 14, the buyer decides to only consider a new bid if the expected amount is less than a €100. If it is the case, the buyer sends a request to remain in the auction, containing a label `:continue`. Then, it recurses back to the beginning updating the new bid amount (line 16). On the other hand, if the bidding amount went beyond the buyer's expectation, then the buyer can send a message to `:quit` the auction (line 18), before terminating the interaction.

The *whole* series of messages exchanged can be depicted as the protocol shown

```
3  def buyer(auctioneer_pid, amount) do
4     send(auctioneer_pid, {:bid, true}
5
6     receive do
7        # {:sold}        -> :yay
8        {:higher, value} ->
9           if value < 100 do
10             send(auctioneer_pid, {:continue})
11             buyer(auctioneer_pid,  amount + 10)
12             ...
13          end
14    end
15 end
```

Listing 1.2: Buyer in an auction with issues

in Figure 1.1. It consists of two parties, a buyer and an auctioneer, that both must follow the protocol from their point-of-view. The protocol dictates that the buyer must send a message to the auctioneer containing a label `bid` with some payload of type number. Then, the auctioneer makes a choice to either send a label `sold` or a label `higher` with some number. In case of `higher`, then the buyer can choose to either send `quit`, to terminate the session, or send `continue`, to restart the protocol from the beginning. If the buyer follows the protocol, but the auctioneer veers away (or vice versa), then the interaction fails. Correct computation occurs only if both parties adhere to their side of the protocol.

Since this protocol is only implicit (*i.e.,* not written down), the implementation might unknowingly deviate from it. These deviations might be small enough to go unnoticed, but then, cause the program to misbehave at runtime. For example, consider an erroneously `buyer` function, redefined in Listing 1.2. The buyer mistakenly sends a bid request with a payload *true* in line 4. This may cause *communication mismatch* on the auctioneer's side, who might be only handling bids with a payload of type number, as expected in Figure 1.1. Payload mismatches can also be induced by the auctioneer. In line 8, if the buyer receives a label `higher` with a value `:ok`, the program will misbehave, since the buyer would be expecting a number and not a boolean (*i.e.,* `higher`(`number`)). This incorrect value

will then propagate to the subsequent `if` statement (line 9), where the expression '`:ok < 100`' will evaluate to an unexpected result.[1]

Another example is shown in line 7. If this line is (accidentally) removed, the buyer would only be able to handle messages labelled `higher`. If the auctioneer replies with a label `sold`, as allowed by Figure 1.1, the buyer will not be able to handle the label `sold`, causing the program to *deadlock*. So, we need a way to define protocols explicitly and pass them to the compiler, which should be able to pinpoint these issues earlier on.

These behavioural issues, albeit crucial, are typically ignored during compilation. For example, languages such as Scala, Rust and Go, offer some basic checks [5, 9, 10]. They are able to restrict channels (used to transfer messages) to a fixed type, thus preventing incorrectly formed messages from being transferred. Conversely, languages such as Erlang and Elixir, offer complete freedom in message-passing, thus allowing any messages to be sent to any process [11, 12]. Nevertheless, none of these languages offer a way to check messages in the context of the whole interaction (*e.g.*, by using a protocol), leaving them susceptible to behavioural issues. In the case of Elixir, erroneous programs, such as Listing 1.2, will be deemed *correct* by the present Elixir compiler, even though they will be ill-behaved during runtime. This is because the Elixir compiler and other static analyser tools tend to focus only on the functional part of the language. For example, the Elixir compiler ensures that the correct syntax is used, or that functions are called with the correct number of parameters. Other tools, such as the Dialyzer [13, 14], extend these static validations. The Dialyzer uses the `@spec` annotations (*e.g.*, line 2 of Listing 1.1) to enforce that the parameters and return types are of the expected type (*e.g.*, when invoking the function `buyer`, the second argument, called `amount` on line 3 of Listing 1.1, should be a number).

The aforementioned behavioural software defects can be detected during early stages of development using *behavioural types* [15], or more specifically, *session*

[1] In Elixir, '`:ok < 100`' evaluates to *false*, while in other languages (*e.g.*, Scala), a similar operation causes a compile-time issue.

types [15–17]. A (binary) session type is a protocol between two processes, which defines the order and type of messages that are allowed (and expected) to be transferred. Session types can be used to statically verify that concurrent programs are free from communication mismatches or deadlocks, thus ascertaining that the interaction between concurrent processes progresses safely. Moreover, since the developer is encouraged to define the interaction explicitly, it results in software that is structured better and easier to understand. This improves both the software reliability and the development process [18]. Session types work by ensuring that each *sent* message is *received* safely on the other end, and vice versa. Other than the sending and receiving constructs, sessions types may contain branching, choice and recursion. Furthermore, since messages may contain some payload, session types describe both the label of the message and the payload types.

In practice, there have been many applications of session types for existing languages that work either as static analysers (*e.g.*, Go [19], Scala [20], OCaml [21, 22] and Rust [23, 24]), or as monitors, running at runtime (*e.g.*, Scala [25], Python [26], OCaml [27] and Erlang [28]). Our implementation takes the most predominant approach (*i.e.,* static analyser in the form of a type checker) and creates a static type system for a subset of the Elixir language, to ensure protocol adherence.

1.1 Aims and Objectives

The aim of this work is to improve the development of concurrent code, specifically within the Elixir language. We can improve this by detecting common behavioural issues that arise in message-passing systems, such as communication mismatches and certain deadlocks. To do so, we need to be able to encode the implicit communication protocols (*e.g.*, Figure 1.1), as explicit protocols via session types, ideally by integrating them in a natural way in existing Elixir programs (*e.g.*, Listing 1.3), in a way that it is not disruptive to the developer. Using these session

types, we need a type system, backed by a formal foundation, that verifies the code statically, *i.e.,* at pre-deployment stages.

We can outline these aims in the following two objectives:

O1. *Establish a formal foundation in the form of a type system.* The type system should be defined by the static typing rules and the operational semantics. We also have to prove some properties related to the type system, including session fidelity, which ascertains that the functions adhere to the session type protocols precisely.

O2. *Develop a proof-of-concept session type implementation in Elixir.* This practical type checker should, to a reasonable extent, mirror the formal type system. The implementation should also integrate seamlessly within an existing Elixir ecosystem, where it is the not intrusive to the developer's workflow. We should also analyse the applicability (and limitations) of our tool through practical case studies.

1.2 Solution Overview

We revisit Listing 1.1 to see how session types could be integrated within the existing code with *minimal* changes, to prevent behavioural issues. First, we will formalise the session type, and then we insert it in the code. Then, using the new additions, we should be able to detect behavioural issues (*e.g.*, Listing 1.2) at compile-time. We can infer the session type from the protocol in Figure 1.1, where we take the buyer's point-of-view and formalise it in a session type called *auction*, as follows:

$$auction = !\texttt{bid}(\mathsf{number}).\& \left\{ \begin{array}{l} ?\texttt{sold}().\mathsf{end}, \\ ?\texttt{higher}(\mathsf{number}).\ \oplus \left\{ \begin{array}{l} !\texttt{quit}().\mathsf{end}, \\ !\texttt{continue}().auction \end{array} \right. \end{array} \right.$$

```
1  defmodule Auction do
2     use ElixirST
3     @session "auction = !bid(number).
4                            &{?sold().end,
5                              ?higher(number).
6                                       +{!quit().end,
7                                         !continue().auction}}"
8     @spec buyer(pid, number) :: atom
9     def buyer(auctioneer_pid, amount) do
10       send(auctioneer_pid, {:bid, amount})
11
12       receive do
13          {:sold}          -> :yay
14          {:higher, value} -> decide(auctioneer_pid, amount, value)
15       end
16    end
17
18    @spec decide(pid, number, number) :: atom
19    defp decide(auctioneer_pid, amount, value) do
20       if value < 100 do
21          send(auctioneer_pid, {:continue})
22          buyer(auctioneer_pid,  amount + 10)
23       else
24          send(auctioneer_pid, {:quit})
25          :ok
26       end
27    end
28
29    @dual "auction"
30    @spec auctioneer(pid, number) :: atom
31    def auctioneer(buyer_pid, min) do
32       ...
33    end
34 end
```

Listing 1.3: Auction system annotated with *session types*

The type *auction* states that the buyer process has to send (*i.e.,* !) a label `bid` with a payload of type number (*i.e.,* !`bid`(number)). Then, it has to branch (*i.e.,* &) in two ways. If it receives (*i.e.,* ?) a label `sold`, it has to terminate the session (*i.e.,* end). If it receives a label `higher` with a payload of type number, then it can make a choice (*i.e.,* $\oplus$). It can either send a `quit` label, terminating the session, or else it may send a `continue` label, recursing back to the beginning. The auctioneer process should also follow Figure 1.1 from its point-of-view, *e.g.*, by performing the *dual* actions of *auction*, *i.e.,* $\overline{auction}$.

The next step is to insert the session types *auction* and $\overline{auction}$ in the existing code. We use annotations to add session types. Annotations are commonly used to add checks or information in Elixir code, *e.g.*, the Dialyzer uses the `@spec` annotation. The modified code is depicted in Listing 1.3, where we retrofit the `Auction` module to use two new annotations: **`@session`** and **`@dual`**. In lines 3–7, the `buyer` has to adhere to the *auction* session type (*i.e.,* `@session`). In lines 31–33, we defined another public function called `auctioneer`. By adding the `@dual` annotation (line 29), the function `auctioneer` is forced to follow the dual session type $\overline{auction}$.

Our proposed (session) type system adds session types to a subset of Elixir, serving multiple purposes. By having the interaction explicitly formalised as a session type, one can glance at the session type and get a clearer idea of what is happening, resulting in better code structure and fewer bugs. Moreover, by typechecking the session-typed functions in a module, it enforces that each function precisely follows the expected interaction, as defined by the protocol, flagging issues at pre-deployment stages, *e.g.*, the issues in Listing 1.2 will be caught earlier on if session types are used.

This work is an extension of our paper [29] published in the AGERE 2021 workshop [30], where the session type system (and implementation) for Elixir, was initially presented.

1.3 Document Outline

The next chapter (Chapter 2) provides some background information on type systems and the Elixir language which will help set the context for the remaining chapters. Chapter 3 provides the design of the (session) type system from a formal aspect, covering the first part of Objective O1. Chapter 4 solidifies the foundation of the formal system by proving some properties about our type system, including *session fidelity*, addressing the remainder of Objective O1. Chapter 5 illustrates

the design and implementation of the type system, by creating type checker, called ElixirST. This chapter covers the remaining objective, Objective O2. Due to the nature of this work, the document structure is slight non-standard. Specifically, the evaluation is split in two parts: Chapter 4 validates the formal system from a formal perspective, while Chapter 5 validates it empirically by being implementing the type system as an Elixir type checker. The related work is discussed in Chapter 6. Finally, Chapter 7 concludes.

2. Background

We provide the necessary background to enable a better understanding of our work. We then relegate the comparison of our work to the state-of-the-art implementations in Chapter 6.

To this end, we introduce a *simple* type system, showing its importance to detect compile-time errors. Then, the basics of labelled transition systems and concurrency, used to model the execution of an Elixir program, are discussed. Finally, we take a look at the Elixir language, an inherently dynamic and concurrent programming language.

2.1 Type Systems

Consider a simple language with the following expressions:

$$e ::= \texttt{not}\ e \mid e_1 + e_2 \mid e_1 < e_2 \mid \mathit{boolean} \mid \mathit{number}$$

An expression e can take the form of a negation (*i.e.,* `not` e) which inverts a *true* to a *false* (and vice versa), addition of two expressions (*i.e.,* $e_1 + e_2$) and the comparison of two expressions (*i.e.,* $e_1 < e_2$). An expression can also take the form of a basic value, that includes booleans (*i.e., true*, *false*) and numbers. Using

this syntax, we can construct an infinite number of different programs, such as

$$5 \ + \ 2 < 10 \tag{1}$$

$$\texttt{not}\ \mathit{false} \tag{2}$$

$$\texttt{not}\ (8 \ < \ 2) \tag{3}$$

$$1 \ + \ 1 < \mathit{false} \tag{4}$$

Expressions (1–3) can be evaluated (*i.e.,* reduced) fully, resulting in a value, *e.g.*, consider expression (1):

$$5 + 2 \ < \ 10 \quad \rightarrow \quad 7 \ < \ 10 \quad \rightarrow \quad \mathit{true}$$

These are considered well-behaved expressions. On the other hand, we can have expressions that get stuck when reducing, resulting in ill-behaved expressions, *e.g.*, expression (4) reduces to the expression $2 < \mathit{false}$ which gets stuck:

$$1 + 1 \ < \ \mathit{false} \quad \rightarrow \quad 2 < \mathit{false} \quad \nrightarrow$$

One way to find out if an expression is well- or ill-behaved is to evaluate the program until it either reduces to a value (*i.e.,* well-behaved), or else it stops due to some error (*i.e.,* ill-behaved). Unfortunately, finding errors after the program is deployed can be expensive to fix and catastrophic if it crashes at a crucial time. Another problem is that some programs may execute for a long time (or even forever), so hoping that the program never crashes may not be an option. The ideal time to spot errors is as early as possible, that is, before we even execute the program, during compilation. This can be achieved via a *static type system.*

A type system flags issues related to the types of expressions, and decides if a program is well- or ill-typed. If a program is well-typed and the type system is sound, then we expect the program to be well-behaved, and thus it should progress safely without crashing caused by type errors. Typically, programs tend to "just

work" once they pass the typechecker [31]. If a program is ill-typed, then it could potentially manifest execution errors.

The language that we introduced earlier only deals with values having a boolean or number types:

$$T ::= \mathsf{boolean} \mid \mathsf{number}$$

We will now add typing rules to each form of expression using the typing judgement $\vdash e : T$, which says that an expression e has type T. Starting with the basic values (referred to as b), any *boolean* literal must have type boolean (and similarly, for numbers), *e.g.*, *true* has type boolean and 4 has type number. This is reflected in the typing rule [tLiteral] – it uses the `type` function which returns the type of a literal, *e.g.*, $\mathtt{type}(5) = \mathsf{number}$.

$$[\textsc{tLiteral}]\ \frac{\mathtt{type}(b) = T}{\vdash b : T} \qquad [\textsc{tNot}]\ \frac{\vdash e : \mathsf{boolean}}{\vdash \mathtt{not}\ e : \mathsf{boolean}}$$

The next rule is [tNot] which states that a negation expression (`not` e), as well as its sub-expression (e), must be booleans.

$$[\textsc{tAdd}]\ \frac{\vdash e_1 : \mathsf{number} \qquad \vdash e_2 : \mathsf{number}}{\vdash e_1 + e_2 : \mathsf{number}} \qquad [\textsc{tLess}]\ \frac{\vdash e_1 : \mathsf{number} \qquad \vdash e_2 : \mathsf{number}}{\vdash e_1 < e_2 : \mathsf{boolean}}$$

The rule [tAdd], expects the addition expression ($e_1 + e_2$) to have type number; moreover, it relies on the premises that both sub-expressions (e_1 and e_2) must be numbers as well. The final rule, [tLess], typechecks the "less than" operator, in which the final result will have a boolean type. It also depends on its sub-expressions, which are expected to have type number.

Example 2.1. Consider expression (3), `not` $(8 < 2)$. We will use the newly defined typing rules to see if it typechecks (*i.e.,* well-typed).

$$\dfrac{\dfrac{\dfrac{\texttt{type}(8) = \mathsf{number}}{\vdash 8 : \mathsf{number}}[\textsc{tLiteral}] \quad \dfrac{\texttt{type}(2) = \mathsf{number}}{\vdash 2 : \mathsf{number}}[\textsc{tLiteral}]}{\vdash 8 < 2 : \mathsf{boolean}}[\textsc{tLess}]}{\vdash \texttt{not}\ (8 < 2) : \mathsf{boolean}}[\textsc{tNot}]$$

Starting from the bottom, we use the rules [tNot], [tLess] and [tLiteral]. Since all rules are used correctly, then the original expression is deemed well-typed with respect to the final boolean type. Thus, it is expected to be well-behaved. ■

Example 2.2. Consider another example (expression (4)) where we will examine whether $1 + 1 < \mathit{false}$ is well-typed.

$$\dfrac{\dfrac{\dfrac{\vdash 1 : \mathsf{number} \quad \vdash 1 : \mathsf{number}}{\vdash 1 + 1 : \mathsf{number}}[\textsc{tAdd}] \quad \dfrac{\texttt{type}(\mathit{false}) = \mathsf{boolean}}{\vdash \mathit{false} : ??}[\textsc{tLiteral}]}{\vdash 1 + 1 < \mathit{false} : \mathsf{boolean}}[\textsc{tLess}]$$

The first rule that we use is [tLess], which expects that the premises' types to be both numbers. A problem arises from the following rule, [tLiteral], which results in a boolean instead of a number, resulting in a clash. Therefore, we can conclude that this example fails typechecking, and thus *may* fail during execution. ■

We have seen the static phase of processing in the language, which typechecks a program with respect to some typing rules (*i.e.,* type system). This, in turn, decides if a program is well-formed. This well-formedness approximates whether a language will execute with or without errors [32]. Note that, typechecking an expression, in principle, is more efficient than computing the actual value, since it approximates rather than performing the actual computations. The next phase is *execution*, which is formally defined by the reduction or transition semantics (which we introduce in Section 2.2). These rules specify the step-by-step way of executing a program.

If we merge the static typing rules with the reduction semantics, we can decide if a language is *safe*. Type safety dictates that "well-typed programs cannot go wrong" [33]. This means that if a program typechecks, we expect it to execute without issues. Type safety can be shown by proving two properties: *preservation* and *progress*. Type preservation, can be achieved by showing that a well-typed program remains well-typed after reduction.

Example 2.3. For $e = \texttt{not}\ (8\ <\ 2)$, in Example 2.1 we have shown that e is well-typed, *i.e.,* $\vdash e : \mathsf{boolean}$. Assuming typical reduction semantics, we can reduce e to some e', *e.g.*, $\texttt{not}\ (8\ <\ 2) \rightarrow \texttt{not}\ \mathit{false}$. Then, we expect that e' remains well-typed with respect to the same type. This holds if the language possesses the preservation property. ■

The other property is *progress*. Progress dictates that well-typed progress cannot "get stuck", meaning that all well-formed expressions must be able to reduce to another expression, unless they are already a value.

Example 2.4. In the previous example, we have shown that e is well-typed and it can reduce to another expression e'. By the progress property, e' should also be able to continue reducing until a value is reached:

$$\texttt{not}\ (\ 8\ <\ 2\ )\quad \rightarrow\quad \texttt{not}\ \mathit{false}\quad \rightarrow\quad \mathit{true}\quad \nrightarrow$$

When it reaches the *true* value, no more reductions are possible. Assuming that the reduction trace is correct, then the progress property is satisfied in this example. ■

2.2 Labelled Transition System

The language and type system introduced in Section 2.1 will be expanded to handle a practical subset of the Elixir language in Chapter 3. In order to observe how Elixir

programs evaluate, we define its transition semantics.

A program is made up of a *term* t (where $t \in \textsc{Terms}$) that transitions into another term, producing an action $\alpha \in \textsc{Act}$. The action α can be an internal or external side-effect (*e.g.*, sending a message). Transitions are denoted as $t \xrightarrow{\alpha} t'$, where a term t transitions to t', producing an action α. The transition $t \xrightarrow{\alpha} t'$ is a shorthand notations for the relation $\rightarrow \subseteq (\textsc{Terms} \times \textsc{Act} \times \textsc{Terms})$, where $t, t' \in \textsc{Terms}$ and $\alpha \in \textsc{Act}$.

Since we need to handle actions during transitions, we have to model how a program evaluates using a transition semantics, defined in terms of a *labelled transition system* (LTS) [34], as follows.

Definition 2.1 (*Labelled Transition System*)**.** A *labelled transition system* (LTS) is defined by a triple $\langle \textsc{Terms}, \textsc{Act}, \rightarrow \rangle$, where

- $\textsc{Terms}$ is the set of terms (or states) that our program may exist as;
- $\textsc{Act}$ is the set of actions (or labels) that may be produced during a transition;
- $\rightarrow \subseteq (\textsc{Terms} \times \textsc{Act} \times \textsc{Terms})$ is a transition relation. ■

In Section 3.5, we provide a complete LTS which models the runtime semantics of an Elixir program. These simulate how Elixir programs execute in the Erlang Virtual Machine.

2.3 Concurrency

Concurrency refers to units of execution, referred to as *processes*, which can run independently, with other processes [1, 35]. Structuring programs in a concurrent fashion may increase its complexity, however it has its benefits, *e.g.*, if a concurrent program is executed on a multi-core system, its processes can execute in parallel, resulting in an increase in performance. A number of concurrency models have been developed, which differ in how processes interact and share data between them. We

consider the shared state and message-passing concurrency, while focusing on the latter.

2.3.1 Shared State Concurrency

Concurrency using a shared state works by having a common memory area which is shared between multiple processes. These processes can typically read and write simultaneously on the same memory, allowing for faster access when sharing data between processes, since the original data can be accessed directly. This might lead to concurrency issues, such as data races, where processes access data in an inconsistent order. Constructs such as locks, mutexes and semaphores may be used to prevent such issues [35], although they are not trivial, so applying them incorrectly may lead to further issues. Shared state concurrency is widely used, including in C threads [36], and CUDA threads which run on a Graphics Processing Unit (GPU) [37].

2.3.2 Message-Passing Concurrency

Our focus is on concurrency via *message-passing*. In this model of concurrency, processes typically have access to only their own memory space, so the only way to interact with other processes is to send messages via a communication medium (*e.g.*, a channel). When messages are sent over a medium, the messages need to be handled either in a *synchronous* or an *asynchronous* manner. In synchronous message-passing, the process sending a message waits until the receiver is able to handle the message being sent. The following messages may only be sent if the preceding messages were received. In asynchronous message-passing, the process sending the messages does not need to wait for the other processes to handle the message.

We consider the two main paradigms in message-passing concurrency, which are channel-based and the actor-based concurrency. Our work focuses on the latter,

message-passing using the actor model.

Channels

In channel-based concurrency, messages are transferred over a communication channel. A channel is shared between different processes, which consists of two ends: a transmitter and a receiver. A process sends messages in the receiver end, whilst messages are received from the other end. Channels may allow synchronous and asynchronous messaging. Channel-based concurrency is implemented in modern languages, including Go [2] and Rust [5].

Actors

In message-passing concurrency, the *actor model* describes concurrent processes, called *actors*, which share no memory and communicate by exchanging messages directly. This model was first introduced by Hewitt *et al.* [38] and formalised by Agha [39]. An actor is defined as follows:

Definition 2.2 (*The Actor Model*)**.** An *actor* [39] is able to perform three actions:

- Change its state depending on the messages received;
- Send asynchronous messages to other actors;
- Create *new* actors. ■

In an *actor system*, processes (or *actors*), can *spawn* other processes. Each process executes in a separate environment and can only communicate with other processes via *message-passing*. Processes can send *asynchronous messages* to other processes using their unique address, called a process identifier (*pid*) which is assigned to each process during creation. Then, messages are collected in the process' mailbox where the addressee can choose any message to read.

The actor model is becoming increasingly popular due to its advantages: it has better fault tolerance, *i.e.,* if one process crashes, it will not accidentally crash

another process, since processes do not share memory. For the same reason, concurrent programs can easily scale depending on the number of CPU cores available.

The actor model is used in several languages, including Scala, Swift, Erlang and Elixir [6, 7, 9]. Our focus is on the Elixir language, which we introduce in the following section (Section 2.4).

2.4 Elixir

The Elixir [7] programming language is a dynamically typed, functional language, which offers message-passing concurrency via the *actor model* (Definition 2.2). Elixir runs on the Erlang virtual machine (BEAM).

Elixir takes advantage of the *Erlang* ecosystem, which consists of the Erlang language, the OTP (Open Telecom Platform) library and the BEAM (Bogdan's/Björn's Erlang Abstract Machine) [40]. The Erlang language, as well as OTP (which contains some standard Erlang libraries such as the `supervisor`/`gen_server` features), are compiled to bytecode that runs on the BEAM. Similarly, Elixir is compiled to bytecode that runs on the same platform. The BEAM can handle thousands of processes running concurrently.

Elixir was open-sourced by José Valim in 2012, in which the lanuage offers additional powerful mechanisms when compared to Erlang. These include metaprogramming and tools which help to streamline testing, documentation and package publishing. Nowadays, Elixir has been gaining adoption in a multitude of areas and companies, notably Discord, Pinterest and Apple [41, 42].

2.4.1 Functional Aspect

We explore some of the Elixir language aspects, starting from its sequential (*i.e.*, functional) part, and then introduce some of its concurrency features.

Modules and Functions Elixir programs are typically structured as modules containing a number of functions. These functions can be executed sequentially or spawned in concurrent fashion.

```
1 defmodule Math do
2   @spec add(number, number) :: number
3   def add(a, b) do
4     a + b
5   end
6 end
```

In this example, we have a module called `Math` (line 1) which contains a single public function `add` (line 3), that applies and returns the addition of two numbers. For example, executing `Math.add(1,2)` returns 3. Public functions are defined by the **`def`** keyword; these can be called from outside the defining module. On the other hand, private functions are defined by the **`defp`** keyword and can only be called from *within* the module.

Annotations In Elixir, it is common to decorate functions with annotations to either add details or impose assertions. An example is shown in line 2, where the `@spec` annotation is used to add type information to the function parameters and return value. In this case, the `add` function takes two numbers as parameters and returns another number. This information may be used to produce documentation containing types, or used by the Dialyzer. The Dialyzer is a static analyser that detects type errors using success typing [13]. The Dialyzer (or Dialyxir[1]) ascertains that the function parameters and return types follow the specifications provided by `@spec`. Annotations are a common pattern in Elixir. Another example includes Elixir's `doctest` feature[2] which uses the `@doc` annotation to decorate functions with documentation, code examples and tests inside the implementation itself. We

[1]Dialyzer is an Erlang tool, so we use Dialyxir (https://github.com/jeremyjh/dialyxir) which ports the Dialyzer for the Elixir environment.

[2]https://elixir-lang.org/getting-started/mix-otp/docs-tests-and-with.html#doctests

will exploit annotations to add further assertions, which we describe in Chapter 3.

Pattern Matching A powerful mechanism of Elixir is its pattern matching operator, *i.e.,* `=`. It compares the values on the right-hand side with the pattern on the left-hand side. If the pattern contains values, then it checks for equality, and if there are variables, it assigns them with a value.

```
7  x = 7
8  {4, y} = {4, :hello}        # y = :hello
9  {5} = {9}                   # fails
```

In line 7 of this example, the variable `x` is assigned to a value 7, having type number. Line 8 has a tuple containing a number 4 and an atom[3] `:hello`. This tuple is pattern matched with the left-hand side, where the first element checks out (*i.e.,* both 4) and the second element, `y` is assigned to a value `:hello`, which has type atom. In the final line, we attempt to pattern match a tuple containing a value 9 with the expected tuple containing a value 5. This action fails.

Control Flow In a programming language, you need to be able to make choices and follow different paths. In Elixir, we can choose a different path using the case construct; where, given a value, it pattern matches it with different options.

```
10 case var do
11    7 -> :seven
12    9 -> :nine
13    other -> other
14 end
```

In the above example, we compare the value of the variable `var` with 7. If it matches, then it returns `:seven`. If it fails, `var` is compared similarly to 9. Finally, if none of the previous cases match, it will match with the last case, since the variable `other` (line 13) does not have any preconditions.

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

Another common language feature is the if-clause (line 15), which works similar to other languages.

```
15 if Integer.mod(z, 2) == 0 do
16     :even
17 end
```

2.4.2 Concurrent Aspect

We use the basic building blocks established from the functional part of Elixir (Section 2.4.1) to take advantage of the BEAM's concurrency features.

Spawn BEAM, and by extension, Elixir, observe the actor model (Definition 2.2), so we should be able to spawn processes (or actors).

Taking as an example the previous `add/2` function from the module `Math`, we can use the `spawn` function provided by Elixir. It takes the module and function names, along with the parameter values, or alternatively an anonymous function. When executed, it immediately returns the *pid* value of the new spawned process.

```
18 pid = spawn(Math, :add, [1, 3])
```

By design, spawned processes are independent, therefore they do not share memory with the other processes. So, having multiple processes is only useful if they could interact, or exchange data with other processes.

Send and Receive A process can send data to another process' *pid* using the **`send`** operation. On the other hand, a dual process, can stop and wait to receive a message using the **`receive`** construct. This works by picking each message from the process' mailbox and checks for validity via pattern matching, akin to how the **`case`** construct works. This receive operation is blocking and will only release after a valid message is received, or some timeout is reached.

```
19 # process 1
20 send(pid2, :some_value)
21 # ...
```

```
22 # process 2
23 receive do
24   value ->
25       IO.puts("Received #{value}")
26 end
```

In this example, the first process (line 20) sends a message to some *pid*, and the other process (line 23) waits to receive a message, outputting its value when received.

In more advanced concurrent implementations, abstraction features from the OTP library are used; these help in reusing similar patterns. For example, the `gen_server` abstraction manages the server in *client-server* relationship; and the `supervisor` abstraction helps to form a *supervision tree*, managing the creation and death of child processes, following Erlang's *let it crash* philosophy, which handles errors by letting processes crash. These more advanced abstractions are beyond the scope of this project; we will only consider the *send-receive* pattern.

2.4.3 Present Validations

When Elixir code is compiled, the compiler only validates the sequential part of the code. For example, it ensures that the correct syntax is used (*e.g.*, each open bracket has a matching close bracket), or that function calls are allowed from the present module. Considering that Elixir is dynamically typed, typechecking is carried out at runtime, not at compile-time. So, some (partially) valid programs, such as:

```
if condition do
   :ok
else
   :foo + 1
end
```

despite being accepted by the compiler, may not always run correctly. For instance, if the variable `condition` evaluates to true, the program runs without issues, but if

it evaluates to false, then the program reaches a runtime error. Some tools, such as the Dialyzer, can statically analyse the code to catch further errors. The Dialyzer is one example, which analyses the code at compile-time to catch type errors. In Section 2.4.1 we saw how the Dialyzer uses the `@spec` annotation to enforce that the parameters and return types conform to the expected types.

Although the sequential part of the code is thoroughly analysed, the concurrent part of the code is not checked by the present Elixir compiler. This may leave vulnerabilities in the code, where behavioural errors (described in Chapter 1) such as deadlocks and communication mismatches, may cause issues at runtime. In this study we build a static type checker that analyses (a subset of) the Elixir language, with a focus on its concurrency features.

2.5 Conclusion

In this chapter we laid the foundations of how type systems interact with a simple language and introduced how languages may evaluate via labelled transition systems. We also introduced the concept of concurrency, and more specifically, the actor model, which underpins Elixir's concurrent aspect. In the following chapters, we will create and formalise a new session typing system, and provide a proof-of-concept implementation for Elixir.

3. A Formal Analysis

In this chapter we introduce a core subset of the Elixir language. Following this, we provide a formal foundation for Elixir programs by constructing a (session) type system and an operational semantics. The type system is then validated from a formal perspective in Chapter 4, and implemented as a static type checker in Chapter 5.

3.1 Outline of the Approach

Elixir programs are composed of several public and private functions organised in modules, as shown in the example in Figure 3.1. Functions can be considered the basic unit of decomposition in Elixir programs, since difference tasks can be split in different functions. A module may expose some of these functions to external entities by defining them as *public* functions. These public functions provide the only entry points for a module. Internally, these public functions may split their tasks among *private* functions, which exist inside a module and can only be referenced from within the same module. For instance, Figure 3.1 depicts a module m which contains several public (*e.g.*, f_1) and private functions (*e.g.*, f_1'). *E.g.*, the public function f_1 splits its work by calling other private functions, *i.e.*, f_1' and f_m'.

Public functions can be spawned to create processes, where they can communicate with other processes via message-passing. In Figure 3.1, the public

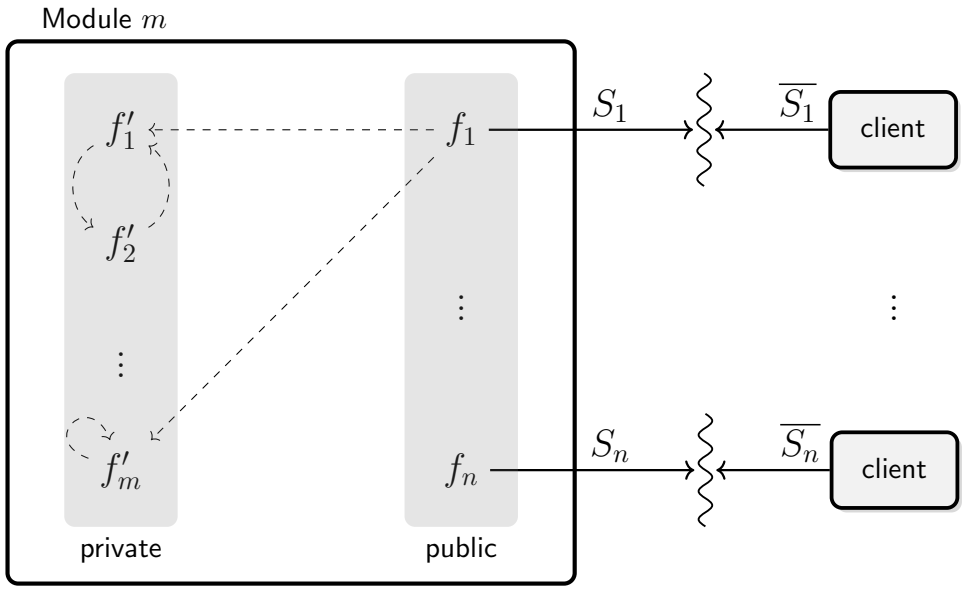


Figure 3.1: A sample Elixir module, consisting of public and private functions, which interact with client processes

function f_1 is spawned, creating a process which interacts with a client process. Processes tend to communicate by exchanging messages in a structured but implicit manner. This structure, or protocol, is presently being assumed by both interacting parties, since Elixir does not provide a way to explicitly define these protocols (and enforce them). This forces the processes to 'guess' which messages are going to be received or which messages are expected to be sent, which could result in behavioural issues explained in Chapter 1.

Our design addresses this shortcoming, using a twofold approach:

1. It provides a way to attach a protocol (using session types) to each public function, thus defining the structure of the interaction explicitly. This allows the interacting processes to know, rather than guess, how the interaction will progress. For example, the function f_1 in Figure 3.1 interacts with a client process following a certain protocol, which we define explicitly as S_1. This allows other interacting processes (*e.g.*, the client) to safely interacting with f_1 **if** they follow a protocol compatible with S_1 (*e.g.*, $\overline{S_1}$).

2. These protocols are then used to enforce that the implementations of the public

functions follow them precisely. This is achieved by building a type system that statically typechecks each public function against a session type, thus flagging issues at pre-deployment stages. For example, the public function f_1 (Figure 3.1) is typechecked statically against the protocol S_1, ascertaining that its implementation is faithful to the interactions defined by the protocol S_1. Furthermore, this is not a trivial approach, since only public functions *know* which session types to follow (since they are explicitly defined). In turn, these public functions may call private functions which do not have an explicit session type. Despite this, these private functions still have to follow the (fragmented) protocol inherited from the callee function.

3.2 Elixir Syntax

We introduce the syntax for a fragment of the Elixir language. This syntax is shown in Figures 3.2 and 3.3, and features Elixir modules, functions, terms, expressions, types and the *new* annotations. We let variables x, y range over variable names and symbol `l` ranges over labels. Variable m denotes module names and f denotes function names.

3.2.1 Session Types

As motivated in Chapter 1, we design a *static* type system which applies session types to a fragment of the Elixir language. This type system assumes the standard expression types (Figure 3.2), including basic types, such as boolean, number, atom and pid, and inductively defined types, such as tuples ($\{T_1, \ldots, T_n\}$) and lists ($[T]$).

Remark. *Throughout this work, we use a shorthand notation where* $\widetilde{T}$ *(i.e., tilde over symbol) stands for the possibly empty sequence* $T_1, \ldots, T_n$. *Other (meta)variables including* p, P, D, v, x *and* e, *also follow the same convention, e.g.,* $\widetilde{x} = x_1, \ldots, x_n$ *and* $\widetilde{p} = p_1, \ldots, p_n$. ■

$$
\begin{array}{rrll}
\text{Session types} & S & ::= & \&\{?\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I} & \text{Branch} \\
& & \mid & \oplus\{!\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I} & \text{Choice} \\
& & \mid & \mathsf{rec}\,\mathsf{X}\,.\,S & \text{Recursion} \\
& & \mid & \mathsf{X} & \text{Recursion variable} \\
& & \mid & \mathsf{end} & \text{Terminate} \\
\text{Expression types} & T & ::= & \mathsf{boolean} \mid \mathsf{number} \mid \mathsf{atom} \mid \mathsf{pid} & \\
& & \mid & \{T_1,\ \ldots,\ T_n\} \mid [T] &
\end{array}
$$

Figure 3.2: Types

Expression types, albeit dynamically checked, already exist in the present Elixir language. Our type system extends these types to *new* types, called session types, which will be used to statically check that the message-passing interaction complies to a pre-defined protocol. Session types are made up of the branching, choice, recursion and termination constructs, as shown in Figure 3.2.

The *branching* session type, $\&\{?\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}$, dictates that a message is received, containing any one of the labels $\mathtt{l}_i$. This message must be accompanied by a payload of type $\widetilde{T}_i$. The interaction then progresses as S_i. Dual to the branching type, we have the *choice* type, $\oplus\{!\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}$. It specifies that a message labelled $\mathtt{l}_i$, along with a payload of type $\widetilde{T}_i$, can be sent. The interaction progresses as S_i. The crucial difference between the branching and choice types is where the choose of message lies, *i.e.,* in the branching session type, an external party decides which message to send, so the receiving party must handle all available branches. Dually, the choice type offers an internal option, where at least one of the message choices may be chosen. Note that, the labels in the branching and choice types need to be pairwise distinct, *i.e.,* two messages that have the same label cannot be sent (or received) at one time. Moreover, for clarity, the symbols & and $\oplus$ can be omitted

for singleton options, *e.g.*, $\oplus\{!\mathtt{A}(\mathsf{number}).S_1\}$ is equivalent to $!\mathtt{A}(\mathsf{number}).S_1$.

Session types can also take the form of a *recursion* construct, $\mathsf{rec}\ \mathsf{X}\,.\,S$, where the recursion variable X is bound inside the session type S. For recursion, we take the more liberal approach, *i.e., equi-recursion* [31]. This means that the recursive type ($\mathsf{rec}\ \mathsf{X}\,.\,S$) and its unfolding ($S\,[^{\mathsf{rec}\ \mathsf{X}\,.\,S}/_{\mathsf{X}}]$) are deemed equivalent, and thus can be used interchangeably. The remaining form is the *termination* session type, end. It terminates the session and prevents any further message exchanges. For brevity, end may be omitted, *e.g.* $?\mathtt{B}()$ is equivalent to $?\mathtt{B}().\mathsf{end}$.

If a process following session type S interacts with another process following the *dual* session type of S, then it is guaranteed that their interaction progresses safely (*i.e.,* no communication mismatches or deadlocks). The dual session type of S, written as $\overline{S}$, describes the *dual* actions depicted by S (Definition 3.1).

Definition 3.1 (*Duality*)**.** The session type $\overline{S}$ is the dual of S:

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

$$\overline{\oplus\{!\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}} = \&\{?\mathtt{l}_i(\widetilde{T}_i).\overline{S_i}\}_{i\in I} \qquad \overline{\mathsf{X}} = \mathsf{X} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \blacksquare$$

Example 3.1. Consider a server process that follows the session type $S = \&\{?\mathtt{l}_1(), ?\mathtt{l}_2(), ?\mathtt{l}_3(), ?\mathtt{l}_4()\}$, *i.e.,* the process must receive a message labelled $\mathtt{l}_1$, $\mathtt{l}_2$, $\mathtt{l}_3$ or $\mathtt{l}_4$. This server can interact safely with a client process, if the client sends one of the labels $\mathtt{l}_1$, $\mathtt{l}_2$, $\mathtt{l}_3$ or $\mathtt{l}_4$. This is depicted formally as the protocol $\overline{S} = \oplus\{!\mathtt{l}_1(), !\mathtt{l}_2(), !\mathtt{l}_3(), !\mathtt{l}_4()\}$. This session type $\overline{S}$ is the dual session type of S. However, the client process may also interact safely if it decides to only send labels $\mathtt{l}_1$ or $\mathtt{l}_3$, *i.e.,* it follows the session type $\oplus\{!\mathtt{l}_1(), !\mathtt{l}_3()\}$ which is *smaller* than the dual session type $\overline{S}$. This is possible through session type subtyping [43], which goes beyond our scope. $\blacksquare$

Example 3.2. Recall the annotated auction example from Listing 1.3. The interaction between the buyer and the auctioneer can progress safely if both parties

follow the protocol depicted in Figure 1.1. We established that protocol from the buyer's point of view is called *auction*:

$$\mathit{auction} = !\texttt{bid}(\mathsf{number}).\& \left\{ \begin{array}{l} ?\texttt{sold}().\mathsf{end}, \\ ?\texttt{higher}(\mathsf{number}). \oplus \left\{ \begin{array}{l} !\texttt{quit}().\mathsf{end}, \\ !\texttt{continue}().\mathit{auction} \end{array} \right. \end{array} \right.$$

This *auction* session type starts with a singleton choice, containing a label `bid` and a payload of type number (*i.e.,* $\widetilde{T} = \mathsf{number}$). Then, it can branch (&) to either labels `sold` or `higher`. If it branches to the latter, it can then make an internal choice ($\oplus$) to send the label `quit` or `continue`. From the opposite side of the interaction, the auctioneer follows the **dual** session type of *auction*, called $\overline{\mathit{auction}}$:

$$\overline{\mathit{auction}} = ?\texttt{bid}(\mathsf{number}). \oplus \left\{ \begin{array}{l} !\texttt{sold}().\mathsf{end}, \\ !\texttt{higher}(\mathsf{number}).\& \left\{ \begin{array}{l} ?\texttt{quit}().\mathsf{end}, \\ ?\texttt{continue}().\overline{\mathit{auction}} \end{array} \right. \end{array} \right.$$

In $\overline{\mathit{auction}}$, the auctioneer has to first receive a label `bid` with a payload of type number, and then it has to make a choice. It can either accept the bid, sending a `sold` label and then terminating. Or else, it can send the new higher bid amount (*i.e.,* $!\texttt{higher}(\mathsf{number})$) before either terminating (*i.e.,* $?\texttt{quit}()$) or continuing the auction process (*i.e.,* $?\texttt{continue}().\overline{\mathit{auction}}$). ■

3.2.2 Modules and Functions

The preceding expression and session types are used within Elixir programs, as defined by the syntax in Figure 3.3. As shown in Figure 3.1, Elixir programs are organised as modules, *i.e.,* `defmodule` m `do` $\widetilde{P}$ $\widetilde{D}$ `end`. Modules are defined by their name, m, and contain a sequence of public $\widetilde{D}$ and private $\widetilde{P}$ functions.[1] Public functions, `def` $f(y, \widetilde{x})$ `do` t `end`, are defined by the **`def`** keyword, and can

[1] Although we group public and private functions together, their order is irrelevant.

$$
\begin{array}{rrll}
\text{Module} & M & ::= & \texttt{defmodule}\ m\ \texttt{do}\ \widetilde{P}\ \widetilde{D}\ \texttt{end} \\
\text{Public function} & D & ::= & K \quad B \quad \texttt{def}\ f(y,\ \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \\
\text{Private function} & P & ::= & B \quad \texttt{defp}\ f(y,\ \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \\
\text{Type ann.} & B & ::= & \texttt{@spec}\ f\big(\widetilde{T}\big)\ ::\ T \\
\text{Session ann.} & K & ::= & \texttt{@session}\ \text{“}\mathsf{X} = S\text{”} \\
 & & \mid & \texttt{@dual}\ \text{“}\mathsf{X}\text{”}
\end{array}
$$

$$
\begin{array}{rrll}
\text{Basic values} & b & ::= & \mathit{boolean} \mid \mathit{number} \mid \mathit{atom} \mid \mathit{pid} \mid [\,] \\
\text{Values} & v & ::= & b \mid [\,v_1 \mid v_2\,] \mid \{v_1,\ \ldots,\ v_n\} \\
\text{Identifiers} & w & ::= & b \mid x \\
\text{Patterns} & p & ::= & w \mid [\,w_1 \mid w_2\,] \mid \{w_1,\ \ldots,\ w_n\} \\
\text{Terms} & t & ::= & e \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Expression} \\
 & & \mid & x = t_1;\ t_2 \qquad\qquad\qquad\qquad\qquad\qquad \text{Let statement} \\
 & & \mid & \texttt{send}\,(w, \{\texttt{:l}, e_1,\ \ldots,\ e_n\}) \qquad\qquad\qquad\qquad \text{Send} \\
 & & \mid & \texttt{receive do} \qquad\qquad\qquad\qquad\qquad\qquad \text{Receive} \\
 & & & \quad \big(\{\texttt{:l}_i, p_i^1,\ \ldots,\ p_i^n\} \to t_i\big)_{i \in I}\texttt{end} \\
 & & \mid & f\,(w,\ e_1,\ \ldots,\ e_n) \qquad\qquad\qquad\qquad\qquad\quad \text{Function call} \\
 & & \mid & \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} \qquad\qquad\qquad\quad \text{Case}
\end{array}
$$

$$
\begin{array}{rrll}
\text{Expressions} & e & ::= & w \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{Identifiers} \\
 & & \mid & \texttt{not}\ e \mid e_1\ \diamond\ e_2 \qquad\qquad\qquad\qquad\qquad\quad \text{Operations} \\
 & & \mid & [\,e_1 \mid e_2\,] \mid \{e_1,\ \ldots,\ e_n\} \qquad\qquad\qquad\qquad\quad \text{Lists \& tuples} \\
\text{Operators} & \diamond & ::= & <\ \mid\ >\ \mid\ <=\ \mid\ >=\ \mid\ ==\ \mid\ != \\
 & & \mid & +\ \mid\ -\ \mid\ *\ \mid\ /\ \mid \texttt{and} \mid \texttt{or}
\end{array}
$$

Figure 3.3: Elixir syntax

be called from any module. In contrast, private functions, `defp` $f(y, \widetilde{x})$ `do` t `end`, are defined by the **`defp`** keyword, and can only be called from within the defining module. Functions are defined by their name, f, and their body, t. They are also parametrised by a sequence of *unique* variables, $y, \widetilde{x}$, where $\widetilde{x} = x_1, \ldots, x_n$. The total number of parameters is called the *arity*. The first parameter, y, is reserved for the variable containing the *pid* of the dual process. For instance, the public function `buyer` from Listing 1.1 has arity 2, and the first parameter (`auctioneer_pid`) represents the *pid* of the dual process (auctioneer). Moreover, a module can only contain functions with unique names, unless the arity is different. Functions serve different purposes in our design: public functions are used to spawn individual processes; and private functions are used as a looping mechanism, since in Elixir, looping is typically done via recursion.

Elixir is a dynamically typed language, so the variables types are analysed at runtime. On the other hand, since we are building a static type checker, we cannot have variables with unknown types at compile-time. Therefore, we decorate all unknown variables in function definitions with their types. To achieve this, we exploit the `@spec` annotation, provided by the Dialyzer, to add type specifications preceding *all* function definitions. This annotation takes the form of $f(\widetilde{T}) :: T$, which describes the parameter types ($\widetilde{T}$) and the return type (T) of the function f.

The most crucial part of this type system is the analysis of the functions' (concurrent) behaviour with respect to some protocol. To be able to do this, we decorate public functions with the session types defined in Figure 3.2. We can annotate public functions directly using `@session` "$\mathsf{X} = S$", or indirectly using the dual session type, `@dual` "X". The notation $\mathsf{X} = S$ is shorthand for $\mathsf{rec}\ \mathsf{X}\,.\,S$. Moreover, this notation also allows for the session type S, to be assigned a label X, and thus can be referenced by `@dual` annotations defined within the same module.

3.2.3 Terms and Expressions

The body of a function is made up of some term, t. Terms can take the form of an expression, a *let* statement, a send or receive construct, a case statement or a function call, as defined in Figure 3.3.

The *let* statement, $x = t_1;\ t_2$, evaluates term t_1, binding its result to x in t_2, and then it evaluates t_2. We can write $t_1; t_2$, as *syntactic sugar* for $x = t_1;\ t_2$ (given that x is not used in t_2); the semicolon (;) can be omitted if t_1 and t_2 are on different lines. The *send* statement, $\texttt{send}\,(x, \{\texttt{:l}, e_1, \ \ldots, \ e_n\})$, allows a process to send a message to the *pid* stored in the variable x, containing a message $\{\texttt{:l}, e_1, \ \ldots, \ e_n\}$, where `:l` is the label.

Remark. *The contents of the messages, i.e., the label and the payloads, are unrolled into a tuple, with the first element, a literal atom, acting as the label, followed by the payloads. For example, consider the session type* `!bid`(*number*)*, where a message containing a label* `bid`*, along with a payload of type number, needs to be sent. A valid Elixir snippet that observes this protocol would be:* `send(pid, {:bid, 500})`*, where* `pid` *contains the dual pid, and* `{:bid, 500}` *is the labelled message being transferred.* ■

The *receive* construct, $\texttt{receive do}\ (\{\texttt{:l}_i, p_i^1, \ \ldots, \ p_i^n\} \rightarrow t_i)_{i \in I}\texttt{end}$, allows a process to receive a message, by blocking until a valid message is fetched from the process' mailbox. A message is chosen if its label matches with one of the labels $\texttt{:l}_i$, and also the payloads must correspond to the patterns $p_i^1, \ \ldots, \ p_i^n$. Then it continues executing as t_i. Note that patterns, p (defined in Figure 3.3), can take the form of a variable (*e.g.*, x), a basic value (*e.g.*, *false*), a tuple (*e.g.*, $\{x\}$) or a list (*e.g.* $[\,x \mid y\,]$, where x is the head and y is the tail of the list).

The *case* statement, $\texttt{case}\ e\ \texttt{do}\ (p_i \rightarrow t_i)_{i \in I}\texttt{end}$, evaluates some expression e and matches the result with the possible patterns p_i, continuing as t_i. We assume that the union of all patterns $(p_0, \ \ldots, \ p_i, \ \ldots, \ p_n)$ cover all possible values of e. For instance, if e has type list (*i.e.*, $[\,T\,]$), then two possible cases of p_i need to be $[\,]$

and $[\, x \mid y \,]$. Similarly, if the type of e is a number and the first pattern p_1 is 4, then we assume that one of the remaining patterns, p_i, contains a catch-all variable x, thus accepting any number that e might evaluate to (since e could evaluate to *any* number, not just 4).

Finally, terms can take the form of an *expression*, e (refer to Figure 3.3). An expression can be a variable (*e.g.*, x), a basic value (*i.e., boolean*, *number*, *atom*, *pid*), a negation operation (*i.e.,* `not` e), a binary operation, a list or a tuple. Binary operations include comparison operations ($<$, $>$, $<=$, $>=$, $==$, $!=$), arithmetic operations ($+$, $-$, $*$, $/$) and boolean operations (`and`, `or`). Lists contain a sequence of elements having a uniform type, *e.g.*, $[\, 1, 5, 10 \,]$, while tuples may contain a sequence of heterogeneous elements, *e.g.*, $\{\, 1, \texttt{:hello}, \mathit{true} \,\}$.

Given some term t, we can compute the *free variables* using the function $\mathbf{fv}(t)$. Similarly, we can get a set of *bound variables* using the function $\mathbf{bv}(t)$. The full definitions can be found in Appendix A. Using the **fv** function, we can define *closed* and *open* terms, as follows.

Definition 3.2 (*Closed and Open Terms*)**.** A term t is *closed* whenever t does not contain any *free variables*, *i.e.,* $\mathbf{fv}(t) = \emptyset$. Dually, a term t is *open* whenever $\mathbf{fv}(t) \neq \emptyset$. ■

Example 3.3. Consider the following function definition (`successor`), which takes a *pid* (`y`) and a number (`num`) as parameters. This function increments `num` and sends the value as a labelled message to *pid* `y`:

t
```
def successor(y, num) do
  x = num + 1
  send(y, {:val, x})
end
```

When we compute the free variables of the function's body (t), using the **fv** function, we get $\mathbf{fv}(t) = \{\texttt{num}, \texttt{y}\}$, since `num` and `y` are not bound inside the body.

Thus, the body t is considered an *open* term.

On the other hand, we can substitute the free variables with some values, *e.g.*,

$$t' \left\{ \begin{array}{l} \texttt{x = 5 + 1} \\ \texttt{send}(\iota, \texttt{\{:val, x\}}) \end{array} \right.$$

In this case t' has no free variables, *i.e.,* $\mathbf{fv}(t') = \emptyset$, so t' is said to be a *closed* term. Moreover, in the first line, `x` becomes bound to the result of $1 + 6$ by the *let* statement, so $\mathbf{bv}(t') = \{\texttt{x}\}$. ■

3.3 Session Typing

In Figure 3.1, we presented how Elixir programs can be structured, which consist of modules made up of several functions. Moreover, we introduce a new approach of how we can discipline the side-effects (*i.e.,* messages), using explicit protocols, called session types. In this section, we build a session typing system that statically verifies that these protocols are being adhered to. To achieve this, we typecheck the modules and their functions. We analyse functions by typechecking their body, which in turn is made up of terms and expressions. We take a bottom-up approach, where we first present the *expression typing* rules (Section 3.3.1), and move towards the *term typing* rules (Section 3.3.3), and finally, the *module typing* rules (Section 3.3.4). This type system uses a variable binding environment:

$$\Gamma ::= \emptyset \mid \Gamma,\ x : T$$

The *variable binding* environment, Γ, maps (data) variables to their basic types ($x{:}T$). We write $dom(\Gamma)$ to dictate the set of variables mapped by Γ, and $\Gamma, x{:}T$ to extend Γ with the new mapping $x{:}T$, where $x \notin dom(\Gamma)$. The empty environment is represented by $\emptyset$.

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

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

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

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

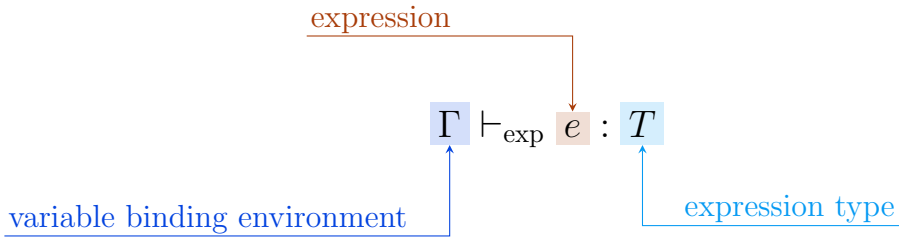
$$[\textsc{tBoolean}]\ \frac{\Gamma \vdash_{\text{exp}} e_1 : \mathsf{boolean} \qquad \Gamma \vdash_{\text{exp}} e_2 : \mathsf{boolean} \qquad \diamond \in \{\texttt{and},\ \texttt{or}\}}{\Gamma \vdash_{\text{exp}} e_1\ \diamond\ e_2 : \mathsf{boolean}}$$

$$[\textsc{tComparisons}]\ \frac{\diamond \in \{<,\ >,\ <=,\ >=,\ ==,\ !=\} \qquad \Gamma \vdash_{\text{exp}} e_1 : T \qquad \Gamma \vdash_{\text{exp}} e_2 : T}{\Gamma \vdash_{\text{exp}} e_1\ \diamond\ e_2 : \mathsf{boolean}} \qquad [\textsc{tNot}]\ \frac{\Gamma \vdash_{\text{exp}} e : \mathsf{boolean}}{\Gamma \vdash_{\text{exp}} \texttt{not}\ e : \mathsf{boolean}}$$

Figure 3.4: Expression typing

3.3.1 Expression Typing

The *expression typing* rules are used to analyse expressions, e, as defined in Figure 3.3. These rules, adapted from [44], are defined in Figure 3.4, where they use the judgement:



This judgement states, that "expression e has type T, subject to the *variable binding* environment Γ." From Figure 3.4, the [tLiteral] rule analyses basic

values by comparing their type to the expected type. Their type is obtained using the function type, *e.g.*, $\mathsf{type}(5) = \mathsf{number}$.

Definition 3.3 (*Type*)**.**

$$\mathsf{type}(\mathit{boolean}) = \mathsf{boolean} \qquad \mathsf{type}(\mathit{number}) = \mathsf{number}$$

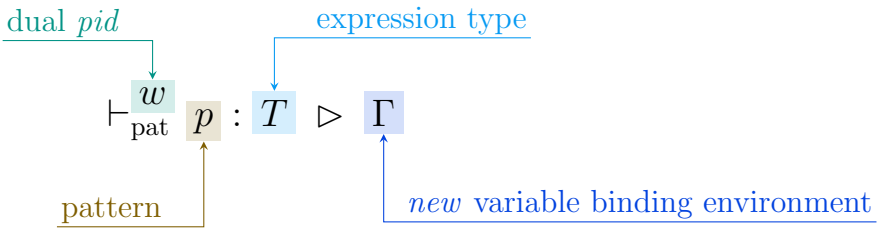
$$\mathsf{type}(\mathit{atom}) = \mathsf{atom} \qquad \mathsf{type}(\iota) = \mathsf{pid}, \text{ where } \iota \text{ is a } \mathit{pid} \text{ instance} \quad \blacksquare$$

Rule [tVariable] checks that variables have the correct types, as assumed in the *variable binding* environment Γ. Rule [tTuple] inspects tuples, by checking that each element in the tuple has the correct type, *e.g.*, the type of $\{2, \mathit{true}\}$ should be $\{\mathsf{number}, \mathsf{boolean}\}$. Rules [tEList] and [tList] check the types of empty and non-empty lists, respectively. Rule [tEList] is the only polymorphic rule, meaning that the empty list matches with any list type, *i.e.*, $[\,]$ has type $[T]$ for any T.

Rules [tArithmetic], [tBoolean] and [tComparisons] analyse the types of binary operations. For instance, [tArithmetic] ensures the expressions in arithmetic operations have type number. Lastly, [tNot] checks the type of the negation operation.

3.3.2 Pattern Typing

In the *receive* construct (Figure 3.3), messages are compared to the patterns in each branch. When one pattern matches, the values from the messages are bound to new variables. Similarly, the *case* construct may also produce new variables as a result of pattern matching. These new variables are mapped to their types using the *pattern typing* rules, shown in Figure 3.5, using the judgement



which is read as, "pattern p is matched to type T, where it produces new variables

$$\boxed{\vdash^{w}_{\text{pat}} p : T \ \triangleright\ \Gamma}$$

$$[\textsc{tpLiteral}]\ \frac{\emptyset \vdash_{\text{exp}} b : T \qquad b \neq [\,]}{\vdash^{w}_{\text{pat}} b : T \ \triangleright\ \emptyset} \qquad [\textsc{tpVariable}]\ \frac{x \neq w}{\vdash^{w}_{\text{pat}} x : T \ \triangleright\ x : T}$$

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

$$[\textsc{tpList}]\ \frac{\vdash^{w}_{\text{pat}} w_1 : T \ \triangleright\ \Gamma_1 \qquad \vdash^{w}_{\text{pat}} w_2 : [T] \ \triangleright\ \Gamma_2}{\vdash^{w}_{\text{pat}} [\,w_1 \mid w_2\,] : [T] \ \triangleright\ \Gamma_1, \Gamma_2} \qquad [\textsc{tpEList}]\ \frac{}{\vdash^{w}_{\text{pat}} [\,] : [T] \ \triangleright\ \emptyset}$$

Figure 3.5: Pattern typing

and their types are collected Γ; under the assumption that the variable containing the dual *pid*, w, remains unchanged."

Rule [tpLiteral] acts as a pattern checking mechanism, ensuring that the literals match the value obtained. Rule [tpVariable] introduces new (and unique) variables from the pattern. It also prevents variables from clashing with the dual *pid* identifier w. Rule [tpTuple] is used pattern match tuples, and rules [tpEList] and [tpList] are used to pattern match lists. Note that, new variable mappings are joined together using Γ_1, Γ_2, where their domains must be distinct.

Example 3.4. From the auction example (Listing 1.3), recall this snippet, where we introduce a new variable called `value` in one of the branches:

```
receive do
  ...
  {:higher, value} -> ...
end
```

From the session type *auction*, we can infer that this branch matches with the session type ?`higher`(number). Therefore, we can apply the axiom [tpVariable], as follows:

$$\vdash^{w}_{\text{pat}} \texttt{value} : \mathsf{number} \ \triangleright\ \texttt{value} : \mathsf{number}$$

This results in a new environment, binding the variable `value` to type number. ■

3.3.3 Term Typing

Expressions are used within terms, as defined in Figure 3.3. In this section, we typecheck these terms with respect to their base types, along with the added session types. In addition to the *variable binding* environment (Γ) defined in Section 3.3, the type system uses two further environments:

$$\textit{session typing env.} \quad \Delta ::= \emptyset \mid \Delta,\ f_n : S$$

$$\textit{function inf. env.} \quad \Sigma ::= \emptyset \mid \Sigma,\ f_n : \left\{ \begin{array}{l} \texttt{params} = \widetilde{x},\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{body} = t,\ \texttt{return\_type} = T,\ \texttt{dual} = y \end{array} \right\}$$

The *session typing* environment, Δ, maps function names and arity to their session type ($f_n : S$). Similar to Γ, we can extend Δ, or obtain its domain (*i.e.,* $dom(\Delta)$). If a function f_n has a *known* session type, then it can be found in Δ, *i.e.,* $f_n \in dom(\Delta)$. Each module has a *function information* environment, Σ, that holds information related to the function definitions – once created, Σ remains *static*. For a function f, with arity n, $\Sigma(f_n)$ returns the tail list of parameters (`params`) and their types (`param_types`), the function's body (`body`), and its return type (`return_type`). It also returns the variable name that holds the dual process' *pid* (`dual`), which we reserved as the first parameter in Figure 3.3.

Definition 3.4 (*Well-Formedness of* Σ)**.** The *function information* environment, Σ, is *well-formed* iff all functions that are mapped in this environment ($f_n \in dom(\Sigma)$) observe the following condition:

$$\mathbf{fv}(\Sigma(f_n).\texttt{body}) \setminus \Sigma(f_n).\texttt{params} \setminus \Sigma(f_n).\texttt{dual} = \emptyset$$

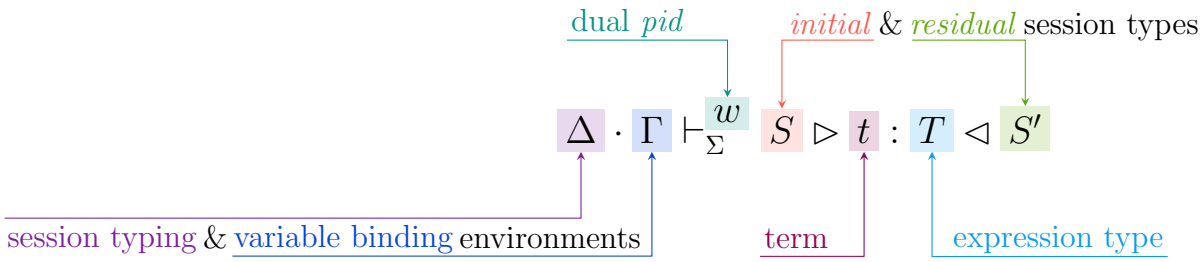
Informally, this condition states that the body of function f_n is *closed* (Definition 3.2), if all of its parameters (*i.e.,* `params` and `dual`) are bound. ■

Example 3.5. Recall the function `successor` from Example 3.3:

t
```
def successor(y, num) do
  x = num + 1
  send(y, {:val, x})
end
```

We saw that the only free variables in the body t were `y` and `num`, *i.e.,* $\mathbf{fv}(t) = \{\texttt{y}, \texttt{num}\}$, given that $t = \Sigma(\texttt{successor}_2).\texttt{body}$. The free variables, `y` and `num`, are equivalent to the variables stored in `dual` and `params`, respectively. Thus, when we bind `y` and `num` to a value, the function's body becomes *closed* (*i.e.,* $\mathbf{fv}(t) = \emptyset$). This pattern is enforced by the *Well-Formedness of* Σ Definition. ■

Figure 3.6 presents the syntax-directed *term typing* rules, where a rule is assigned to each possible term t. The rules use a similar notation formulated by Harvey *et al.* [45]. The term typing rules are based on the following term typing judgement:



This judgement states that "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', while interacting with a dual process with *pid* identifier w. This typing is valid under some *session typing* environment Δ, *variable binding* environment Γ and *function information* environment Σ." Since the *function information* environment Σ is static for the whole module (and by extension, for all sub-terms), it is left implicit in the rules of Figure 3.6.

$$\boxed{\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'}$$

$$[\textsc{tLet}]\ \frac{\Delta \cdot \Gamma \vdash^{w} S \rhd t_1 : T' \lhd S'' \qquad \Delta \cdot (\Gamma, x : T') \vdash^{w} S'' \rhd t_2 : T \lhd S' \qquad x \neq w}{\Delta \cdot \Gamma \vdash^{w} S \rhd x = t_1;\ t_2 : T \lhd S'}$$

$$[\textsc{tBranch}]\ \frac{\begin{array}{c}\forall i \in I \qquad \forall j \in 1..n \\ \vdash^{w}_{\text{pat}} p_i^j : T_i^j \ \rhd\ \Gamma_i^j \qquad \Delta \cdot \left(\Gamma, \Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash^{w} S_i \rhd t_i : T \lhd S'\end{array}}{\Delta \cdot \Gamma \vdash^{w} \&\left\{?\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i \in I} \rhd \mathtt{receive\ do}\ \left(\left\{:\mathtt{l}_i, \widetilde{p_i}\right\} \to t_i\right)_{i \in I} \mathtt{end} : T \lhd S'}$$

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

$$[\textsc{tRecKnownCall}]\ \frac{\begin{array}{c}\Sigma\left(f_n\right) = \Omega \qquad \Omega.\mathtt{return\_type} = T \qquad \Omega.\mathtt{param\_types} = \widetilde{T} \\ \Delta\left(f_n\right) = S \qquad \forall i \in 2..n \qquad \Gamma \vdash_{\text{exp}} e_i : T_i\end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd f\left(w,\ e_2,\ \ldots,\ e_n\right) : T \lhd \mathsf{end}}$$

$$[\textsc{tRecUnknownCall}]\ \frac{\begin{array}{c}\Sigma\left(f_n\right) = \Omega \qquad f_n \notin dom(\Delta) \qquad \Omega.\mathtt{dual} = y \\ \Omega.\mathtt{params} = \widetilde{x} \quad \Omega.\mathtt{param\_type} = \widetilde{T} \quad \Omega.\mathtt{body} = t \quad \Omega.\mathtt{return\_type} = T \\ (\Delta, f_n : S) \cdot \left(\Gamma, y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\right) \vdash^{y} S \rhd t : T \lhd S' \\ \forall i \in 2..n \quad \Gamma \vdash_{\text{exp}} e_i : T_i\end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd f\left(w,\ e_2,\ \ldots,\ e_n\right) : T \lhd S'}$$

$$[\textsc{tCase}]\ \frac{\begin{array}{c}\Gamma \vdash_{\text{exp}} e : U \\ \forall i \in I \qquad \vdash^{w}_{\text{pat}} p_i : U \ \rhd\ \Gamma_i' \qquad \Delta \cdot (\Gamma, \Gamma_i') \vdash^{w} S \rhd t_i : T \lhd S'\end{array}}{\Delta \cdot \Gamma \vdash^{w} S \rhd \mathtt{case}\ e\ \mathtt{do}\ \left(p_i \to t_i\right)_{i \in I} \mathtt{end} : T \lhd S'}$$

$$[\textsc{tExpression}]\ \frac{\Gamma \vdash_{\text{exp}} e : T}{\Delta \cdot \Gamma \vdash^{w} S \rhd e : T \lhd S}$$

Figure 3.6: Term typing

Session types can describe sending and receiving actions, along with recursion and termination. The sending and receiving actions are covered by the [tBranch] and [tChoice] rules. The recursive parts (via function calls), are inspected using rules [tRecUnknownCall] and [tRecKnownCall].

Consider the most complex rule, [tBranch], which typechecks the **`receive`** construct. Rule [tBranch] expects an initial branching session type $\&\{\ldots\}$, where it is compared to a **`receive`** construct. Each branch in the session type must match with a corresponding branch in the **`receive`** construct, where *both* the labels ($\mathtt{l}_i$) and payload types ($\widetilde{T_i}$) must match. The types within each **`receive`** branch are computed using the pattern typing judgement ($\vdash^{w}_{\text{pat}} p : T \triangleright \Gamma$), defined in Figure 3.5 (see Example 3.4). Each **`receive`** branch is then checked recursively using the term typing rules. Each branch produces a value having common type T and a common residual session type S' – this makes it possible to utilise the *fork-join* pattern described in Section 5.3.

The **`send`** statement is typechecked using the [tChoice] rule. This rule expects an initial choice session type $\oplus\{\ldots\}$, which is then compared to the message in the **`send`** statement. The label inside the message must match with a single label ($\mathtt{l}_i$) offered by the choice session type. The payloads of the message must also match with the types expected in the session type ($\widetilde{T_i}$). The types of the payloads are computed using the expression typing rules, described in Figure 3.4. Furthermore, the **`send`** statement also includes the *pid* of the addressee of the message – this *pid* must match with the dual *pid* (w) which is imposed from the rule itself. This ensures that messages are only sent to the correct addressee.

Remark. *We use a notion of a function with a* 'known' *session type, or an* 'unknown' *session type. Concretely, a function f_n has a* known *session type if f_n is mapped to a session type S in the session typing environment Δ, i.e., $f_n \in dom(\Delta)$ or $f_n : S \in \Delta$. Conversely, a function f_n has an* unknown *session type, if this function does not exists in the* session typing *environment, i.e., $f_n \notin dom(\Delta)$.* ■

Function calls are typechecked using the rules [tRecKnownCall] and [tRecUnknownCall], depending whether the functions have a *known* or *unknown* session type, respectively. For instance, public functions always have a *known* session type, since they are decorated explicitly using the `@session` or `@dual` annotations, as define in Figure 3.3.

In case of a function f_n with a *known* session type, the rule [tRecKnownCall] verifies that the expected initial session type is equivalent to the function's *known* session type. Its session type is obtained from the *session typing* environment, *i.e.,* $\Delta(f_n) = S$. Without typechecking the function's body, this rule ensures that the parameters have the correct types (using the expression typing rules), and it ascertains that this session type S is fully consumed, thus the residual type becomes end.

A function, f_n, with an *unknown* session type, is inspected using the [tRecUnknownCall] rule. This rule ensures that the parameters have the correct types, and then it analyses the function's body (obtained from Σ) with respect to a new session type. This session type is inherited from the expected initial session type, since the function being called must fully exhaust the remaining session type. Furthermore, this session type is appended to the *session typing* environment Δ for future reference, *i.e.,* $\Delta' = (\Delta, f_n{:}S)$. Since Δ' now contains the session type of f_n, should f_n be is recursively called again (within the same call chain), rule [tRecKnownCall] will take precedence over [tRecUnknownCall], thus bypassing the need to re-analyse its body.

The remaining three rules analyse the functional aspect of the code. Rule [tLet], typechecks a *let* statement, $x = t_1;\ t_2$, where t_1 has an initial session type S and a residual session type S', and t_2 has an initial session type is S' and a residual session type S''. The overall session type progresses from S to S'.

Rule [tCase] typechecks a case construct, by first analysing the type of the expression being matched, and then obtains the patterns in each branch using the pattern typing rules. Similar to [tBranch], all cases need to end up with a

common type T and residual session type S'.

Finally, rule [tExpression] analyses expressions using the expression typing rules. Since expressions produce no side-effects, the residual session type remains the same as the initial session type.

3.3.4 Module Typing

Session typechecking is initiated by analysing an Elixir module (M), using the judgement $\vdash M$. A module is typechecked by inspecting each of its public functions, ascertaining that they correspond and fully consume some pre-defined session type – this is depicted in the [tModule] rule:

$$[\text{tModule}]\ \frac{\begin{array}{c}\forall f_n \in \texttt{functions}(\widetilde{D}) \quad \Sigma_{f_n}(f_n) = \Omega \quad \Sigma = \texttt{details}(\widetilde{P}) \\ \Omega.\texttt{params} = \widetilde{x} \quad \Omega.\texttt{param\_types} = \widetilde{T} \quad \Omega.\texttt{body} = t \quad \Omega.\texttt{return\_type} = T \quad \Omega.\texttt{dual} = y \\ S = \texttt{session}(f_n) \qquad \left(f_n : S\right) \cdot \left(\widetilde{x} : \widetilde{T}\right) \vdash^{y}_{\Sigma \cup \Sigma_{f_n}} S \rhd t : T \lhd \texttt{end}\end{array}}{\vdash \texttt{defmodule}\ m\ \texttt{do}\ \widetilde{P}\ \widetilde{D}\ \texttt{end}}$$

The [tModule] rule uses three helper functions: `functions`, `session` and `details` (defined in pages 116–117). The function $\texttt{functions}(\widetilde{D})$ returns a list of all function names (and arity) of the public functions ($\widetilde{D}$) that are going to be checked individually. The function $\texttt{session}(f_n)$ obtains the session type S for some function f_n, either immediately from `@session`, or by computing the dual session type from `@dual`. When function f_n executes, it must follow session type S (obtained from $\texttt{session}(f_n)$). Finally, the function $\texttt{details}(\widetilde{P})$ populates the *function information* environment (Σ) with details about the *private* functions ($\widetilde{P}$) within the module. Moreover, [tModule] uses Σ_{f_n}, which implicitly contains information about the current public function being checked.

After getting the information required from these three auxiliary functions, [tModule] then inspects public function individually (shown by the orange box), using the *term typing* judgement explained in the preceding section (Section 3.3.3).

The judgement compares the public functions' body to their session type, ensuring that the session types are adhered to precisely.

When each public function f_n is analysed (refer to the orange box), its *function information* environment is set to $\Sigma \cup \Sigma_{f_n}$. This joint environment contains information about all private functions ($\Sigma = \texttt{details}(\widetilde{P})$) and information about f_n itself (in Σ_{f_n}). The latter creates a restriction, where calls to other public functions are prohibited, following the pattern described earlier in Figure 3.1. So, we can only have *(i)* calls to any private function, and *(ii)* recursive calls to the public function being analysed, f_n, given that they comply with the expected session type.

Furthermore, when typechecking each public function f_n against its annotated session type S (refer to the orange box), the initial *session typing* environment Δ is set to contain only one mapping, *i.e.*, $\Delta = (f_n : S)$. Thus, all initial private functions start with an *unknown* session type (since these functions are not annotated with a session type). This means that private function calls need to be typechecked using the [tRecUnknownCall], where they are assigned a session type. When they are called (recursively) again, they have to subscribe to the session type binding that they were initially assigned, following the rule [tRecKnownCall] as explained in Section 3.3.3.

3.4 Typing in Action

We reconsider the auction example from Chapter 1 and typecheck it using the type system that we just defined in Section 3.3. The `Auction` module, from Listing 1.3, has two public functions, `buyer` and `auctioneer`, which should follow the *auction* and $\overline{\textit{auction}}$ protocols, respectively.

Typechecking is initiated by the [tModule] rule ($\vdash M$, Section 3.3.4), where

the module is typechecked using the judgement:

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

In this case, $\widetilde{D}$ contains the two public functions (*i.e.,* `buyer` and `auctioneer`), and $\widetilde{P}$ contains one private function (*i.e.,* `decide`). The premise of [tModule] dictates that each function should be typechecked using the term typing judgement $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$ from Figure 3.6. It also states that each function (f_n) should follow its initial session type S (obtained from $\texttt{session}(f_n)$), and reach the residual session type (S', which set to end by [tModule]). The latter ensures that if a well-typed function terminates, then it fully consumes the initial session type S. We focus on the function `buyer` ($f_n = \texttt{buyer}_2$), which should follow the *auction* session type. We start by typechecking its body, t:

t
```
send(auctioneer_pid, {:bid, amount})

receive do
  {:sold}           -> :yay
  {:higher, value} -> decide(auctioneer_pid, amount, value)
end
```

Since this body t contains multiple sequential statements, it is typechecked using the [tLet] rule (Figure 3.6), as follows:

$$\Delta \cdot \Gamma \vdash^{w} !\texttt{bid}(\mathsf{number}).S_1 \triangleright t : \mathsf{atom} \triangleleft \mathsf{end}$$

where $w =$ `auctioneer_pid`, $\Delta = (\texttt{buyer}_2 : \mathit{auction})$, and

$$S_1 = \& \left\{ \begin{array}{l} ?\texttt{sold}().\mathsf{end}, \\ ?\texttt{higher}(\mathsf{number}).\ \oplus \left\{ \begin{array}{l} !\texttt{quit}().\mathsf{end}, \\ !\texttt{continue}().\mathit{auction} \end{array} \right. \end{array} \right\}$$

The term t is split in two statements; a send statement, followed by a receive construct. The [TLET] rule typechecks them in order, so the send statement is first typechecked using the [TCHOICE] rule, as follows:

$$\Delta \cdot \Gamma \vdash^{w} !\texttt{bid}(\mathsf{number}).S_1 \triangleright \texttt{send(auctioneer\_pid, \{:bid, amount\})} : \{\mathsf{atom}, \mathsf{number}\} \triangleleft S_1$$

This [TCHOICE] rule dictates that the label being sent must match with one of the labels offered by the choice session type, in this case the session type only offers a single choice (*i.e.,* $!\texttt{bid}(\mathsf{number})$). Moreover, the addressee of this message, `auctioneer_pid`, must also match with the *pid* of the dual process, w. Having verified these details, the residual session type results in S_1, meaning that the subsequent receive statement must adhere to the initial session type S_1 and residual session type end. The receive construct is then typechecked using [TBRANCH]:

$$\Delta \cdot \Gamma \vdash^{w} \& \left\{ \begin{array}{l} ?\texttt{sold}().\mathsf{end}, \\ ?\texttt{higher}(\mathsf{number}).S_2 \end{array} \right\} \triangleright \left[\begin{array}{l} \texttt{receive do} \\ \quad \texttt{\{:sold\}} \texttt{ -> :yay} \\ \quad \texttt{\{:higher, value\}} \texttt{ -> } t_2 \\ \texttt{end} \end{array} \right] : \mathsf{atom} \triangleleft \mathsf{end}$$

where $t_2 = \texttt{decide(auctioneer\_pid, amount, value)}$. The branching session type in this rule dictates that two branches are required, labelled `sold` and `higher`. The branch the with label `higher` introduces a new variable called `value`, extending Γ, and by Example 3.4 we know that the variable `value` has type number. Furthermore, the terms inside each branch must match with a corresponding continuation branch, *i.e.,* end and S_2, respectively. The first branch (`:yay`) matches immediately with the session type end, since there are no further interactions. The second branch, t_2, must be compared with the initial session type

S_2:

$$S_2 = \oplus \left\{ \begin{array}{l} !\texttt{quit}().\mathsf{end}, \\ !\texttt{continue}().\mathit{auction} \end{array} \right\}$$

Since t_2 is a call to a private function, called $\texttt{decide}_3$, which has an unknown session type (*i.e.,* $\texttt{decide}_3 \notin \mathit{dom}(\Delta)$), then the function's body must be typechecked using the [tRecUnknownCall] judgement, as follows, where it inherits the remaining session type S_2:

$$\Delta \cdot \Gamma \vdash^{w} S_2 \rhd \texttt{decide(auctioneer\_pid, amount, value)} : \mathsf{atom} \lhd \mathsf{end}$$

The body of the function `decide` can be typechecked using the [tChoice] rule, since the S_2 session type offers a choice to send a label (`quit` or `continue`). This should successfully terminate the typechecking for the `buyer` function, since the initial session type *auction* becomes fully exhausted. This [tChoice] rule is the last check required, thus declaring the `buyer` function as well-typed. The remaining function, `auctioneer`, should also be typechecked in a similar manner, to ensure that it is obeys $\overline{\mathit{auction}}$.

As another example, we consider the first line from the problematic `buyer` function (from Listing 1.2) and attempt to typecheck it with respect to the *auction* protocol. This `buyer` function (shown below) attempts to send a `bid` accampanied by a *true* value:

```
send(auctioneer_pid, {:bid, true})
# ...
```

This can be typechecked using [tChoice]:

$$\Delta \cdot \Gamma \vdash^{w} !\texttt{bid}(\mathsf{number}).S_1 \rhd \texttt{send(auctioneer\_pid, \{:bid, true\})} : \mathsf{atom} \lhd S_1$$

The [tChoice] rule matches both the label and the types of the payloads. So, if we compare the message $\{\texttt{:bid, true}\}$ to the expected session type $!\texttt{bid}(\mathsf{number})$, we

can notice that the payload (*i.e., true*) is a boolean rather than a number. Thus, typechecking fails, flagging this as an ill-typed function.

3.5 Semantics

The final part of the formal analysis involves formulating a transition semantics (or operational semantic) of our language defined in Section 3.2. The semantics show how a program executes step-by-step, thus modelling the runtime behaviour of a typical Elixir program. Using these transition semantics, we will be able to validate our type system further, which we will then see in detail in the subsequent chapter.

From Figure 3.1 we know that modules consist of several functions. To execute these functions, we can spawn them, which results in a new process executing their body. A body t executes by *transitioning* from one state to another, where it produces some action α as a result. Concretely, we write this as $t \xrightarrow{\alpha} t'$, meaning that a term t transitions to a new term t', producing an action α. This action α can take the following forms (from Act):

$$
\begin{array}{rcll}
\alpha \in \textsc{Act} & ::= & \iota!\{\texttt{:l},\ \widetilde{v}\} & \text{Send message} \\
& \mid & ?\{\texttt{:l},\ \widetilde{v}\} & \text{Receive message} \\
& \mid & f_n & \text{Function call} \\
& \mid & \tau & \text{Silent}
\end{array}
\quad
\begin{array}{l}
\left.\vphantom{\begin{array}{l}a\\b\end{array}}\right\} \text{external} \\
\left.\vphantom{\begin{array}{l}a\\b\end{array}}\right\} \text{internal}
\end{array}
$$

The actions (α) can produce an *external* side-effect, *i.e.,* a message is sent ($\iota!\{\texttt{:l},\ \widetilde{v}\}$) or received ($?\{\texttt{:l},\ \widetilde{v}\}$) and we discipline these external side-effects using session types. Additionally, there are *internal* actions, *i.e.,* a function call (f_n), and a *silent* transition (τ), which have no observable side-effects from external processes.

We formulate the transitions from t to t' (producing α) in a form of a *labelled transition system* (LTS) [34], described in Definition 2.1. Our LTS consists of a

$$\boxed{t \xrightarrow[\Sigma]{\alpha} t'} \quad [\textsc{rLet}_1]\ \frac{t_1 \xrightarrow{\alpha} t_1'}{x = t_1;\ t_2 \xrightarrow{\alpha} x = t_1';\ t_2} \qquad [\textsc{rLet}_2]\ \frac{}{x = v;\ t \xrightarrow{\tau} t\,[^{v}/_{x}]}$$

$$[\textsc{rChoice}_1]\ \frac{e_k \to e_k'}{\texttt{send}\,(\iota, \{\texttt{:l}, v_1,\ \ldots,\ v_{k-1},\ e_k,\ \ldots,\ e_n\}) \xrightarrow{\tau} \texttt{send}\,(\iota, \{\texttt{:l}, v_1,\ \ldots,\ v_{k-1},\ e_k',\ \ldots,\ e_n\})}$$

$$[\textsc{rChoice}_2]\ \frac{}{\texttt{send}\,(\iota, \{\texttt{:l}, v_1,\ \ldots,\ v_n\}) \xrightarrow{\iota!\{\texttt{:l}, v_1, \ldots, v_n\}} \{\texttt{:l}, v_1,\ \ldots,\ v_n\}}$$

$$[\textsc{rBranch}]\ \frac{\exists j \in I \qquad \texttt{l}_j = \texttt{l}_i \qquad \mathbf{match}(\widetilde{p_j}, \widetilde{v}) = \sigma}{\texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I}\texttt{end} \xrightarrow{?\{\texttt{:l}_j, \widetilde{v}\}} t_j\sigma}$$

$$[\textsc{rCall}_1]\ \frac{e_k \to e_k'}{f\,(v_1,\ \ldots,\ v_{k-1},\ e_k,\ \ldots,\ e_n) \xrightarrow{\tau} f\,(v_1,\ \ldots,\ v_{k-1},\ e_k',\ \ldots,\ e_n)}$$

$$[\textsc{rCall}_2]\ \frac{\Sigma\,(f_n) = \Omega \qquad \Omega.\texttt{body} = t \qquad \Omega.\texttt{params} = x_2,\ \ldots,\ x_n \qquad \Omega.\texttt{dual} = y}{f\,(\iota, v_2,\ \ldots,\ v_n) \xrightarrow{f_n} t\,[^{\iota}/_{y}]\,[^{v_2,\ \ldots,\ v_n}/_{x_2,\ \ldots,\ x_n}]}$$

$$[\textsc{rCase}_1]\ \frac{e \to e'}{\texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} \xrightarrow{\tau} \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end}}$$

$$[\textsc{rCase}_2]\ \frac{\exists j \in I \qquad \mathbf{match}(p_j, v) = \sigma}{\texttt{case}\ v\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} \xrightarrow{\tau} t_j\sigma} \qquad [\textsc{rExpression}]\ \frac{e \to e'}{e \xrightarrow{\tau} e'}$$

Figure 3.7: Term transition semantic rules

triple $\langle(\textsc{Terms}, \Sigma), \textsc{Act}, \to\rangle$, where Terms represents the set containing all forms of terms, *i.e.,* $t, t' \in \textsc{Terms}$. The transition $t \xrightarrow[\Sigma]{\alpha} t'$ is a shorthand notations for the relation $\to \subseteq ((\textsc{Terms}, \Sigma) \times \textsc{Act} \times ((\textsc{Terms}, \Sigma))$. Note that, transitions use information from a well-formed *function information* environment (Σ), which we leave implicit in the transitions, since Σ remains static during transitions. Therefore, we use the notation $t \xrightarrow{\alpha} t'$ for transitions.

The transitions are defined by the *term* transition semantic rules, depicted in Figure 3.7. We start from the most straightforward rules, [rLet$_1$] and [rLet$_2$], which deal with the evaluation of a *let* statement, $x = t_1;\ t_2$. We use a *call-by-*

value semantic, where the first term t_1 transitions fully to a value, before being substituted for x in t_2, using the *Variable Substitution* Definition (Definition A.7). *Substitution* is a partial function that swaps variables for their actual values. Substitutions are denoted as $[v/x]$, or $[v_1, v_2/x_1, x_2]$ in case of multiple substitutions. For example, the expression $(2 + x)\,[4/x]$ results in $2 + 4$ after the substitutions.

The *send* statement, $\texttt{send}\,(\iota, \{\texttt{:l}, e_1, \ \ldots, \ e_n\})$, evaluates by first reducing each part of the message to a value, starting from the leftmost expression. So first, expression e_1 is reduced to v_1, until the rightmost expression, e_n, is finally reduced to v_n. This is carried out in [RCHOICE$_1$] using the expression reduction rules defined in Figure 3.8. Rule [RCHOICE$_1$] produces no side-effects, so the action α is set to τ (*i.e.,* a *silent* transition). Then, when the whole message is reduced to a value $\{\texttt{:l}, v_1, \ \ldots, \ v_n\}$, rule [RCHOICE$_2$] is used to perform the actual message sending operation, resulting in action $\alpha = \iota!\,\{\texttt{:l}, v_1, \ \ldots, \ v_n\}$, where ι is the *pid* value of the addressee.

The *receive* construct, $\texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I}\texttt{end}$, is evaluated in [RBRANCH]. When a message is received (*i.e.,* $\alpha = ?\,\{\texttt{:l}_j, \ \widetilde{v}\}$), it is matched with a valid branch from the *receive* construct, using the label $\texttt{:l}_j$ as the key. Then, the payload of the message ($\widetilde{v}$) is compared to the patterns in the selected branch ($\widetilde{p_j}$) using $\mathbf{match}(\widetilde{p_j}, \widetilde{v})$. This **match** function (Definition 3.5) produces some substitutions σ, mapping the newly defined variables (from $\widetilde{p_j}$) to their values. The substitutions σ are used to swap the free variables in t_j with their values.

Definition 3.5 (*Pattern Matching*)**.** The **match** function pairs patterns with a corresponding value, resulting in a sequence of substitutions (called σ), *e.g.* $\mathbf{match}(p, v) = [v_1/x_1]\,[v_2/x_2] = [v_1, v_2/x_1, x_2]$. Note that, a sequence of **match** outputs are combined together, where the empty substitutions (*i.e.,* $[\,]$) are ignored.

$$\mathbf{match}(\widetilde{p}, \widetilde{v}) = \mathbf{match}(p_1, v_1), \ \ldots, \ \mathbf{match}(p_n, v_n) \qquad \textit{where } \widetilde{p} = p_1, \ \ldots, \ p_n \textit{ and } \widetilde{v} = v_1, \ \ldots, \ v_n$$

$$\mathbf{match}(p, v) = \begin{cases} [\,] & p = b, v = b \text{ and } p = v \\ [v/x] & p = x \\ \mathbf{match}(w_1, v_1),\ \mathbf{match}(w_2, v_2) & p = [\,w_1 \mid w_2\,], v = [\,v_1 \mid v_2\,] \\ \mathbf{match}(w_1, v_1),\ \ldots,\ \mathbf{match}(w_n, v_n) & p = \{w_1,\ \ldots,\ w_n\} \text{ and } v = \{v_1,\ \ldots,\ v_n\} \end{cases} \blacksquare$$

Definition 3.6 (*Variable Patterns*)**.** Computes an ordered set of variables from a given pattern p.

$$\mathbf{vars}(\widetilde{p}) = \mathbf{vars}(p_1,\ \ldots,\ p_n) = \mathbf{vars}(p_1) \cup \cdots \cup \mathbf{vars}(p_n)$$

$$\mathbf{vars}(p) = \begin{cases} \emptyset & p = b \\ \{x\} & p = x \\ \mathbf{vars}(w_1) \cup \mathbf{vars}(w_2) & p = [\,w_1 \mid w_2\,] \\ \cup_{i \in 1..n} \mathbf{vars}(w_i) & p = \{w_1,\ \ldots,\ w_n\} \end{cases} \blacksquare$$

Example 3.6. Consider the pattern $p_1 = \{x, 2, y\}$ and the value $v_1 = \{8, 2, \mathit{true}\}$. By the **vars** definition, we can extract the variables from p_1, *i.e.,* $\mathbf{vars}(p_1) = \{x, y\}$. We can also use the **match** definition to assign values to these variables. In this case, $\mathbf{match}(p_1, v_1) = \sigma$ where $\sigma = [8/x]\,[\mathit{true}/y]$ or $\sigma = [8, \mathit{true}/x, y]$.
The **match** definition is also used to enforce correctness of the values. *E.g.*, we replace p_1 by the new pattern $p_2 = \{x, 2, \mathit{false}\}$. In this case, $\mathbf{match}(p_2, v_1)$ fails, since p_2 expects a *false* value as the third element, but finds a *true* value instead. $\blacksquare$

A function call is evaluated in [rCall$_1$], by reducing all of its parameters to a value, using the expression reduction rules. Then, in [rCall$_2$], a function f with arity n produces an internal action depicting the function's name, *i.e.,* $\alpha = f_n$. This is used to fetch the function's parameter name and body from Σ. The body

$\boxed{e \to e'}$

$$[\textsc{reOperation}_1]\ \frac{e_1 \to e_1'}{e_1 \ \diamond \ e_2 \to e_1' \diamond e_2} \qquad [\textsc{reOperation}_2]\ \frac{e_2 \to e_2'}{v_1 \ \diamond \ e_2 \to v_1 \diamond e_2'}$$

$$[\textsc{reOperation}_3]\ \frac{v = v_1 \ \diamond \ v_2}{v_1 \ \diamond \ v_2 \to v} \qquad [\textsc{reNot}_1]\ \frac{e \to e'}{\texttt{not}\ e \to e'} \qquad [\textsc{reNot}_2]\ \frac{v' = \neg v}{\texttt{not}\ v \to v'}$$

$$[\textsc{reList}_1]\ \frac{e_1 \to e_1'}{[\,e_1 \mid e_2\,] \to [\,e_1' \mid e_2\,]} \qquad [\textsc{reList}_2]\ \frac{e_2 \to e_2'}{[\,v_1 \mid e_2\,] \to [\,v_1 \mid e_2\,]}$$

$$[\textsc{reTuple}]\ \frac{e_k \to e_k'}{\{v_1,\ \ldots,\ v_{k-1},\ e_k,\ \ldots,\ e_n\} \to \{v_1,\ \ldots,\ ,v_{k-1},\ e_k',\ \ldots,\ e_n\}}$$

Figure 3.8: Expression semantic rules

transitions further after substituting the parameters with their values.

A case construct is evaluated in [$\textsc{rCase}_1$], by first reducing the expression which is being matched. Then, [$\textsc{rCase}_2$] matches the value with the correct branch, using the **match**, akin to [rBranch]. Finally [rExpression] reduces an expression using the expression semantic rules.

The expression semantic rules are defined in Figure 3.8, which use the form $e \to e'$, where e is reduced to e', producing no internal or external actions. Binary operations are reduced in [$\textsc{reOperation}_1$] and [$\textsc{reOperation}_2$], and then evaluated in [$\textsc{reOperation}_3$]. The negation operation is similarly reduced in [$\textsc{reNot}_1$] and [$\textsc{reNot}_2$]. Tuples and lists are reduced in [reTuple], [$\textsc{reList}_1$] and [$\textsc{reList}_2$].

3.6 Conclusion

In this chapter, we formalised a core Elixir language and created a session typing system for it. This type system typechecks functions with respect to an interaction protocol, which we define explicitly via session types. This chapter fulfils the first part of Objective O1, since we provided a formal foundation for Elixir programs

in the form of a type system and transitional semantics. The remaining part of this objective is addressed in the following chapter (Chapter 4), where the type system is validated from a formal perspective. In Chapter 5, this type system is implemented as a proof-of-concept implementation in Elixir.

4. Metatheory

In Chapter 3 we formalised a type system that verifies whether (public) functions follow their session endpoint specification. This chapter addresses the remaining part of Objective O1, where the type system is validated by proving its formal properties.

We analyse these properties by associating the static session typing rules (from Section 3.3) with the transition semantics (from Section 3.5). We do this by proving type preservation, which states that if a well-typed term transitions, the resulting term then remains well-typed [31]. Given that we are dealing with a *session* typing system, and not just an ordinary functional typing system, we extend this preservation property to prove *session fidelity* – which states that if a well-(*session*-)typed term follows an initial session type S and transitions with action α, then the resulting term remains well-typed with respect to S'. This session type S' is obtained by from S and α. This ensures that the program will not encounter behavioural errors (*e.g.*, certain deadlocks or communications mismatches) when executing.

To accomplish this, we define some auxiliary propositions before forming the Session Fidelity Theorem. We analyse the *dynamic* type system (defined in Section 3.5); bridging closed terms with the transition semantics, leading to the Closed Term Proposition (Proposition 6). Following this, we discuss the *static* type system (defined in Section 3.3), and more specifically, determine properties

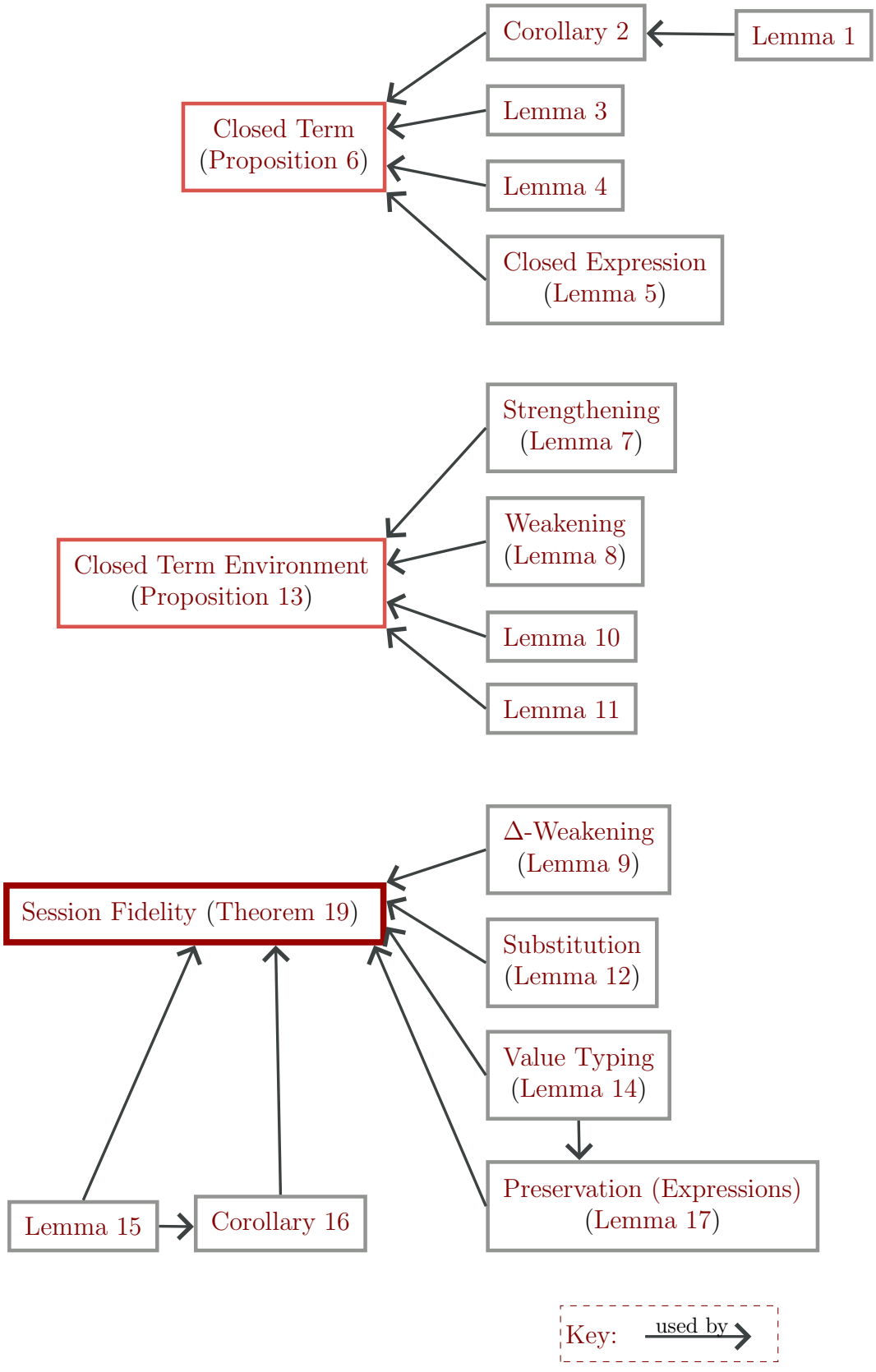


Figure 4.1: Lemmas and propositions leading to session fidelity

related to closed terms and their environments. This results in the Closed Term Environment Proposition (Proposition 13). These two aforementioned propositions act as a sanity check on the transition semantics, and the static type system, respectively. Finally, following these two propositions, we merge the static type system with the dynamic type system, where we form (and prove) the Session Fidelity Theorem. This is the most crucial part, since it verifies that: *(i)* a well-typed system remains well-typed even after transitioning; and that *(ii)* this same system is faithful to the protocol depicted by the session types.

The interactions of all the lemmata and propositions used, leading up to the Session Fidelity Theorem, are illustrated in Figure 4.1. We provide detailed proofs where possible, but exclude those that are less demanding to improve the readability of the chapter.

4.1 Validating the Transition Semantics

In Section 3.5, we defined a transition semantics in the form of an LTS, modelling an Elixir program at runtime. This transition semantics can be analysed in a number of ways: *(i)* the semantics can be compared with the actual implementation of the Elixir compiler itself, which goes beyond the scope of our study; *(ii)* it can be compared with the static semantics (*i.e.,* type system), which we will see through the Session Fidelity Theorem; and *(iii)* it can be analysed with respect with its *closeness* properties. In this section, we consider the last property (Item *iii*).

Open programs (*i.e.,* programs with free variables) are seen as incomplete programs, and cannot execute correctly due to missing information. Conversely, a program is complete, if it is *closed*, *i.e.,* it has no free variables (Definition 3.2). So, we require that when a closed program evaluates, it must remain closed. We show this in the Closed Term Proposition (Proposition 6).

Before proving Proposition 6, we must analyse some properties about closed terms, where we see how they affect variable substitutions (Definition A.7). In

Lemma 1, if some variable x does not exist in term t, then, if we replace x with some value, t must remain the same, *i.e.*, $t\,[^{v}/_{x}] = t$. Restricting this statement, we can say that, if x is not a free variable in t, then the same result should hold (Corollary 2). Lemma 3 consists of two statements that compare the free variables in terms (or expressions) with those that include a substitution.

Lemma 1. *$x \notin \boldsymbol{fv}(t) \cup \boldsymbol{bv}(t)$ implies $t\,[^{v}/_{x}] = t$*

Proof. By induction on the structure of t. □

Corollary 2. *$x \notin \boldsymbol{fv}(t)$ implies $t\,[^{v}/_{x}] = t$*

Proof. Straightforward from Lemma 1. □

Lemma 3.

i. *$x \in \boldsymbol{fv}(t)$ implies $\boldsymbol{fv}(t\,[^{v}/_{x}]) = \boldsymbol{fv}(t) \setminus \{x\}$*

ii. *$x \in \boldsymbol{fv}(e)$ implies $\boldsymbol{fv}(e\,[^{v}/_{x}]) = \boldsymbol{fv}(e) \setminus \{x\}$*

Proof. By induction on the structures of t and e for Items *i* and *ii* respectively. □

Lemma 4. *$\boldsymbol{match}(p, v) = [^{v_1,\,\cdots,\,v_n}/_{x_1,\,\ldots,\,x_n}]$, implies $\boldsymbol{vars}(p) = \{x_1,\ \ldots,\ x_n\}$*

Proof. By induction on the structure of p. □

Lemma 5 (Closed Expression)**.** *$\boldsymbol{fv}(e) = \emptyset$ and $e \to e'$ implies $\boldsymbol{fv}(e') = \emptyset$*

Proof. By induction on the structure of e. □

Finally, Lemmata 1–5 lead to the Closed Term Proposition (Proposition 6). By this proposition, we can say that a closed term t remains closed, even after t transitions to some new term t', producing an action α. Lemma 5 is analogous; it states that expressions remain closed after reductions.

Proposition 6 (Closed Term). *If* $\boldsymbol{fv}(t) = \emptyset$ *and* $t \xrightarrow{\alpha} t'$, *then* $\boldsymbol{fv}(t') = \emptyset$

Proof. By induction on the structure of t. We provide two main cases (see Appendix B for the full proof).

$[\boldsymbol{t = \mathtt{send}\,(w, \{\mathtt{:l}, e_1, \ldots, e_n\})}]$ Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [rCHOICE$_1$] From this rule, we know that $\alpha = \tau$ and

$$t' = \mathtt{send}\,(\iota, \{\mathtt{:l}, v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\})$$

$$e_k \to e'_k \tag{5a}$$

Since $\mathbf{fv}(t) = \emptyset$, then by the **fv** definition

$$\mathbf{fv}(\iota) = \emptyset \tag{5b}$$

$$\mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..k-1 \tag{5c}$$

$$\mathbf{fv}(e_i) = \emptyset \text{ for } i \in k-1..n \tag{5d}$$

Applying the Closed Expression Lemma to eqs. (5a) and (5d), results in $\mathbf{fv}(e_k) = \emptyset$. Using this information along with eqs. (5b–d) and the **fv** definition, results in $\mathbf{fv}(t') = \emptyset$ as required.

2. [rCHOICE$_2$] In this case $t = \{\mathtt{:l}, v_1, \ldots, v_n\}$ and $t' = \{\mathtt{:l}_\mu, v_1, \ldots, v_n\}$. Since from the premise $\mathbf{fv}(t) = \emptyset$, then using the **fv** definition,

$$\mathbf{fv}(\iota) = \emptyset, \quad \mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..n \tag{5e}$$

To show that $\mathbf{fv}(\{\mathtt{:l}_\mu, v_1, \ldots, v_n\}) = \emptyset$, we can apply eq. (5e) to the **fv** definition.

$[\boldsymbol{t = \mathtt{receive\ do}\ (\{\mathtt{:l}_i, \widetilde{p}_i\} \to t_i)_{i \in I}\ \mathtt{end}}]$ From the premise, we know that $\mathbf{fv}(t) = \emptyset$, so by the **fv** definition,

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p}_i) = \emptyset \quad \text{for all } i \in I \tag{6a}$$

Given that current structure of t, we can deduce $t \xrightarrow{\alpha} t'$ using [rBRANCH],

where $\alpha = ?\,\{\texttt{:l}_j,\ v_1,\ \ldots,\ v_n\}$ for some $j \in I$, and

$$\mathbf{match}(\widetilde{p_j}, \widetilde{v}) = \sigma \text{ where } \sigma = [v'_1,\ \ldots,\ v'_k/x_1,\ \ldots,\ x_k] \tag{6b}$$

$$t' = t_j\sigma$$

From eq. (6b), we can apply Lemma 4 to get

$$\mathbf{vars}(\widetilde{p_j}) = \{x_1,\ \ldots,\ x_k\} \tag{6c}$$

Substituting eq. (6c) in eq. (6a) (for $i = j$), we get $\mathbf{fv}(t_j) \setminus \{x_1,\ \ldots,\ x_k\} = \emptyset$. Our aim is to get $t_j\sigma = \emptyset$, so we check if $x \in \mathbf{fv}(t_j)$. If this is valid, then by Lemma 3, we can conclude that $\mathbf{fv}(t_j\,[v'_1/x_1]) \setminus \{x_2,\ \ldots,\ x_k\} = \emptyset$. In case when $x \notin \mathbf{fv}(t_j)$, the same can be concluded by Corollary 2. Applying the same procedure for a total of k times, results in $\mathbf{fv}(t_j\,[v'_1,\ \cdots,\ v'_k/x_1,\ \ldots,\ x_k]) = \emptyset$, as required.

The remaining cases are found in Appendix B. □

4.2 Properties of Typing

We provide some information regarding our type systems, including the strengthening and weakening properties. These will allow us to reason about the *variable binding* environment, Γ, in out typing judgement $\Delta \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$. Consequently, by Proposition 13, we shall conclude that when typing closed terms, the initial the typing judgement makes no assumptions on the *variable binding* environment, and can thus be *empty*, *i.e.*, $\Gamma = \emptyset$.

Certain changes in the mappings of the Δ and Γ environments, do not affect the derivation of a typing judgement. Starting from the Strengthening Lemma (Lemma 7), we show that under some restrictions, we can safely remove mappings from the *variable binding* environment Γ, without affecting the overall typing result. This lemma consists of two statements: *(i)* we can strengthen Γ, by removing the mapping $x{:}T$, with the condition that x cannot be a free variable in the concerned

term t; *(ii)* the second statement is similar to *(i)*, with the difference that it is applied to expressions instead of terms.

Lemma 7 (Strengthening)**.**

i. *If* $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \triangleright t : T \triangleleft S'$ *and* $x \in \boldsymbol{fv}(t)$*, then* $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$

ii. *If* $\Gamma, x : T' \vdash_{exp} e : T$ *and* $x \in \boldsymbol{fv}(e)$*, then* $\Gamma \vdash_{exp} e : T$

Proof. We prove Item *i* by induction on the derivation of $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \triangleright t : T \triangleleft S'$. Similarly, for Item *ii*, we prove it by induction on $\Gamma, x : T' \vdash_{\text{exp}} e : T$. □

Dual to the Strengthening Lemma is the Weakening Lemma (Lemma 8), which is able to expand the *variable binding* environment Γ without any side-effects.

Lemma 8 (Weakening)**.**

i. *If* $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$*, then* $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \triangleright t : T \triangleleft S'$

ii. *If* $\Gamma \vdash_{exp} e : T$*, then* $\Gamma, x : T' \vdash_{exp} e : T$

Proof. We prove Item *i* by induction on the derivation of $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$, and Item *ii* is proved by induction on the derivation of $\Gamma \vdash_{\text{exp}} e : T$. □

Similar to the Weakening Lemma which weakens (*i.e.,* extends) the *variable binding* environment Γ, the Δ-Weakening Lemma extends the *session typing* environment Δ without affecting the typing results.

Lemma 9 (Δ-Weakening)**.** *If* $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$*, then* $(\Delta, \Delta') \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$

Proof. In Appendix B. □

In Section 3.3.2, we presented the pattern typing judgement, $\vdash^{w}_{\text{pat}} p : T \triangleright \Gamma$. By Lemma 10, we can match the variables from the pattern p to the variables inside the mapping produced by this judgement (Γ). For example, from $\vdash^{w}_{\text{pat}} \{x, 4, y\} : T \triangleright \Gamma$, the domain of the mapping Γ will contain $\{x, y\}$.

Lemma 10. *Given some pattern p, such that $\vdash^{w}_{pat} p : T \triangleright \Gamma$, then $\boldsymbol{vars}(p) = dom(\Gamma)$*

Proof. Follows by induction on $\vdash^{w}_{\text{pat}} p : T \triangleright \Gamma$. In Appendix B. □

Lemma 11. *If $\boldsymbol{fv}(e) = \emptyset$ and $\Gamma \vdash_{exp} e : T$, then $\emptyset \vdash_{exp} e : T$*

Proof. Follows by induction on the derivation of $\Gamma \vdash_{\text{exp}} e : T$.

[**tVariable**] From the rule $e = x$ and by the **fv** definition, $\mathbf{fv}(x) \neq \emptyset$. Therefore, case holds trivially.

[**tLiteral**], [**tEList**] The *variable binding* environment (Γ) is unused, so cases hold immediately.

[**tTuple**] From the rule, we know that $e = \{e_1, \ldots, e_n\}$ and

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 1..n \tag{7a}$$

Since $\mathbf{fv}(e) = \emptyset$, by the current structure of e and the **fv** definition, we get

$$\mathbf{fv}(e_1) = \emptyset \quad \text{for all } i \in 1..n \tag{7b}$$

Using eqs. (7a) and (7b) and the inductive hypothesis, we get $\emptyset \vdash_{\text{exp}} e_i : T_i$ for $i \in 1..n$. Applying the latter to [tTuple], results in $\emptyset \vdash_{\text{exp}} e : T$, as required, so case holds.

The remaining cases are analogous to the previous case. □

Our type system can possess the property of session fidelity, if well-typed terms remains well-typed after transitioning. As terms transition, in particular in the rules [rLet$_2$], [rCall$_2$] and [rBranch], variables are substituted with values. The Substitution Lemma (Lemma 12) ensures that when free variables inside of terms and expressions are substituted with a closed value, the resulting terms and expressions remains well-typed. As a result, the substituted variables become redundant, and thus can be removed from the *variable binding* environment, Γ. This lemma consists of two statements, where substitution is performed in *(i)* terms, and *(ii)* expressions.

Lemma 12 (Substitution)**.**

i. If $\Gamma \vdash_{exp} v : T'$ *and* $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \triangleright t : T \triangleleft S'$*, then* $\Delta \cdot \Gamma \vdash^{w[v/x]} S \triangleright t\,[v/x] : T \triangleleft S'$

ii. If $\Gamma \vdash_{exp} v : T'$ *and* $\Gamma, x : T' \vdash_{exp} e : T$*, then* $\Gamma \vdash_{exp} e\,[v/x] : T$

Proof. By induction on the derivation of $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \triangleright t : T \triangleleft S'$ for Item *i*, and by induction on the derivation of $\Gamma, x : T' \vdash_{\text{exp}} e : T$ for Item *ii*. □

In the [tMODULE] (Section 3.3.4), we explained how typechecking is initiated, *i.e.,* it is performed by typechecking each public function individually using the typing judgement $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$. The *session typing* environment (Δ), and the *variable binding* environment (Γ) are the prior assumptions that the typing judgement has to make. We assert that when each public function is typechecked, this typing judgement does not need any prior assumptions bar one. The only assumption is in Δ, which should contain a mapping containing the current function's session type. Since the function definitions are always closed (*i.e.,* their body has no free variables), then the *variable binding* environment Γ should not contain any prior assumptions, *i.e.,* Γ can be set to the empty mapping ($\emptyset$). We show this in Closed Term Environment Proposition (Proposition 13), which acts as a sanity check over our type system defined in Section 3.3.

Proposition 13 (Closed Term Environment)**.** *If* $\boldsymbol{fv}(t) = \emptyset$ *and* $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T \triangleleft S'$*, then* $\Delta \cdot \emptyset \vdash^{w} S \triangleright t : T \triangleleft S'$

Proof. By induction on the typing derivation $\Delta \cdot \Gamma \vdash^{w} S \triangleright t : T' \triangleleft S'$. We consider the main cases:

[tBRANCH] From the rule, we know that $t = \texttt{receive do } (\{\texttt{:l}_i, p_i^1, \ldots, p_i^n\} \to t_i)_{i \in I} \texttt{end}$, and

$$\forall i \in I \cdot \begin{cases} \vdash^{w}_{\text{pat}} p_i^j : T_i^j \triangleright \Gamma_i^j \text{ for all } j \in 1..n & \text{(8a)} \\ \Delta \cdot \left(\Gamma, \Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^{w} S_i \triangleright t_i : T \triangleleft S' & \text{(8b)} \end{cases}$$

Since $\mathbf{fv}(t) = \emptyset$, then by the *Free Variables* Definition, we know that

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i^1, \ldots, p_i^n) = \emptyset \quad \text{for all } i \in I \tag{8c}$$

From eq. (8c) we have to consider two sub-cases:

1. If for all $x \in \mathbf{vars}(p_i^1, \ldots, p_i^n)$, such that $x \notin \mathbf{fv}(t_i)$, then we can immediately conclude by eq. (8c) that

   $$\mathbf{fv}(t_i) = \emptyset \quad \text{for all } i \in I \tag{8d}$$

   By eq. (8d), (8b) and the inductive hypothesis, we get $\Delta \cdot \emptyset \vdash^w S_i \rhd t_i : T \lhd S'$. In order to add $\Gamma_i^1, \ldots, \Gamma_i^n$ to the *variable binding* environment, we can apply the Weakening Lemma, resulting in $\Delta \cdot \left(\Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^w S_i \rhd t_i : T \lhd S'$.

2. In the other case, where there is some $x \in \mathbf{vars}(p_i^1, \ldots, p_i^n)$, such that $x \in \mathbf{fv}(t_i)$, then by repeatedly applying Lemma 10 to eq. (8a), we get $\mathbf{vars}(p_i^1, \ldots, p_i^n) = dom(\Gamma_i^1, \ldots, \Gamma_i^n)$ for all $i \in I$. This means that if we collect all of the free variables from t_i, they will be equivalent to the domain of $\Gamma_i^1, \ldots, \Gamma_i^n$. Thus, we can strengthen eq. (8b) to reduce $\left(\Gamma, \Gamma_i^1, \ldots, \Gamma_i^n\right)$ into $\left(\Gamma_i^1, \ldots, \Gamma_i^n\right)$ using the Strengthening Lemma, resulting in $\Delta \cdot \left(\Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^w S_i \rhd t_i : T \lhd S'$.

Using the (same) result from both sub-cases, along with eq. (8a), we can apply [tBranch] to get $\Delta \cdot \emptyset \vdash^w S \rhd t : T \lhd S'$, as required.

[**tRecUnknownCall**] From the rule, we know that $t = f\,(w, e_2, , \ldots, e_n)$ and

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{9a}$$

$$(\Delta, f_n : S) \cdot \left(\Gamma, y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\right) \vdash^y S \rhd \bar{t} : T \lhd S' \tag{9b}$$

$$\text{where } \widetilde{x}, \widetilde{T}, \bar{t}, T \text{ and } y \text{ are obtained from } \Sigma$$

Since $\mathbf{fv}(t) = \emptyset$, by the definition of $\mathbf{fv}$, we know that $\mathbf{fv}(e_i) = \emptyset$ for all $i \in 2..n$. Using this information, along with eq. (9a) and Lemma 11, we get

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{9c}$$

Since the *function information* environment, Σ, must be well-formed, we know that for all $f_n \in dom(\Sigma)$,

$$\mathbf{fv}(\Sigma(f_n).\texttt{body}) \setminus \Sigma(f_n).\texttt{params} \setminus \Sigma(f_n).\texttt{dual} = \emptyset$$

From the premise of [tRecUnknownCall], $\Sigma(f_n)$ is defined *explicitly*, thus we can say that the *only* free variables in $\bar{t}$ are y and $\widetilde{x}$. This allows us to strengthen eq. (9b), to remove Γ, using the Strengthening Lemma, to get $(\Delta, f_n : S)\cdot\big(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\big) \vdash^{y} S \triangleright \bar{t} : T \triangleleft S'$. Using this information and eq. (9c) as the premise for [tRecUnknownCall], we get $\Delta \cdot \emptyset \vdash^{w} S \triangleright t : T \triangleleft S'$, as required.

Regarding the remaining cases: Case [tExpression] holds immediately using Lemma 11; Cases [tCase] and [tLet] are similar to [tBranch]. Finally, Cases [tChoice] and [tRecKnownCall] hold effortlessly if we apply the *Free Variables* Definition and Lemma 11. □

4.3 Session Fidelity

Before showing that typing is preserved under reduction, and that terms are faithful to a session type, we must define further auxiliary lemmata. Starting from Lemma 14, it links a type to the basic values (and vice versa), *e.g.* the value 5 has type number.

Lemma 14 (Value Typing)**.**

i. $\Gamma \vdash_{exp} v : \mathsf{boolean}$ *iff* $v = boolean$

ii. $\Gamma \vdash_{exp} v : \mathsf{number}$ *iff* $v = number$

iii. $\Gamma \vdash_{exp} v : \mathsf{atom}$ *iff* $v = atom$

iv. $\Gamma \vdash_{exp} v : \mathsf{pid}$ *iff* $v = \iota$

v. $\Gamma \vdash_{exp} v : [T]$ *iff* $v = [\, v_1 \mid v_2 \,]$ *or* $v = [\,]$

vi. $\Gamma \vdash_{exp} v : \{\widetilde{T}\}$ *iff* $v = \{\widetilde{v}\}$

Proof. By case analysis on the expression typing rules. □

Lemma 15 provides a guarantee that the variables inside the substitutions produced by the **match** function have the expected types. It also ensures that the variables from the same substitutions, which are stored in Γ, are assigned with

the same types. Consequently, Corollary 16 provides the same guarantees but for a *sequence* of patterns and values.

Lemma 15. *For all patterns p and values v,*

$$\left.\begin{array}{r} \boldsymbol{match}(p, v) = [v_1, \ldots, v_n/x_1, \ldots, x_n] \\ \vdash^{w}_{pat} p : T \rhd \Gamma \\ \emptyset \vdash_{exp} v : T \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \Gamma = x_1 : T_1, \ldots, x_n : T_n \\ \emptyset \vdash_{exp} v_i : T_i \quad \textit{for } i \in 1..n \end{array}\right.$$

Proof. In Appendix B. □

Corollary 16. *For all patterns $\widetilde{p} = p^1, \ldots, p^n$, values $\widetilde{v} = v_1, \ldots, v_n$ and $\forall j \in 1..n$, then the following implication holds.*

$$\left.\begin{array}{r} \boldsymbol{match}(\widetilde{p}, \widetilde{v}) = [v'_1, \ldots, v'_k/x_1, \ldots, x_k] \\ \vdash^{y}_{pat} p^j : T^j \rhd \Gamma^j \\ \emptyset \vdash_{exp} v_j : T^j \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \widetilde{\Gamma} = \Gamma^1, \ldots, \Gamma^j = x_1 : T_1, \ldots, x_k : T_k \\ \emptyset \vdash_{exp} v'_i : T_i \quad \textit{for } i \in 1..k \end{array}\right.$$

Proof. In Appendix B. □

Terms and expression can be executed to get tangible results – this happens during transitions (for terms) and reductions (for expressions). Starting with expressions, in the Preservation (Expressions) Lemma (Lemma 17), we show that the type of expressions remains unchanged (or preserved) after an expression is reduced. This means that expressions should have a constant type in all steps of reductions, until the expression cannot be reduced further. The term *Preservation* is sometimes referred to as *Subject Reduction* [31].

Lemma 17 (Preservation (Expressions))**.** *If $\emptyset \vdash_{exp} e : T$ and $e \to e'$, then $\emptyset \vdash_{exp} e' : T$*

Proof. Follows by induction on $\emptyset \vdash_{\text{exp}} e : T$. We consider the main cases:

[tTuple] From the rule, we know that $e = \{e_1, \ldots, e_k, \ldots, e_n\}$, $T = \{T_1, \ldots, T_n\}$ and

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 1..n \tag{10a}$$

Deriving $e \to e'$ using [reTuple] results in $e' = \{v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\}$ and

$$e_k \to e'_k \tag{10b}$$

Applying eqs. (10a) and (10b) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_k : T_k$. By the latter, eq. (10a) and [tTuple], we get $\emptyset \vdash_{\text{exp}} e' : T$, as required.

[**tArithmetic**] From the rule we know that $e = e_1 \diamond e_2$, $T = \mathsf{number}$ and

$$\emptyset \vdash_{\text{exp}} e_1 : \mathsf{number} \tag{11a}$$

$$\emptyset \vdash_{\text{exp}} e_2 : \mathsf{number} \tag{11b}$$

$e \to e'$ can be derived using different rules, so we consider three sub-cases:

1. [**reOperation$_1$**] From this rule we know that $e' = e'_1 \diamond e_2$ and

   $$e_1 \to e'_1 \tag{11c}$$

   Applying eqs. (11a) and (11c) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_1 : \mathsf{number}$. Using this information, along with eq. (11b) in [tArithmetic], results in $\emptyset \vdash_{\text{exp}} e' : \mathsf{number}$, as required.

2. [**reOperation$_2$**] Analogous to [reOperation$_1$].

3. [**reOperation$_3$**] From the rule, we know that $e = v_1 \diamond v_2$ and e' has some value $v = v_1 \diamond v_2$. Since we know that $\emptyset \vdash_{\text{exp}} e : T$, or $\emptyset \vdash_{\text{exp}} v_1 \diamond v_2 : T$, then $\emptyset \vdash_{\text{exp}} e' : T$ follows immediately given that $e' = v = v_1 \diamond v_2$.

Regarding the remaining cases: Cases [tLiteral], [tVariable] and [tEList] hold trivially, since $e \to e'$ does not apply. Cases [tComparison] and [tBoolean] are analogous to [tArithmetic]. Cases [tList] and [tNot] take a similar approach to [tTuple]. □

Evolution of expressions are unambiguous since expressions remain associated to one type, *e.g.*, in $2 + 3 \rightarrow 5$, the expression $2 + 3$, which has type number, is reduced to the value 5, which also has the same type. However, we also have to consider how terms are evaluated. When a term transitions, it produces an action (α), which can contain side-effects such as sending or receiving messages. These side-effects are reflected by a session type that progresses depending on what messages were sent or received, or remains constant if no messages were transferred. This behaviour is shown in the following *After Function* Definition.

Definition 4.1 (*After Function*)**.** The first application of the **after** function uses a session type (S), where it evaluates the new session type expected *after* performing a certain *action* (α). The function, denoted as $\mathbf{after}(S, \alpha)$, provides a continuation session type only if the action α is allowed by the protocol S, shown in Lemma 18. The **after** function is defined as follows:

$$\mathbf{after}(S, \tau) = S$$

$$\mathbf{after}(S, f_n) = S$$

$$\mathbf{after}(\oplus\{!\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}, \iota!\{\mathtt{l}_j, \widetilde{v}\}) = S_j \quad \text{where } j \in I$$

$$\mathbf{after}(\&\{?\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}, ?\{\mathtt{l}_j, \widetilde{v}\}) = S_j \quad \text{where } j \in I$$

The **after** function is overloaded, where it extends the *session typing* environment (Δ) as follows. The function $\mathbf{after}(\Delta, \alpha, S)$, computes a new *session typing* environment given some action α and session type S:

$$\mathbf{after}(\Delta, f_n, S) = \Delta, f_n : S$$

$$\mathbf{after}(\Delta, \alpha, S) = \Delta \qquad \text{if } \alpha \neq f_n$$ ■

Lemma 18. *If* $\mathbf{agree}(S, \alpha)$ *is defined, then there exists* S'*, such that* $\mathbf{after}(S, \alpha) = S'$

Proof. By induction on the definition $\mathbf{agree}(S, \alpha)$. □

The *After Function* Definition serves two purposes: *(i)* it implicitly says whether an action α is following a protocol S, so $\mathbf{after}(S, \alpha)$ will fail if the action α does not follow S; we show this in Lemma 18, and *(ii)* the **after** function provides the continuation session type, *i.e.,* $\mathbf{after}(S, \alpha) = S'$.

We now consider the most crucial theorem of the type system of Chapter 3, the Session Fidelity Theorem. This is similar to the preservation property, however it also takes the interaction protocols (*i.e.,* session types) into consideration. Intuitively, it states that when a term t is well-typed it must remain well-typed even after it is transitions. Concretely, a closed term t is well-typed, if this judgement holds:

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright t : T \triangleleft S' \tag{12}$$

where S and S' are initial and residual session types, respectively, and T is the basic expression type. Then, as term t transitions to a new term t', it produces an action α:

$$t \xrightarrow{\alpha} t' \tag{13}$$

Session fidelity holds, if the new term t' remains well-typed, shown by this judgement:

$$\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright t' : T \triangleleft S' \tag{14}$$

where the evolved S'' and Δ' are computed using the *After Function* Definition, *i.e.,* $\mathbf{after}(S, \alpha) = S''$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta'$.

The session fidelity property gives multiple guarantees. First of all, the expression type must remain preserved, which is shown by the constant type T in eqs. (12) and (14). Furthermore, the term t must follow an interaction protocol starting from the initial session type S up to the residual session type S' (eq. (12)). Then, when t transitions to t' (eq. (13)), term t' must follow the evolved initial session type S'' and residual session type S' (eq. (14)).

When typechecking each function, as initiated by [TMODULE], the residual session type S' is set to the end type. This is important, since the session fidelity

guarantees that as term transitions, the actions produced will follow the protocol defined by S up to S'. Furthermore, if a term t is terminates (*i.e.,* no more transitions are allowed), then we know that the initial protocol S must have reduced to end, thus fully exhausting the defined protocol.

Example 4.1. Consider a closed term t_1, which is well-typed using this judgement: $\Delta \cdot \emptyset \vdash^{w} S_1 \rhd t_1 : T \lhd \mathsf{end}$. Term t_1 must follow the initial session type S_1 and residual session type end. The session fidelity property states that if t_1 transitions to t_2, producing action α, then t_2 must follow the new evolved session type S_2 (produced using the **after** function, *i.e.,* $S_2 = \mathbf{after}(S_1, \alpha_1)$). Finally, if the term terminates (*i.e.,* becomes a value v), then the protocol S_1 must have been fully exhausted, reaching the final session type end. This example transition is shown in the following trace:

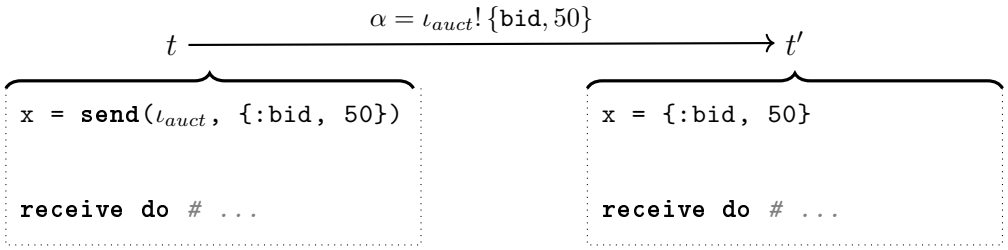
$$\underset{S_1}{t_1} \xrightarrow{\alpha_1} \underset{\mathbf{after}(S_1,\alpha_1)=S_2}{t_2} \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_3} \underset{\mathbf{after}(S_3,\alpha_3)=S_4}{t_4} \xrightarrow{\alpha_4} \underset{\mathbf{after}(S_4,\alpha_4)=\mathsf{end}}{v} \not\longrightarrow$$

■

Example 4.2. We consider a concrete example to show how typing behaves under transitions. From the auction example (Listing 1.3), the buyer process needs to obey the *auction* protocol:

$$\mathit{auction} = !\mathtt{bid}(\mathsf{number}).\& \left\{ \begin{array}{l} ?\mathtt{sold}().\mathsf{end}, \\ ?\mathtt{higher}(\mathsf{number}).\ldots \end{array} \right\}$$

In this example, the buyer process starts by sending a message containing a `bid` label to the auctioneers's *pid* (ι_{auct}), as shown in the snippet of the function body t.



As the process evaluates, term t transitions to t', where it sends a message as a side-effect. This side-effect is denoted as action α, where $\alpha = \iota_{server}!\{\texttt{bid}, 50\}$. By the *After Function* Definition, *auction* evolves to a new session type S'':

$$S'' = \textbf{after}(\mathit{auction}, \alpha) = \ \& \left\{ \begin{array}{l} ?\texttt{sold}().\mathsf{end}, \\ ?\texttt{higher}(\mathsf{number}).\ldots \end{array} \right\}$$

For t' to remain well-typed, it must now match with the evolved session type S'', where it has to be able to receive a message labelled `stop` or `higher`. The term t' may transition further until it reaches the end session type, where it fully consumes the *auction* session type. ■

In the Session Fidelity Theorem (Theorem 19), when a well-typed term t transitions, the term t must be faithful to some initial and final session types, S and S', respectively. The term t must be a closed term, in which when it transitions to t' (*i.e.*, $t \xrightarrow{\alpha} t'$), an action α is produced which affects the initial session type S and the *session typing* environment Δ, as defined by the *After Function* Definition. Furthermore, the term typing judgement makes no prior assumptions (*e.g.*, Γ is empty), as explained in Proposition 13. Theorem 19 utilises the information from the *function information* environment, Σ, which provides information about the functions in a module, and was used similarly in the [tModule] typing rule.

Theorem 19 (Session Fidelity)**.** *If* $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$ *and* $t \xrightarrow[\Sigma]{\alpha} t'$*, then there exists some* S'' *and* Δ'*, such that* $\Delta' \cdot \emptyset \vdash^{w}_{\Sigma} S'' \rhd t' : T \lhd S'$ *for* $\textbf{\textit{after}}(S, \alpha) = S''$ *and* $\textbf{\textit{after}}(\Delta, \alpha, S) = \Delta'$

Proof. By induction on the typing derivation $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$.

[**TLET**] From the rule, we know that $x \neq w$, and

$$t = (x = t_1;\ t_2) \tag{15a}$$

$$\Delta \cdot \emptyset \vdash^{w} S \rhd t_1 : T' \lhd S''' \tag{15b}$$

$$\Delta \cdot (x : T') \vdash^{w} S''' \rhd t_2 : T \lhd S' \tag{15c}$$

From the structure of t (eq. (15a)), term reduction ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**RLET$_1$**] From this rule, we know that $t' = (x = t_1';\ t_2)$ and

$$t_1 \xrightarrow{\alpha} t_1' \tag{15d}$$

By eqs. (15b) and (15d) and the inductive hypothesis we obtain

$$\Delta' \cdot \emptyset \vdash^{w} S'' \rhd t_1' : T' \lhd S''' \tag{15e}$$

where $\textbf{after}(S, \alpha) = S''$ and $\textbf{after}(\Delta, \alpha, S) = \Delta'$. Also, by the *After Function* Definition, we know that Δ' is an extension of Δ, so we can apply the Δ-Weakening Lemma on eq. (15c) to get

$$\Delta' \cdot (x : T') \vdash^{w} S''' \rhd t_2 : T \lhd S' \tag{15f}$$

Using eqs. (15e) and (15f) as the premise for rule [TLET], we obtain:

$$[\text{TLET}]\ \frac{\Delta' \cdot \emptyset \vdash^{w} S'' \rhd t_1' : T' \lhd S''' \qquad \Delta' \cdot (x : T') \vdash^{w} S''' \rhd t_2 : T \lhd S' \qquad x \neq w}{\Delta' \cdot \emptyset \vdash^{w} S'' \rhd x = t_1';\ t_2 : T \lhd S'}$$

where $\Delta' \cdot \emptyset \vdash^{w} S'' \rhd t' : T \lhd S'$ is the expected result.

2. [**RLET$_2$**] From the rule, we know that $t = (x = v;\ t_2)$, $t' = t_2\,[v/x]$ and $\alpha = \tau$. Since $t_1 = v$, by eq. (15b) and [TEXPRESSION], then $\emptyset \vdash_{\text{exp}} v : T'$ holds. If we apply this latter information and eq. (15c) to the Substitution Lemma, we obtain $\Delta \cdot \emptyset \vdash^{w[v/x]} S''' \rhd t_2\,[v/x] : T \lhd S'$. This is the expected

result, since by the *Variable Substitution* Definition, $w\,[v/x] = w$; and by the **after** definition, $S''' = S$ and $\Delta' = \Delta$.

[tBranch] From the rule, we know that for some $n \in \mathbb{N}$ and

$$S = \&\left\{?\mathtt{l}_i\left(T_i^1,\ \ldots,\ T_i^n\right).S_i\right\}_{i \in I} \tag{16a}$$

$$t = \mathtt{receive\ do}\ \left(\left\{:\mathtt{l}_i, p_i^1,\ \ldots,\ p_i^n\right\} \to t_i\right)_{i \in I}\mathtt{end} \tag{16b}$$

From the premise, we also know that some properties regarding each individual branch from the `receive` construct:

$$\forall i \in I \begin{cases} \vdash^w_{\text{pat}} p_i^j : T_i^j \ \triangleright\ \Gamma_i^j \ \text{ for all } j \in 1..n & \text{(16c)} \\ \Delta \cdot \left(\Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash^w S_i \triangleright t_i : T \triangleleft S' & \text{(16d)} \end{cases}$$

From the structure of t (eq. (16b)), term reduction ($t \xrightarrow{\alpha} t'$) can only be derived using [rBranch], where execution progresses to a single branch (*i.e.*, t_μ), rather than all branches. The right branch is chosen by matching its label, $\mathtt{l}_{i \in I}$, to the label received in the incoming message, $\mathtt{l}_\mu$. Thus, for some $k \in \mathbb{N}$, there exists some $\mu \in I$ where $\mathtt{l}_\mu = \mathtt{l}_i$, and

$$\alpha = ?\left\{:\mathtt{l}_\mu,\ v_1,\ \ldots,\ v_n\right\} \tag{16e}$$

$$\mathbf{match}((p_\mu^1,\ \ldots,\ p_\mu^n), (v_1,\ \ldots,\ v_n)) = [v_1', \ldots, v_k'/x_1, \ldots, x_k] \tag{16f}$$

$$t' = t_\mu\,[v_1', \ldots, v_k'/x_1, \ldots, x_k]$$

From eq. (16e), α refers to the message received from the dual process. We can compare the contents of this message to the original session type S (eq. (16a)), to obtain information regarding the types of the individual values inside α. We know that α contains a label $\mathtt{l}_\mu$ and n values. Thus for $j \in 1..n$, each value v_j, has a corresponding type $\boldsymbol{T_\mu^j}$ from the session type S, where S contains $?\mathtt{l}_\mu\left(\boldsymbol{T_\mu^1},\ \ldots,\ \boldsymbol{T_\mu^n}\right).S_\mu$. Formally, this can be written as

$$\emptyset \vdash_{\text{exp}} v_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{16g}$$

Applying eqs. (16c), (16f) and (16g) into Corollary 16, results in $\widetilde{\Gamma_\mu} = \Gamma_\mu^1,\ \ldots,\ \Gamma_\mu^n = x_1 : T_1,\ \ldots,\ x_k : T_k$ and

$$\emptyset \vdash_{\text{exp}} v_m' : T_m \ \text{ for } m \in 1..k \tag{16h}$$

Applying eq. (16h) and $\Delta \cdot \widetilde{\Gamma_\mu} \vdash^w S_\mu \rhd t_\mu : T \lhd S'$ (from eq. (16d) for $i = \mu$) repeatedly to the Substitution Lemma, we get

$$\Delta \cdot \emptyset \vdash^w S_\mu \rhd t_\mu \left[v'_1, \ldots, v'_k / x_1, \ldots, x_k\right] : T \lhd S' \tag{16i}$$

Since $\mathbf{after}(\&\{?\mathtt{l}_i(\widetilde{T_i}).S_i\}_{i\in I}, \alpha) = S_\mu$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$, then eq. (16i) is the expected result.

[tCHOICE] From the rule, we know that for some $\mu \in I$, $T = \{\mathsf{atom}, T^1_\mu, \ldots, T^n_\mu\}$ and

$$S = \oplus\{!\mathtt{l}_i(\widetilde{T_i}).S_i\}_{i\in I} \tag{17a}$$

$$t = \mathtt{send}\,(\iota, \{:\mathtt{l}_\mu, e_1, \ldots, e_n\}) \tag{17b}$$

$$\emptyset \vdash_{\text{exp}} e_j : T^j_\mu \quad \text{for all } j \in 1..n \tag{17c}$$

From the structure of t (eq. (17b)), term reduction ($t \xrightarrow{\alpha} t'$) can be derived by several rules, so we have to consider two sub-cases:

1. Derived by the rule [rCHOICE$_1$], we know that $\alpha = \tau$ and

$$t' = \mathtt{send}\,(\iota, \{:\mathtt{l}, v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\})$$

$$e_k \to e'_k \tag{17d}$$

   Applying eq. (17c) (for $j = k$) and eq. (17d) to the Preservation (Expressions) Lemma, we get $\emptyset \vdash_{\text{exp}} e'_k : T_k$. Applying this and eq. (17c) to [tCHOICE] results in $\Delta \cdot \emptyset \vdash^w S \rhd t' : T \lhd S_\mu$. Since $\mathbf{after}(S, \tau) = S$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$, this holds.

2. [rCHOICE$_2$] From this rule we know that

$$t' = \{:\mathtt{l}_\mu, v_1, \ldots, v_n\}$$

$$\alpha = \iota!\{:\mathtt{l}_\mu, v_1, \ldots, v_n\} \tag{17e}$$

   where α (eq. (17e)) is the message being sent to the dual process with *pid* ι.

   Recall eq. (17c), where we have $\emptyset \vdash_{\text{exp}} e_j : \mathbf{T}^{\mathbf{j}}_\mu$ for $j \in 1..n$. Notice, that the types $\boldsymbol{T}^{\boldsymbol{j}}_{\boldsymbol{\mu}}$ were obtained from the session type S (eq. (17a)), where S

contains $!\mathtt{l}_\mu(\ \boldsymbol{T^1_\mu},\ \ldots,\ \boldsymbol{T^n_\mu}\ ).S_\mu$. Now, by the premise of [rChoice$_2$], since $e_j = v_j$, then

$$\emptyset \vdash_{\text{exp}} v_j : T^j_\mu \quad \text{for all } j \in 1..n \tag{17f}$$

By the Value Typing Lemma, we also know that $\emptyset \vdash_{\text{exp}} :\mathtt{l}_\mu : \mathsf{atom}$. Using this latter information and eq. (17f) in [tTuple] and [tExpression], we get the required result:

$$\begin{array}{r} [\text{tTuple}] \\ [\text{tExpression}] \end{array} \frac{\emptyset \vdash_{\text{exp}} :\mathtt{l}_\mu : \mathsf{atom} \qquad \forall j \in 1..n \qquad \emptyset \vdash_{\text{exp}} v_j : T^j_\mu}{\emptyset \vdash_{\text{exp}} \{:\mathtt{l}_\mu, v_1,\ \ldots,\ v_n\} : \{\mathsf{atom}, T^1_\mu,\ \ldots,\ T^n_\mu\}} \\ \overline{\Delta \cdot \emptyset \vdash^y S_\mu \rhd \{:\mathtt{l}_\mu, v_1,\ \ldots,\ v_n\} : T \lhd S_\mu} \tag{17g}$$

Result from eq. (17g) holds as required, since $\mathbf{after}(S, \alpha) = S_\mu$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$.

[**tRecKnownCall**] From the rule, we know that

$$t = f\,(w, e_2,\ \ldots,\ e_n) \tag{18a}$$

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for } \forall i \in 2..n \tag{18b}$$

From the structure of t (eq. (18a)), term reduction ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**rCall$_1$**] From this rule, we know that $t = f\,(v_1,\ \ldots,\ v_{k-1},\ e_k,\ \ldots,\ e_n)$, $\alpha = \tau$, $w = v_1$ and

$$t' = f\,(v_1,\ \ldots,\ v_{k-1},\ e'_k,\ \ldots,\ e_n)$$

$$e_k \to e'_k \tag{18c}$$

Applying eq. (18b) (for $i = k$) and eq. (18c) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e'_k : T_k \tag{18d}$$

By eqs. (18b) and (18d) and [tRecKnownCall], we get

$$\Delta \cdot \emptyset \vdash^w S \rhd f\,(v_1,\ \ldots,\ v_{k-1},\ e'_k,\ \ldots,\ e_n) : T \lhd S' \tag{18e}$$

eq. (18e) holds since $\mathbf{after}(S, \tau) = S$ and $v_1 = w$.

2. [$\textbf{RCALL}_2$] From the rule, we know that $\alpha = f_n$, $w = \iota$ and

$$t = f\,(\iota, v_2,\ \ldots,\ v_n) \tag{18f}$$

$$t' = \bar{t}\,[\iota/y]\,[v_2,\ \ldots,\ v_n/x_2,\ \ldots,\ x_n]$$

$$\Sigma\,(f_n) = \Omega \text{ where } \begin{cases} \Omega.\texttt{return\_type} = T \\ \Omega.\texttt{param\_types} = T_2,\ \ldots,\ T_n \end{cases} \tag{18g}$$

$$\Delta\,(f_n) = S \tag{18h}$$

Since all *known* functions (*i.e.,* $f_n \in dom(\Delta)$) by eq. (18h)) are already typechecked once before, then from the *function information* environment (*i.e.,* Σ) and eq. (18g), we can assume that

$$\Delta \cdot \Gamma' \vdash^{y} S \triangleright \bar{t} : T \triangleleft \mathsf{end} \tag{18i}$$

where Γ' contains *only* the mapping from the parameter names to their types, *i.e.,* $\Gamma' = (y : \mathsf{pid}, x_2 : T_2,\ \ldots,\ x_n : T_n)$ – our aim is to change Γ' to $\emptyset$. This assumption in eq. (18i) is possible since a well-formed Σ dictates that the only free variables in a function body are the parameter types, or formally, for all $f_n \in dom(\Sigma)$, we have

$$\mathbf{fv}(\Sigma(f_n).\texttt{body}) \setminus \Sigma(f_n).\texttt{params} \setminus \Sigma(f_n).\texttt{dual} = \emptyset$$

By eq. (18f) and Value Typing Lemma we know that $\emptyset \vdash_{\text{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (18i) to the Substitution Lemma results in

$$\Delta \cdot (x_2 : T_2,\ \ldots,\ x_n : T_n) \vdash^{y[\iota/y]} S \triangleright \bar{t}\,[\iota/y] : T \triangleleft \mathsf{end} \tag{18j}$$

where by the *Variable Substitution* Definition, $y\,[\iota/y] = \iota = w$.

Applying the Substitution Lemma multiple times to eqs. (18b) and (18j), results in

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright \bar{t}\,[\iota/y]\,[v_2,\ \ldots,\ v_n/x_2,\ \ldots,\ x_n] : T \triangleleft \mathsf{end} \tag{18k}$$

as required, since $\mathbf{after}(S, f_n) = S$ and $S' = \mathsf{end}$. Also, $\mathbf{after}(\Delta, f_n, S) = (\Delta, f_n : S)$, but from eq. (18h), f_n is already mapped to S in the *session typing* environment, therefore $(\Delta, f_n : S) = \Delta$, as needed.

[TRECUNKNOWNCALL] From the rule, we know

$$t = f\,(w, e_2, , \ \ldots, \ e_n) \tag{19a}$$

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{19b}$$

From the premise we also know that

$$(\Delta, f_n : S) \cdot \left(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\right) \vdash^{y} S \rhd \bar{t} : T \lhd S' \quad \text{where } \widetilde{x}, \widetilde{T}, \bar{t}, T \text{ and } y \text{ are obtained from the } \textit{function information} \text{ environment}(i.e., \Sigma) \tag{19c}$$

From the structure of t (eq. (19a)), term reduction ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [RCALL$_1$] From this rule we know that $\alpha = \tau$, and

$$t' = f\,(v_1, \ \ldots, \ v_{k-1}, \ e'_k, \ \ldots, \ e_n)$$

$$e_k \to e'_k \tag{19d}$$

Applying eq. (19b) (for $i = j$) and eq. (19d) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e'_j : T_j \tag{19e}$$

Using eq. (19b) and eq. (19e) in the rule [TRECUNKNOWNCALL], results in

$$\Delta \cdot \emptyset \vdash^{w} S \rhd f\,(v_1, \ \ldots, \ v_{k-1}, \ e'_k, \ \ldots, \ e_n) : T \lhd S'$$

This holds since $\mathbf{after}(S, \tau) = S$ and $\mathbf{after}(\Delta, \tau, S) = \Delta$.

2. [RCALL$_2$] From the rule, we know that $\alpha = f_n$ and

$$t = f\,(\iota, v_2, \ \ldots, \ v_n) \tag{19f}$$

$$w = \iota \tag{19g}$$

$$t' = \bar{t}\,[\iota/y]\,[v_2, \ \cdots, \ v_n/x_2, \ \ldots, \ x_n]$$

By eq. (19f) and the Value Typing Lemma we know that $\emptyset \vdash_{\text{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (19c) to the Substitution Lemma results in

$$(\Delta, f_n : S) \cdot \left(\widetilde{x} : \widetilde{T}\right) \vdash^{y[\iota/y]} S \rhd \bar{t}\,[\iota/y] : T \lhd S' \tag{19h}$$

where by the *Variable Substitution* Definition and eq. (19g), $y\,[\iota/y] = \iota = w$. Applying the Substitution Lemma repeatedly to eqs. (19b) and (19h), results in

$$(\Delta, f_n : S) \cdot \emptyset \vdash^w S \rhd \bar{t}\,[\iota/y]\,[v_2, \ldots, v_n/x_2, \ldots, x_n] : T \lhd S'$$

where $\mathbf{after}(S, f_n) = S$ and $\mathbf{after}(\Delta, f_n, S) = (\Delta, f_n : S)$, as required.

[**TCase**] From the rule, we know that for some type U,

$$t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} \tag{20a}$$

$$\emptyset \vdash_{\text{exp}} e : U \tag{20b}$$

$$\forall i \in I \begin{cases} \vdash^w_{\text{pat}} p_i : U \ \rhd\ \Gamma'_i & \text{(20c)} \\ \Delta \cdot \Gamma'_i \vdash^w S \rhd t_i : T \lhd S' & \text{(20d)} \end{cases}$$

By eq. (20a), term reduction, $t \xrightarrow{\alpha} t'$, can be derived using two rules, so we consider two sub-cases:

1. [**RCase$_1$**] From the rule we know that $t' = \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end}$, and from the premise we know that

   $$e \to e' \tag{20e}$$

   By eqs. (20b) and (20e) and the Preservation (Expressions) Lemma, we get

   $$\emptyset \vdash_{\text{exp}} e' : U \tag{20f}$$

   Using eqs. (20c), (20d), (20f), and [TCase], we get

   $$\Delta \cdot \emptyset \vdash^w S \rhd \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} : T \lhd S'$$

   which holds as expected since $\mathbf{after}(S, \tau) = S$ and $\mathbf{after}(\Delta, \tau, S) = \Delta$.

2. [**RCase$_2$**] From the rule, we know that $t = \texttt{case}\ v\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end}$, $e = v$ and for some $j \in I$,

   $$\mathbf{match}(p_j, v) = \sigma \text{ where } \sigma = [v_1, \ldots, v_n/x_1, \ldots, x_n] \tag{20g}$$

   $$t' = t_j\sigma \tag{20h}$$

   By eqs. (20b), (20c) and (20g) and lemma 15, we know that $\Gamma'_j = x_1 :$

$T_1, \ldots, x_n : T_n$ and

$$\emptyset \vdash_{\text{exp}} v_k : T_k \text{ for all } k \in 1..n \tag{20i}$$

Then, by repeatedly applying the Substitution Lemma to eq. (20i), (20d for $i = j$), we get

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright t_j\sigma : T \triangleleft S'$$

This holds since $\textbf{after}(S, \tau) = S$ and $\textbf{after}(\Delta, \tau, S) = \Delta$. □

4.4 Conclusion

We have shown that the session typing system created for a subset of Elixir from Chapter 3 obeys the session fidelity property. Meaning, that a well-typed program will always remain well-typed when transitioning. More importantly, in the context of session types, a public function decorated with a session type (*i.e.,* uses `@session` or `@dual` annotations) will perform the actions depicted by the session type, and fully consume it if the function terminates. The goal of Objective O1 was to provide a solid formal foundation in the form of a type system. This objective was partially addressed in Chapter 3 where a (session) type system was introduced. Throughout this chapter, we fulfilled the remaining part of Objective O1 by formally analysing the type system. Further formal analyses can be carried out, such as the proving the progress property, which go beyond the scope of this study.

5. Elixir Implementation

In Chapter 3, we presented a session type system for Elixir. In this chapter, we realise the type system as a proof-of-concept implementation, where we create a type checker for the Elixir language, called ElixirST. Its source code and relevant documentation can be found at

https://github.com/gertab/ElixirST

We also take a look at how this tool fits within the Elixir compilation pipeline, and see how it maps to the formal model. Then, a case study that applies ElixirST to a different scenario is illustrated.

This chapter attempts to fulfil Objective O2. This objective requires that a session type implementation is developed in Elixir, that mirrors the type system formalised in Chapter 3. Furthermore, this objective also demands that session types should be added in a way that is not disruptive to the developer's workflow, thus developers will be allowed to keep using established idioms and patterns from the original language. Throughout this chapter, we will evaluate if this objective is reached. It depends on whether practical concurrency issues are flagged (and the developer is warned) at compile-time. It also depends on how well ElixirST integrates within the existing Elixir workflow.

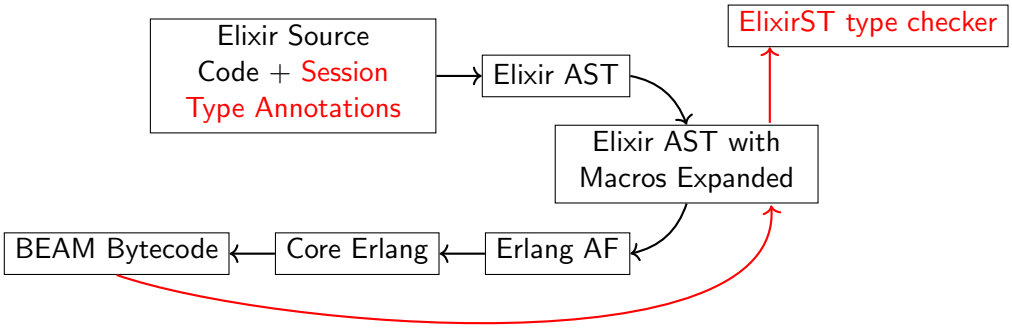


Figure 5.1: Stages of Elixir compilation along with the session type implementation (in red)

5.1 Integration within Elixir

In order to achieve Objective O2, we build ElixirST using existing Elixir mechanisms, which preserves the original Elixir language and its compiler. We do so by exploiting the compilation pipeline, where we inject our session typing analysis as an additional layer within this pipeline.

The compilation pipeline for Elixir is shown in Figure 5.1 (black arrows), which transforms the Elixir source code into executable BEAM bytecode. As a first step, Elixir source code is parsed into an Abstract Syntax Tree (AST). Then, since Elixir is a macro-full language (*i.e.,* a lot of the features provided are macros), all the non-*special form*[1] macros are expanded into *special form* macros. This new expanded AST is converted into Erlang Abstract Format, and simplified further into Core Erlang. The final step involves compiling the Core Erlang code into BEAM bytecode. This BEAM bytecode is split into different 'chunks' that contain several crucial information. First of all, there is a chunk that contains the executable format, which is used to execute programs on the BEAM (Erlang VM). Moreover, in the `debug_info` chunk,[2] we can find the expanded Elixir AST, obtained earlier, which we will then use to typecheck.

Our implementation is embedded within this compilation pipeline (Figure 5.1,

[1] *Special form* macros are the basic building blocks of the Elixir language, which cannot be expanded further.

[2] https://github.com/erlang/otp/pull/1367

red). Modules can be extended by linking other modules using the **use** macro.[3] We use this feature, whereby the ElixirST typechecker can be injected to any existing Elixir module using this macro, *i.e.,* **use** `ElixirST`. Following this, ElixirST extends existing modules by offering our two new annotations: `@session` and `@dual`. These are used to decorate functions with a session type.

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

The `@session` annotation is used to decorate functions with labelled session types directly. The `@dual` annotation takes a session type label, and computes its dual session type (using Definition 3.1).

To use this newly appended information, we save the session types inside the aforementioned `debug_info` chunk of the BEAM bytecode. This is done using Elixir's `on_definition` compile-time hook. At this point, the BEAM bytecode contains all the information needed for analysis (*i.e.,* the function ASTs and their session types). Therefore, we use another hook (called `after_compile`) to initiate the analysis. First, the session types are converted from strings to usable structures, by creating a lexer and a parser (using the Erlang modules `leex`[4] and `yecc`,[5] respectively) based on the syntax defined in Figure 3.2. Finally, all session-typed functions are typechecked by performing a depth-first, pre-order traversal of their AST (facilitated by Elixir's `prewalk`[6] function), while following the typechecking rules defined in Section 3.3.

ElixirST integrates well within VSCode, which is a common source-code editor. For instance, the ElixirLS language server extension in VSCode compiles Elixir code automatically, and in the process it picks up any error messages that ElixirST generates during typechecking, displaying them in clear view to the developer.

[3] https://elixir-lang.org/getting-started/alias-require-and-import.html#use
[4] https://erlang.org/doc/man/leex.html
[5] https://erlang.org/doc/man/yecc.html
[6] https://hexdocs.pm/elixir/1.12/Macro.html#prewalk/2

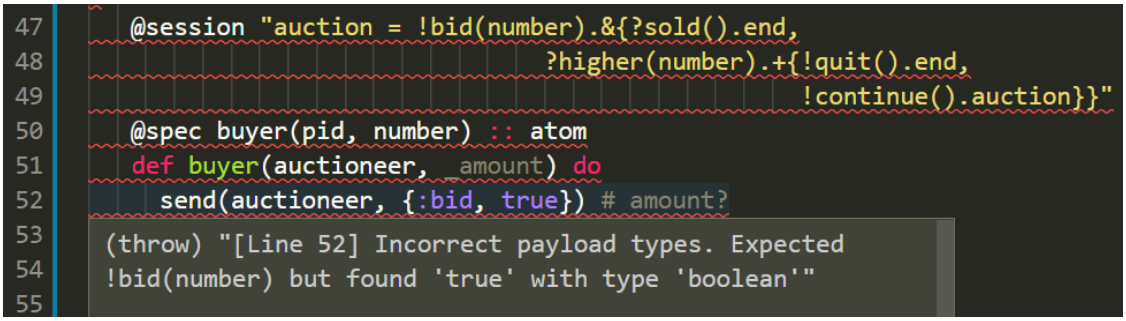


Figure 5.2: Erroneous implementation (from Listing 1.2) in VSCode (running ElixirLS)

Example 5.1. To give some insight of the usability of ElixirST, we refer to the auction example from Listing 1.2, which had an erroneous `buyer` implementation. For example, in Figure 5.2, VSCode flags this implementation as having "incorrect payload types" on *line 52*, since the payload was expected to be a number but a *true* value was found instead. It also cites the line number in the message indicating where the issue originates from.

This shows that ElixirST can be used seamlessly in common development environments (*e.g.*, VSCode), as required in Objective O2. Furthermore, ElixirST is platform independent, as it also works in other environments. For instance, if Elixir code is compiled from a terminal (instead of VSCode), the error messages produced by ElixirST will still be outputted to the developer. ■

5.2 Uniting Elixir and Our Model

We now take a look at how our formal model defined in Chapter 3 maps to the Elixir language from a practical aspect. One should note that ElixirST typechecks programs that follow a subset of the Elixir language, as described by the syntax in Figure 3.3. In this syntax, we require that each function is decorated with the `@spec` annotation, explicitly defining the function specifications. This annotation is already provided by the Elixir language, which is used for code documentation

(by adding type definitions), and is also used by the Dialyzer [14]. The Dialyzer ascertains that the function parameter and return types follow the specifications provided by `@spec`. We exploit this annotation to determine the parameter and return types of all functions, in order to be able to statically typecheck that the terms follow the correct expression types.

Example 5.2. In Listing 1.1, a `buyer` function was defined which takes two parameters: `auctioneer_pid` and `amount`, as follows:

```
@spec buyer(pid, number) :: atom
def buyer(auctioneer_pid, amount) do ... end
```

Then, we use the function specifications (*i.e.,* `@spec buyer...`) to link the parameter names to their types. In this case, the `auctioneer_pid` variable has type `pid`, while the `amount` variable has type `number`. Finally, the function `buyer` returns an atom when it terminates. ■

Another concept that we have to consider is how processes are created. *The Actor Model* Definition (from Section 2.3) dictates that processes can spawn other processes. Our design takes a restricted approach to this definition, since we are considering interactions involving only two parties rather than any number of concurrent parties. To do so, ElixirST provides a procedure that can be used to define new processes, which is abstracted through a custom `spawn` function. This function spawns two session-typed processes that follow their respective session type endpoint. The `spawn` function (Listing 5.1) takes two pairs of parameters: a function reference and the values to initiate their parameters (repeated for the two functions being spawned). The parameter values should exclude the first one, since the first parameter is reserved for the *pid* of the dual process (as defined in Figure 3.3) – this *pid* is set dynamically by the `spawn` function. This function is defined in Listing 5.1.

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

Listing 5.1: ElixirST's `spawn` function

The `spawn` function's workflow is shown in Figure 5.3, where two functions (called `leftFn` and `rightFn`) are spawned. The procedure starts by spawning one 'empty' process, called pre-left (with *pid* `left_pid`, as shown in lines 3–9). Then, another function is spawned, called pre-right (with *pid* `right_pid`, as shown in lines 11–15), where it is passed the `left_pid` value. Then, pre-right sends its *pid* address (`right_pid`) to the pre-left process (lines 5 and 13), thus, both processes become aware of each other's *pid*. Finally, each process calls its respective function (lines 7 and 14), thus continuing as `leftFn` and `rightFn`, respectively.

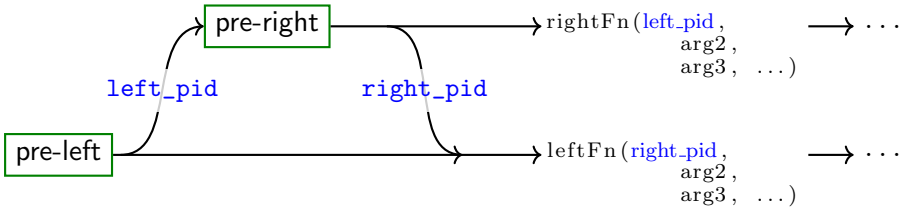


Figure 5.3: Spawning two processes (green boxes represent *spawned* concurrent processes)

Example 5.3. The `Auction` module from Listing 1.3 contains two functions: a `buyer` and an `auctioneer`. The interaction between these two parties can be started using ElixirST's `spawn` function, as follows.

```
{buyer_pid, auctioneer_pid} =
      ElixirST.spawn(&buyer/2, [50], &auctioneer/2, [200])
```

This call results in two processes being spawned, where the `buyer` function is initiated with the parameter value 50, and the `auctioneer` function is initiated with the parameter value 200. This function call immediately returns a tuple containing the *pid* of both processes. ■

5.3 Flexibility

In this section we discuss the flexibility of our solution, thus further satisfying Objective O2. We show that our solution, provided via ElixirST, is not very intrusive to the develop's workflow, since existing design patterns and idioms can still be used. We show this concretely by discussing two common features, as shown in Figures 5.4 and 5.5.

Consider Figure 5.4, where there are two different implementations that achieve the same result, *i.e.,* first a message containing `:A` or `:B` is received, and then a message containing `:C` is sent to a process with *pid* *p*. Although both implementations are valid, the snippet on the right is more efficient and

```
1 receive do
2    {:A} -> send(p, {:C})
3            :ok
4    {:B} -> send(p, {:C})
5            :ok
6 end
```

```
1 receive do
2    {:A} ->
3            :ok
4    {:B} ->
5            :ok
6 end
7 send(p, {:C})
```

Figure 5.4: An Elixir `receive` snippet, along with its equivalent using the fork-join concept (right)

maintainable, since the send statement is written only once rather than duplicated as shown in the snippet on the left-hand side. This is possible through the use of the *fork-join* pattern.

In Figure 5.4, both snippets follow the session type $\&\{?\texttt{A}().!\texttt{C}().\mathsf{end}, ?\texttt{B}().!\texttt{C}().\mathsf{end}\}$, which has a common continuation type $!\texttt{C}().\mathsf{end}$. Several type systems [46] force programs with a common continuation to be structured in a nested fashion, as shown on the left-hand side of Figure 5.4. However, for efficiency and maintainability, the snippet on the right-hand side is more desirable. ElixirST handles this, where the use of the *fork-join* pattern is allowed. This is a simple example, where we only considered a single send statement. However, in larger programs, this could translate to a huge block of code, where the *fork-join* pattern helps to prevent duplicated code.

Another aspect that we consider is the flexibility that macros offer in Elixir. Our design in Section 3.2 focuses on a small part of the language. However, when Elixir is used in practice, a larger range of constructs are typically used by the developer. We show how our type system (Chapter 3), despite typechecking a small macro language, still accepts a much larger language fragment.

For instance, in Listing 1.1 we made use of an `if` construct which is not part of the syntax defined in Figure 3.3. This is possible through a powerful native feature in Elixir, called metaprogramming. In Section 5.1, we saw that Elixir is a macro-full language whereby, during the first part of the compilation, all macros are eliminated

```
1 if value < 100 do
2    send(auctioneer, {:continue})
3    buyer(auctioneer,  amount + 10)
4 else
5    send(auctioneer, {:quit})
6    :ok
7 end
```

```
1 case(value < 100) do
2    false ->
3      :erlang.send(auctioneer, {:quit})
4      :ok
5    true ->
6      :erlang.send(auctioneer, {:continue})
7      buyer(auctioneer, amount + 10)
8 end
```

Figure 5.5: Snippet from Listing 1.1 in normal form (left), along its equivalent with expanded macros (right)

through expansion. This allows us to only consider a small number of constructs. As a result, we get a number of features for free, including existing constructs (*e.g.*, `unless` construct, pipe operator) and custom features. For instance, with Elixir's `defmacro` [40], we can define a custom Domain-Specific Language (DSL), which would still be typecheckable by our type system. Concretely, as shown in Figure 5.5, an `if` construct is expanded into an equivalent `case` construct, so explicit typechecking for an `if` statement is not needed. This shows that ElixirST accepts a more expressive language than the one typechecked in Chapter 3.

In this section, we have shown that ElixirST works with common patterns (*e.g.*, fork-join) and a large fragment of the Elixir language (via macros). This allows developers to program the way they are used to, with ElixirST performing checks in the background. Thus, ElixirST's flexibility helps towards achieving Objective O2.

5.4 Improving the Type System from a Practical Perspective

In this section we identify limitations from the type system (Chapter 3) and its implementation that prevent us from fully achieving Objective O2. Then, we suggest ways of improving them.

Recall the [tBranch] rule (from Figure 3.6 in Section 3.3), which is used to process any incoming messages and match them with the branching session type. This rule allows any messages to be received, like unsolicited ones, which are not sent from the dual process, may end up being picked from the process' mailbox. This is due to the nature of actor systems in the BEAM, which allows any (external) process to exchange messages to other processes, given they have their *pid*. This may be improved by following the design proposed by Mostrous and Vasconcelos [46], which tag individual messages within a session with a unique reference. Thus, messages are cherry-picked from the mailbox only if they have a matching reference.

Another limitation is caused by the [tRecKnownCall] and [tRecUnknownCall]

```
1 @session "!A().!B().end"
2 def fun1(p) do
3    fun2(p)
4    send(p, {:B})
5 end
6
7 defp fun2(p) do
8    send(p, {:A})
9 end
```

```
1 @session "!A().!B().end"
2 def fun1(p) do
3    send(p, {:A})
4    fun2(p)
5 end
6
7 defp fun2(p) do
8    send(p, {:B})
9 end
```

Figure 5.6: An incorrect implementation for a function following the session type !A().!B().end (left), along with an improved version (right)

rules. As explained in Section 3.3, these rules handle function calls. These rules imply that a function call consumes *all* of the remaining session type until it reaches the residual type. Consider the snippets shown in Figure 5.6, where the functiond `fun1` must follow the session type !A().!B().end. The snippet on the left-hand side starts with a (private) function call to `fun2` (line 3). Using the [tRecUnknownCall] rule, it is assumed that `fun2` follows the session type !A().!B().end. When `fun2` is executed (line 8), it is found that it follows a different session type (*i.e.,* !A().end), rather than the one assumed earlier; thus this implementation is deemed ill-typed, even though it is well-behaved. On the other hand, the snippet on the right-hand side performs a tail-call to the private function `fun2`. Rule [tRecUnknownCall] assumes that its session type is !B().end, which matches the behaviour in the function's body (line 8), deeming it well-typed. Due to this limitation, function calls must fully exhaust the remaining session types, and thus, they cannot be followed by any dangling `send` or `receive` constructs. This can be improved by computing the proper session types of *unknown* functions (in [tRecUnknownCall]), or compute the residual session type (in [tRecKnownCall]), using a sort of session type subtraction, although recursion makes this far from trivial.

The final design that we discuss is the branch types within the rules [tBranch] and [tCase]. The rules dictate that each branch (or case) has a common residual session type. Moreover, they also require that each branch (or case) must have a

```
1 if true do
2   :ok
3 else
4   2
5 end
```

```
1 if true do
2   :ok
3 else
4   :ko
5 end
```

Figure 5.7: Snippet showing an incorrect (left) and a correct (right) way of using multiple branches

common expression type T, which further restricts the implementation flexibility. For instance, Figure 5.7 shows a correct and incorrect implementation for an `if` statement. The snippet on the left-hand side is incorrect since the two cases (in lines 2 and 4) have a different type (*i.e.,* atom and number, respectively). The snippet on the right-hand side shows a revised implementation, where both cases end with a common same type (*i.e.,* atom). One avenue that could be used to ease this limitation would be to extend the expression types to use union and intersection types, as formalised by Castagna [47].

5.5 Case Study

In the previous sections, we discussed how ElixirST integrates within Elixir with respect to specific language constructs. In this section we take a different approach, where we present an existing external third-party service, and analyse whether ElixirST is able to be used within a module which interacts with this external service.

Flight System Using a Third-Party API

To fulfil Objective O2, ElixirST must work in real-world applications. To see if this can be achieved, we build an Elixir application that interacts with a third-party flight service API, called Duffel [48]. Duffel offers a real-time flight selling service, where flights can be fetched and booked via a REST application programming interface (API).

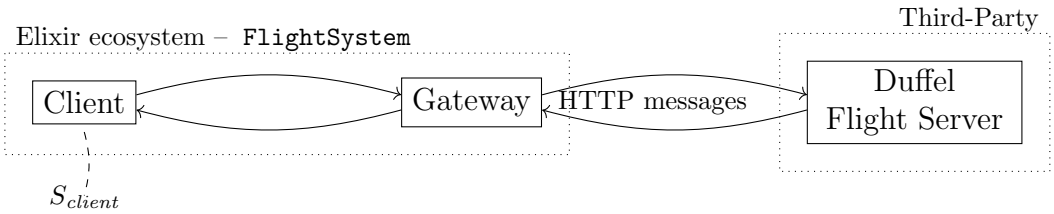


Figure 5.8: Interactions with Duffel API

The application is built as an Elixir module, called `FlightSystem`, which interacts with the Duffel Flight Server, as shown in Figure 5.8. It consists of a client which can request to book flights from the Duffel Server. This server can only accept HTTP messages (*e.g.*, GET or POST request), so we use a gateway which acts as a middleman between the client and the third-party server.

From the Duffel API documentation,[7] we can get a list of API calls that can be made, along with their details. Although each call can be invoked separately, their order is not trivial and certain calls can only be invoked after some requirements have been met. Taking a concrete example, the `Order` API[8] states:

> "Once you have searched for flights by creating an offer request, and you have chosen which offer you want to book, you will then want to create an order."

From this, we realise that before invoking the `Order` API, we have to get the updated list of offers using the `Offer Requests` API. Taking a deeper look at the documentation reveals a form of hierarchy of calls that need to be made in order to book a flight. So, we formalise this behaviour as a session type, S_{client}, which describes the least amount of interactions that need to be made by a client in order to search, browse, and finally, book a flight.

[7] https://duffel.com/docs/api/overview/welcome
[8] https://duffel.com/docs/api/orders/schema

$$S_{client} = \oplus\left\{\begin{array}{l}\texttt{!request}\left(\begin{array}{l}\text{origin: }\mathsf{binary}, \text{destination: }\mathsf{binary},\\ \text{dep\_date: }\mathsf{binary}, \text{class: }\mathsf{atom}, \text{pass\_no: }\mathsf{number}\end{array}\right).S_{offers},\\ \texttt{!cancel}()\end{array}\right.$$

$$S_{offers} = \mathsf{rec}\ \mathsf{Y}\,.\,\&\left\{\begin{array}{l}\texttt{?offer}\left(\begin{array}{l}\text{offer\_no: }\mathsf{number}, \text{total\_amount: }\mathsf{number},\\ \text{currency: }\mathsf{binary}, \text{duration: }\mathsf{number},\\ \text{stops: }\mathsf{number}, \text{segments: }\mathsf{binary}\end{array}\right).S_{details},\\ \texttt{?error}(\mathsf{binary}).S_{client}\end{array}\right.$$

$$S_{details} = \oplus\left\{\begin{array}{l}\texttt{!more\_details}().\&\left\{\begin{array}{l}\texttt{?details}(\ldots).\oplus\left\{\begin{array}{l}\texttt{!make\_booking}(\ldots).\\ \quad\&\{\texttt{?ok}(\ldots), \texttt{?error}(\mathsf{binary})\},\\ \texttt{!cancel}()\end{array}\right\},\\ \texttt{?error}(\mathsf{binary})\end{array}\right\},\\ \texttt{!reject}().\mathsf{Y}\end{array}\right.$$

For clarity, S_{client} is split in two: S_{offers} and $S_{details}$. Furthermore, we add labels to each payload type to make it more apparent what data needs to be transferred – labels are also allowed in ElixirST, given that labels are only used for decorative purposes. We take a brief look at how the client can book a flight. The interaction starts with the client making request to get the available flights (S_{client}). By glancing on S_{client}, one can get more information on what the client needs to include the request details, such as the origin and destination locations. Note that, we use the binary type, which is the type for strings in Elixir. Then, the client starts receiving (and rejecting) one offer at a time, until the client decides to take up an offer (S_{offers}).

To learn more about the flight offer, the client sends a request to get more details (*e.g.*, operating airline and updated price), and awaits the results ($S_{details}$). Once the details are received, the client can decide to either cancel the order, or book the flight. In case of the latter, the booking will be finalised after the client receives back a confirmation code. Throughout this interaction, the server may reply with an error message, which the client also needs to handle (*i.e.,* `?error(binary)`).

```
1  defmodule FlightSystem do
2     use ElixirST
3
4     @session "S_client =  +{!request(origin: binary, destination: binary...}"
5     @spec client(g_pid, binary, binary, binary, atom, number) :: :ok
6     def client(g_pid, origin, destination, dep_date, class, pas_no) do
7        send(g_pid, {:request, origin, destination, dep_date, class, pas_no})
8        IO.puts("Sending request for a flight from #{origin} to...")
9        IO.puts("Waiting for a response from the server...")
10
11       consume_offer(g_pid)
12    end
13
14    @spec consume_offer(pid) :: atom
15    defp consume_offer(g_pid) do
16       receive do
17          {:offer, offer_no, total_amount, currency, dur, stops, segments} ->
18             IO.puts("Offer ##{offer_no}: \n#{currency}#{total_amount}...")
19             accept? = IO.gets("Accept offer ##{offer_no}? y/n: ")
20
21             case accept? do
22                "y\n" -> send(g_pid, {:more_details})
23                   IO.puts("Requesting updated details for offer...")
24                   ...
25                _ -> send(g_pid, {:reject})
26                   consume_offer(g_pid)
27             end
28
29          {:error, message} -> send(pid, {:cancel})
30       end
31    end
32 end
```

Listing 5.2: Session-typed snippet of a flight system written in Elixir

This behaviour formalised by S_{client} is applied in the module `FlightSystem`, which is shown partly in Listing 5.2. `FlightSystem` contains a public function `client`, a private function `consume_offer`, and an omitted public function `gateway`. The `client` function is annotated with the session type S_{client}, thus ElixirST ascertains that the client follows the expected behaviour. The `client` function sends a message containing `:request` and then it calls the private function `consume_offer`. This latter function follows the remaining actions in the session type, *i.e.,* S_{offers}. The dual part of the interaction, the `gateway` function, should follow the dual session type, $\overline{S_{client}}$. This is not enforced, since the `gateway`

function goes beyond the syntax defined in Figure 3.3 (*e.g.*, uses other modules to parse JSON responses) which are not defined by our type system in Figure 3.6. Nevertheless, we can spawn both public functions using our `spawn` function, as follows:

```
ElixirST.spawn(&FlightSystem.client/6,
                      ["MLA", "CDG", "2022-11-24", :economy, 2],
                &FlightSystem.gateway/1, [])
```

To get an insight of what the processes are doing, the `buyer` and `gateway` functions output their current state using `IO.puts`. This output is shown below, where the client makes a request to book a flight from Malta (MLA) to Paris (CDG) and accepts the first offer.

```
>> Sending request for a flight from MLA to CDG on 2022-11-24 for 2 passengers.
>> Waiting for a response from the server...

>> Offer #1:
>> EUR1300.07 (duration: 03:03) Itinerary (0 stops): Iberia IB3167:
>> Malta (MLA) -> Paris (CDG)
>> Accept offer #1? y/n: y

>> Requesting updated details for offer #1

>> Updated details for offer #1 (2 passenger/s):
>> EUR1300.07 (duration: 03:03) Itinerary (0 stops): Iberia IB3167:
>> Malta (MLA) -> Paris (CDG)
>> Departing at 2022-11-24T23:00:00

>> Accepting offer #1...

>> Booking performed successfully. Reference number: HENJK4.
```

In this flight booking example, we showed ElixirST's flexibility and practicality, by successfully integrating it within a system that uses a third-party service. We started by extracting the interaction protocols from the Duffel documentations,

which were left implicit throughout the textual documentation. Even though the interactions were rather complex, we managed to formalise them as the session type S_{client}, which was then used to verify the client side of the interaction using ElixirST. As interactions become more complex, the chance of making a mistake in the implementation increases, so S_{client} helps in identifying these errors. Furthermore, the protocol also becomes more complex to write, so in the future we plan to use a specialised choreography programming language, such as Scribble [49], which can make protocol definitions simpler.

We encountered a limitation in this case study, where the gateway process was left *unverified*, leaving it susceptible to behavioural issues. To fix this, we can expand our type system to accept parts which do not have a known session type at compile-time, by using either gradual session types [50] to verify it statically, or else dynamically, using a runtime monitor on the unverified part, similar to the work by Bartolo Burlò *et al.* [25].

Remark. *The flight booking case study provided an unbiased view showing ElixirST's practicality when applied in a real-world example. However, in Appendix C we present a simple case study where both sides of an interaction are statically verified, by building a 'counter system'.* ■

5.6 Discussion

To guarantee its correctness, ElixirST has been developed alongside unit tests. The tests in ElixirST cover around 60.5% of the Elixir code – this figure was obtained using the `ExCoveralls` tool. Although this coverage leaves a significant portion untested, we further ascertained ElixirST's correctness from a formal aspect. ElixirST is based on the type system from Chapter 3, which has been analysed thoroughly in Chapter 4, where certain properties were proven.

ElixirST was developed to be minimally disruptive in the developer's workflow, while being informative at the same time. This was achieved by providing the

ability to decorate existing Elixir functions with a session type specification, using annotation. This information is then used to verify the functions' correctness via typechecking, all of which happens transparently in the background during compilation. Moreover, as shown in Figure 5.2, our implementation integrates well within modern source-code editors (*e.g.*, VSCode), where issues are flagged immediately during development.

5.7 Conclusion

In this chapter, we implemented the type system from Chapter 3 as a proof-of-concept type checker, called ElixirST. Objective O2 was addressed by showing that ElixirST works within an existing development environment (*e.g.*, VSCode) as shown in Example 5.1, where erroneous implementations that do not follow a session type, were flagged automatically. Furthermore, we showed that the tool is not intrusive to the developer's workflow since the language accepted by ElixirST is quite expressive and certain existing development patterns are allowed. We also showed that ElixirST can be used in practice, via the case study in Section 5.5. However, some limitations became apparent since we could not verify the whole interaction in the case study, which prevented us from completely fulfilling Objective O2.

ElixirST gives one avenue to statically check concurrent software. As concurrent software becomes more widely implemented, we anticipate that the need for structured interactions also increases. Concretely, we envisage that a developer tasked with the creation of an Elixir module, will not just be given the public function specifications, but also an interaction protocol (as a session type) that each function must follow. Then, the developer can implement the functions at his discretion, making use of any private functions needed, as long as the session types are observed.

In the following chapter, we compare our type checker with other implementations.

6. Related Work

Elixir offers limited tools when it comes to correctness guarantees. External tools, for instance, model checkers [51] or runtime verifiers [52] which are built for Erlang, but may be migrated and used in the Elixir language. However, in practice developers tend to rely on test-driven tools, such as Elixir's unit testing framework (called ExUnit[1]). To our knowledge, ElixirST is the first session type checker for the Elixir language. In this chapter, we compare this tool and its design to other session type systems for different languages, as shown in Table 6.1.

6.1 Session Types for Actor-Based Languages

Erlang The closest work to ours was carried out by Mostrous and Vasconcelos [46], where a static session type system was designed to discipline message-passing for a fragment of the Core Erlang language. It works by tagging each sent message with a unique reference, which is the used to identify it at the receiving end. This is achieved by using correlation sets, which then allows for multiple binary sessions to take place concurrently.

The work by Mostrous and Vasconcelos tackles session types for Core Erlang from a theoretical perspective only, thus their algorithmic type system was not realised as a tool. In contrast, our type system takes a more practical approach,

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

	Session Types		*Checking*		*Language*	*Concurrency via*	
	Binary	*Multiparty*	*Static*	*Dynamic*		*Actors*	*Channels*
Mostrous and Vasconcelos [46]	✓		✓		Erlang	✓	
Fowler [28]		✓		✓	Erlang	✓	
Neykova and Yoshida [53, 54]		✓		✓	Python	✓	
Scalas and Yoshida [20]	✓		✓		Scala	✓	✓
Scalas *et al.* [55, 56]	✓		✓		Dotty	✓	✓
Bartolo Burlò *et al.* [25]	✓			✓	Scala		✓
Harvey *et al.* [45]		✓	✓		Ensemble	✓	
Pucella and Tov [57]	✓		✓		Haskell		✓
Kokke and Dardha [58]	✓		✓		Haskell		✓
Padovani [21]	✓		✓		OCaml		✓
Melgratti and Padovani [27]	✓		✓		OCaml		✓
Imai *et al.* [22]	✓		✓		OCaml		✓
Imai *et al.* [59]		✓	✓		OCaml		✓
Jespersen *et al.* [60]	✓		✓		Rust		✓
Kokke [23]	✓		✓		Rust		✓
Lagaillardie *et al.* [24]		✓	✓		Rust		✓
ElixirST	✔		✔		**Elixir**	✔	

Table 6.1: Comparison of our work with other implementations

where it was implemented as a type checker. It also accepts a more expressive language. Specifically, our design allows: variable bindings (*e.g.*, in let statements), expressions (*e.g.*, addition operation), inductive types (*e.g.*, tuples and lists), infinite computation via recursion and explicit protocol definition.

A concrete session type implementation for the Erlang language was created by Fowler [28]. This implementation uses *multiparty* session types (MPST) [61], which allows more than two parties to communicate in a single interaction. The MPST protocols are written using the Scribble protocol language [49], which converts one protocol into smaller protocols which each party should follow. The implementation by Fowler uses these protocols to dynamically verify the actor communication in Erlang, using runtime monitors. This allows for more flexibility, since processes use Erlang/OTP behaviours (*e.g.*, `gen_server`, which structures actors in a hierarchical manner), which are then verified with respect to a session

type. This implementation also accounts for Erlang's *let it crash* philosophy, where processes may fail while executing. Fowler [28] extends the work by Neykova and Yoshida [53]. In contrast, our work, albeit built on a more limited scale (due to Elixir's dynamic nature), provides static guarantees, where issues are flagged at pre-deployment stages.

Python Neykova and Yoshida [53, 54] presented one of the earliest multiparty session type implementations for a dynamically typed language, in this case Python (using the *Cell* actor framework). Similar to [28], protocols are defined using the Scribble language, and runtime monitors are used to dynamically check the interactions. Python annotations are used to assign protocols (*i.e.,* `@protocol`) and roles (*i.e.,* `@role`) to Python classes and functions, respectively. This is similar to our work, where we use annotations (*e.g.*, the `@session`, `@dual`).

Scala Scalas and Yoshida [20] applied binary session types to the Scala language, where session types are abstracted as Scala classes. Then, protocol fidelity is verified using Scala's compiler, which will complain if the implementations do not follow the specified protocols. This work checks for linearity at runtime, *i.e.,* an implementation has to fully exhaust a protocol exactly once. Another implementation was done by Scalas *et al.* [55, 56], where session types were added in Scala 3 (called Dotty). This design utilises dependent function types to verify programs at compile-time. It uses model checking to ensure that the protocols are followed.

Bartolo Burlò *et al.* [25] applied session types in a hybrid setting in Scala. This work typechecks one part of an interaction statically using the work by Scalas and Yoshida [20], and checks the other side of the interaction dynamically. The latter is achieved by synthesising runtime monitors from a given session type. This allows one interacting party to behave as a 'black-box' where its implementation may not be statically checked. This concept is an ideal extension of our work; for instance, in the case study of Section 5.5, one side of the interaction was left

unverified; using Bartolo Burlò *et al.*'s hybrid setting would allow us to attach a runtime monitor to check the behaviour of the unverified side.

Other Actor Languages Harvey *et al.* [45] present a new actor-based language, called EnsembleS, which offers session type as first-class features in the language. EnsembleS statically verifies implementations with respect to session types, while still allowing for adaptation of *new* actors at runtime, given that the actors are compatible with the pre-existing protocols. Thus, actors can be terminated and discovered at runtime, while still maintaining static correctness.

An unconventional approach was taken by De'Liguoro and Padovani [62], where they designed a type system for the processes' mailbox directly. They introduced a *mailbox calculus* which considers mailboxes as first-class citizens. Then, the mailboxes are typechecked to ensure that the processes are free from behavioural issues, such as deadlocks.

We can observe that from the session type systems that we discussed, a common trend emerges. Most dynamically-typed languages are checked dynamically using runtime monitors [28, 53, 54]. Our implementation deviates this pattern, by applying static checking of session types for a subset of Elixir. Another exception is [46], where Mostrous and Vasconcelos design a static type system for Core Erlang, however, there is no implementation for it.

6.2 Session Types for Channel-Based Languages

Our work is built on Elixir, which is actor-based, however, we overview a few implementations that use channel-based message-passing.

Haskell and OCaml One of the earliest binary session type implementations was carried out in Haskell by Pucella and Tov [57], where they use features available from Haskell, such as indexed parameterised monad. Another Haskell

implementation was done by Kokke and Dardha [58]. In this case they make use of Linear Haskell and priorities to guarantee linearity at compile-time.

Padovani [21] create an OCaml library called FuSe, which statically verifies binary session types. However, it checks for linearity at runtime, akin to [20]. FuSe was further extended by Melgratti and Padovani [27] to include contracts. Contracts are assertions that are checked dynamically. In [27], contracts are written in OCaml, which are inserted directly within the existing code, thus forming an inline monitor that flags issues at runtime. Another OCaml implementation was presented by Imai *et al.* [22] in which they check binary session types statically. This work uses parametric polymorphism provided by the language. This project [22] was later extended [59] to support multiparty session types, utilising global combinators.

Rust Jespersen *et al.* [60] created a binary session type implementation for the Rust language. They leverage Rust's affine type system to ensure static checks. However, the implementation has some limitations, *e.g.*, branches and choices are limited to binary options, and sessions can be unsafely dropped prematurely. In this work, annotations (*e.g.*, `#[must_use]`) are also used to add further checks to functions. This work [60] was improved by Kokke [23] by adding support for early cancellation of sessions. This was done by utilising Exceptional GV [63], which by design, safely handles failures via exceptions. In turn, Lagaillardie *et al.* [24] extended [23] to support multiparty session types.

The approaches seen in Rust [23, 24, 60], OCaml [21, 22, 27, 59] and Haskell [57, 58] rely heavily on type-level features of the language, which are all statically typed. As a result, these implementations do not readily apply in the Elixir language, which is a dynamically typed language.

6.3 Type System for Elixir

Elixir is a dynamically typed language, so in order to statically verify the concurrent aspect of Elixir, we also have to typecheck the functional part of the language. Other works have attempted the latter, typically overlooking the concurrent part.

Cassola *et al.* [44] presented a gradual type system for Elixir. It statically typechecks Elixir modules, using a gradual approach, where some terms may be left with an unknown expression type. Similarly to our work, [44] uses the `@spec` annotation to get the parameter types when typechecking the functional part of Elixir. Their type system analyses the Elixir AST *without* the macros expanded, which limits the flexibility of the language accepted. In contrast, our work typechecks the AST with *expanded* macros, so as a result we have to typecheck less constructs while accepting more for free, *e.g.*, by typechecking the **`case`** construct, we are implicitly accepting the **`if`** and **`unless`** constructs since these are expanded into a **`case`** construct.

Although few tools attempt to statically typecheck the Elixir language, there were some attempts for other BEAM languages. For example, Harrison [64] statically checks Core Erlang for errors within the messages being transferred. For instance, it checks that each send statement matches a received statement. However, it takes a more fine-grained approach, since it does not verify the messages with respect to a general protocol.

Another implementation by Rajendrakumar and Bieniusa [65] typecheck Erlang using a bidirectional type system, where typechecking and type inference is performed simultaneously. This work, similar to other Erlang static type systems [66, 67], analyses the functional part of Erlang programs, omitting any concurrency checks.

In the following chapter, we summarise the conclusion of this study.

7. Conclusion

In this study we presented an approach of how binary session types could be used to analyse the concurrent part of the existing Elixir language. The integration of session types in Elixir was first presented in Chapter 3 through a formal type system and analysed formally in Chapter 4. Then, the type system was implemented as a practical static type checker in Chapter 5. The results obtained from this study will in this conclusion be compared with the general objectives of Section 1.1.

The goal of Objective O1 was to provide a formal foundation for Elixir programs in the form of a type system and operational semantics, where we ensure that concurrent interactions in Elixir follow a certain protocol. In Chapter 3, we presented a subset of the Elixir language which was small enough to be formally analysed, but flexible and usable in practice too. A type system was built for this core language in Section 3.3 to typecheck public functions in a module. During typechecking, we ascertain that they follow precisely their declared protocol, expressed via session types. This type system has been further validated in Chapter 4, where we proved the Session Fidelity Theorem. This theorem guarantees that a well-typed program remains free from certain behavioural issues because they continue to follow their prescribed protocol during execution. Concretely, *communication mismatches* and certain kinds of *deadlocks* do not arise at runtime.

The goal of Objective O2 was to realise the aforementioned formal type system as a practical tool, in a non-intrusive manner for the developer. We achieved

this to a certain degree in Chapter 5 by building a tool, called ElixirST, which is able to statically typecheck public functions with respect to their session types. We focused on integrating this tool seamlessly within the developer's workflow, by reusing existing Elixir patterns, such as the *fork-join* pattern, described in Section 5.3. Concretely, ElixirST allows public functions in Elixir programs to be decorated with session types, using the `@session` or `@dual` annotations. By the case study in Section 5.5, we showed that ElixirST is flexible enough to be used in practice. However, it has some limitations that prevents us from fully reaching Objective O2. Specifically, one side of the interaction (in the case study) could not be statically checked due to the nature of the type system presented in Chapter 3.

The aim of this study was to improve the development of concurrent code, specifically within the Elixir language. To analyse this aim, we discussed the results in relation to the objectives, where Objective O1 was reached, and Objective O2 was mostly fulfilled, other than the aforementioned limitations. Thus, we conclude that our tool, in line with the aim, does help in improving the development of concurrent Elixir code, by typechecking Elixir functions with respect to a protocol transparently in the background.

Throughout this study, we realised the increasing importance of getting message-passing concurrency *right*. An emerging practice is to use session types, which are used to ascertain that the behaviour of concurrent programs follows some pre-defined protocol. Several research groups and workshops [68–70] are focusing on this area, where they attempt to bridge theoretical behavioural systems with a practical counterpart, akin to how we structured our work.

Thanks We thank the AGERE workshop [29] and the STARDUST[1] research group [71] for their useful feedback and engaging discussions on previous iterations of this work.

[1]STARDUST (Session Types for Reliable Distributed Systems) is a research group working on integrating session types within actor systems.

7.1 Future Work

Retrofitting session types in the Elixir language can be improved in a number of ways. Given that we presented session types in a binary setting, a natural extension would be to extend them to handle more than two concurrent parties, using *multiparty* session types [61]. Moreover, we currently link session types to individual functions within a module – these could be extended to work *across*, thus being able to refer to the session types from external modules. A limitation that we encountered in Section 5.5 was that a third-party process could not be statically checked, leaving a vulnerability. This issue could be averted if we provide a hybrid form of checking, akin to the work by Bartolo Burlò *et al.* [25], where checking is extended to the unverified side via runtime monitors. Another aspect that we did not consider is Erlang's *let it crash* design, which handles errors by letting processes crash — in our design if a process fails after an interaction has started, it could cause non-deterministic results. We could account for failures by providing exception handling, similar to the work by Fowler *et al.* [63]

In this study we built a static type system for a subset of Elixir, which is an inherently dynamic language. To increase the usability of this project, ideally, the chosen subset must be extended until is becomes close or equivalent to the full language. Due to its dynamic properties, we may need to accept parts with an *unknown* type. Gradual typing [44, 50] could be used to typecheck these untyped parts. To typecheck more functional constructs, we need to implement more flexible expression types, such as union, intersection and negation types, similar to the work formalised by Castagna [47].

The last aspect that we consider is the formal analysis. Although in Chapter 4, we proved session fidelity, we still have to prove other properties, such as the *progress* property, to be able to guarantee *soundness* of our type system. Furthermore, we can utilise interactive formal theorem provers (*e.g.*, Coq [72] and Agda [73]) to verify the correctness of the theorems.

References

[1] P. Gammie, "Concepts, Techniques, and Models of Computer Programming," *J. Funct. Program.*, vol. 19, no. 2, pp. 254–256, 2009. DOI: 10.1017/S0956796808007028.

[2] "Effective Go - The Go Programming Language," Accessed: 25-March-2021. [Online]. Available: https://go.dev/doc/effective_go\#sharing.

[3] K. Cox-Buday, *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media, Inc., 2017.

[4] "Akka documentation," Accessed: 5-January-2022. [Online]. Available: https://doc.akka.io/docs/akka/current/general/actor-systems.html.

[5] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2019.

[6] "The Swift Programming Language: Swift 5.6," Accessed: 1-March-2022. [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html.

[7] D. Thomas, *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. Pragmatic Bookshelf, 2018.

[8] O. Inverso, H. C. Melgratti, L. Padovani, C. Trubiani, and E. Tuosto, "Probabilistic Analysis of Binary Sessions," in *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, I. Konnov and L. Kovács, Eds., ser. LIPIcs, vol. 171, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 14:1–14:21. DOI: 10.4230/LIPIcs.CONCUR.2020.14.

[9] "Akka typed," Accessed: 3-November-2012. [Online]. Available: https://doc.akka.io/docs/akka/2.5/typed/index.html.

[10] M. McGranaghan and E. Bendersky, "Go by Example: Channels," Accessed: 8-April-2021. [Online]. Available: https://gobyexample.com/channels.

[11] The Elixir Team, "Processes - The Elixir Programming Language," Accessed: 15-December-2020. [Online]. Available: https://elixir-lang.org/getting-started/processes.html.

[12] J. Armstrong, *Programming Erlang - Software for a Concurrent World*. 2013.

[13] T. Lindahl and K. Sagonas, "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*, A. Bossi and M. J. Maher, Eds., ACM, 2006, pp. 167–178. DOI: 10.1145/1140335.1140356.

[14] K. Sagonas and D. Luna, "Gradual Typing of Erlang Programs: A Wrangler Experience," in *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, Victoria, BC, Canada: ACM Press, Sep. 2008, pp. 73–82. DOI: 10.1145/1411273.1411284.

[15] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, *et al.*, "Behavioral types in programming languages," *Found. Trends Program. Lang.*, vol. 3, no. 2-3, pp. 95–230, 2016. DOI: 10.1561/2500000031.

[16] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, *et al.*, "Foundations of Session Types and Behavioural Contracts," *ACM Comput. Surv.*, vol. 49, no. 1, 3:1–3:36, 2016. DOI: 10.1145/2873052.

[17] K. Honda, "Types for Dyadic Interaction," in *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, E. Best, Ed., ser. Lecture Notes in Computer Science, vol. 715, Springer, 1993, pp. 509–523. DOI: 10.1007/3-540-57208-2_35.

[18] A. L. Voinea and S. J. Gay, "Benefits of session types for software development," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH 2016, Amsterdam, Netherlands, November 1, 2016*, C. Anslow, T. D. LaToza, and J. Sunshine, Eds., ACM, 2016, pp. 26–29. DOI: 10.1145/3001878.3001883.

[19] D. Castro-Perez, R. Hu, S. Jongmans, N. Ng, and N. Yoshida, "Distributed Programming using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 29:1–29:30, 2019. DOI: 10.1145/3290342.

[20] A. Scalas and N. Yoshida, "Lightweight Session Programming in Scala," in *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, S. Krishnamurthi and B. S. Lerner, Eds., ser. LIPIcs, vol. 56, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 21:1–21:28. DOI: 10.4230/LIPIcs.ECOOP.2016.21.

[21] L. Padovani, "A Simple Library Implementation of Binary Sessions," *J. Funct. Program.*, vol. 27, e4, 2017. DOI: 10.1017/S0956796816000289.

[22] K. Imai, N. Yoshida, and S. Yuen, "Session-ocaml: A session-based library with polarities and lenses," *Sci. Comput. Program.*, vol. 172, pp. 135–159, 2019. DOI: 10.1016/j.scico.2018.08.005.

[23] W. Kokke, "Rusty Variation: Deadlock-free Sessions with Failure in Rust," in *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, Eds., ser. EPTCS, vol. 304, 2019, pp. 48–60. DOI: 10.4204/EPTCS.304.4.

[24] N. Lagaillardie, R. Neykova, and N. Yoshida, "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*, S. Bliudze and L. Bocchi, Eds., ser. Lecture Notes in Computer Science, vol. 12134, Springer, 2020, pp. 127–136. DOI: 10.1007/978-3-030-50029-0_8.

[25] C. Bartolo Burlò, A. Francalanza, and A. Scalas, "On the Monitorability of Session Types, in Theory and Practice," in *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, A. Møller and M. Sridharan, Eds., ser. LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 20:1–20:30. DOI: 10.4230/LIPIcs.ECOOP.2021.20.

[26] R. Neykova, N. Yoshida, and R. Hu, "SPY: Local Verification of Global Protocols," in *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, A. Legay and S. Bensalem, Eds., ser. Lecture Notes in Computer Science, vol. 8174, Springer, 2013, pp. 358–363. DOI: 10.1007/978-3-642-40787-1_25.

[27] H. C. Melgratti and L. Padovani, "Chaperone Contracts for Higher-Order Sessions," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, 35:1–35:29, 2017. DOI: 10.1145/3110279.

[28] S. Fowler, "An Erlang Implementation of Multiparty Session Actors," in *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, M. Bartoletti, L. Henrio, S. Knight, and H. T. Vieira, Eds., ser. EPTCS, vol. 223, 2016, pp. 36–50. DOI: 10.4204/EPTCS.223.3.

[29] G. Tabone and A. Francalanza, "Session Types in Elixir," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based*

on Actors, Agents, and Decentralized Control, AGERE 2021, Virtual Event / Chicago, IL, USA, 17 October 2021, E. Castegren, J. D. Koster, and S. Fowler, Eds., ACM, 2021, pp. 12–23. DOI: 10.1145/3486601.3486708.

[30] E. Castegren, J. D. Koster, and S. Fowler, Eds., *AGERE 2021: Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, Virtual Event / Chicago, IL, USA, 17 October 2021*, ACM, 2021, ISBN: 978-1-4503-9104-7. DOI: 10.1145/3486601.

[31] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002, ch. 20, ISBN: 978-0-262-16209-8.

[32] R. Harper, *Practical Foundations for Programming Languages (2nd. Ed.)* Cambridge University Press, 2016, ISBN: 9781107150300.

[33] R. Milner, "A Theory of Type Polymorphism in Programming," *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, 1978. DOI: 10.1016/0022-0000(78)90014-4.

[34] R. M. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976. DOI: 10.1145/360248.360251.

[35] C. P. Breshears, *The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, 2009.

[36] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Prentice Hall, 2016.

[37] "Cuda C++ Programming Guide," Accessed: 25-March-2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[38] C. Hewitt, P. B. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *IJCAI*, William Kaufmann, 1973, pp. 235–245.

[39] G. A. Agha, *ACTORS - a model of concurrent computation in distributed systems* (MIT Press series in artificial intelligence). MIT Press, 1990, ISBN: 978-0-262-01092-4.

[40] E. Stenman, "The BEAM Book," Accessed: 29-October-2021. [Online]. Available: https://blog.stenmans.org/theBeamBook/.

[41] "Twitter (github)," Accessed: 25-March-2021. [Online]. Available: https://github.com/pinterest.

[42] "Andy Tran (Twitter)," Accessed: 20-January-2022. [Online]. Available: https://web.archive.org/web/20220122094605/https://twitter.com/nivenhuh/status/1483895710932078593.

[43] S. J. Gay and M. Hole, "Subtyping for session types in the pi calculus," *Acta Informatica*, vol. 42, no. 2-3, pp. 191–225, 2005. DOI: 10.1007/s00236-005-0177-z.

[44] M. Cassola, A. Talagorria, A. Pardo, and M. Viera, "A gradual type system for Elixir," *Journal of Computer Languages*, vol. 68, p. 101 077, 2022, ISSN: 2590-1184. DOI: 10.1016/j.cola.2021.101077.

[45] P. Harvey, S. Fowler, O. Dardha, and S. J. Gay, "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)*, A. Møller and M. Sridharan, Eds., ser. LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 10:1–10:30. DOI: 10.4230/LIPIcs.ECOOP.2021.10.

[46] D. Mostrous and V. T. Vasconcelos, "Session Typing for a Featherweight Erlang," in *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, W. D. Meuter and G. Roman, Eds., ser. Lecture Notes in Computer Science, vol. 6721, Springer, 2011, pp. 95–109. DOI: 10.1007/978-3-642-21464-6_7.

[47] G. Castagna, "Programming with union, intersection, and negation types," *CoRR*, vol. abs/2111.03354, 2021. arXiv: 2111.03354.

[48] "Duffel," Accessed: 10-March-2022. [Online]. Available: https://duffel.com/.

[49] K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida, "Scribbling Interactions with a Formal Foundation," in *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, R. Natarajan and A. K. Ojo, Eds., ser. Lecture Notes in Computer Science, vol. 6536, Springer, 2011, pp. 55–75. DOI: 10.1007/978-3-642-19056-8_4.

[50] A. Igarashi, P. Thiemann, Y. Tsuda, V. T. Vasconcelos, and P. Wadler, "Gradual session types," *J. Funct. Program.*, vol. 29, e17, 2019. DOI: 10.1017/S0956796819000169.

[51] Q. Guo, J. Derrick, C. B. Earle, and L. Fredlund, "Model-Checking Erlang - A Comparison between EtomCRL2 and McErlang," in *TAIC PART*, ser. Lecture Notes in Computer Science, vol. 6303, Springer, 2010, pp. 23–38. DOI: 10.1007/978-3-642-15585-7_5.

[52] D. P. Attard, L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, *et al.*, "Better Late Than Never or: Verifying Asynchronous Components at Runtime," in *FORTE*, ser. Lecture Notes in Computer Science, vol. 12719, Springer, 2021, pp. 207–225. DOI: 10.1007/978-3-030-78089-0_14.

[53] R. Neykova and N. Yoshida, "Multiparty Session Actors," *Log. Methods Comput. Sci.*, vol. 13, no. 1, 2017. DOI: 10.23638/LMCS-13(1:17)2017.

[54] R. Neykova and N. Yoshida, "Multiparty session actors," in *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014*, A. F. Donaldson and V. T. Vasconcelos, Eds., ser. EPTCS, vol. 155, 2014, pp. 32–37. DOI: 10.4204/EPTCS.155.5.

[55] A. Scalas, N. Yoshida, and E. Benussi, "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*, J. I. Brachthäuser, S. Ryu, and N. Nystrom, Eds., ACM, 2019, pp. 27–31. DOI: 10.1145/3337932.3338812.

[56] A. Scalas, N. Yoshida, and E. Benussi, "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*, K. S. McKinley and K. Fisher, Eds., ACM, 2019, pp. 502–516. DOI: 10.1145/3314221.3322484.

[57] R. Pucella and J. A. Tov, "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*, A. Gill, Ed., ACM, 2008, pp. 25–36. DOI: 10.1145/1411286.1411290.

[58] W. Kokke and O. Dardha, "Deadlock-Free Session Types in Linear Haskell," *CoRR*, vol. abs/2103.14481, 2021. arXiv: 2103.14481.

[59] K. Imai, R. Neykova, N. Yoshida, and S. Yuen, "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)*, R. Hirschfeld and T. Pape, Eds., ser. LIPIcs, vol. 166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 9:1–9:30. DOI: 10.4230/LIPIcs.ECOOP.2020.9.

[60] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, "Session types for Rust," in *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, P. Bahr and S. Erdweg, Eds., ACM, 2015, pp. 13–22. DOI: 10.1145/2808098.2808100.

[61] K. Honda, N. Yoshida, and M. Carbone, "Multiparty Asynchronous Session Types," *J. ACM*, vol. 63, no. 1, 9:1–9:67, 2016. DOI: 10.1145/2827695. [Online]. Available: https://doi.org/10.1145/2827695.

[62] U. de'Liguoro and L. Padovani, "Mailbox Types for Unordered Interactions," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, T. D. Millstein, Ed., ser. LIPIcs, vol. 109, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 15:1–15:28. DOI: 10.4230/LIPIcs.ECOOP.2018.15.

[63] S. Fowler, S. Lindley, J. G. Morris, and S. Decova, "Exceptional asynchronous session types: Session types without tiers," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 28:1–28:29, 2019. DOI: 10.1145/3290341.

[64] J. Harrison, "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*, N. Chechina and A. Francalanza, Eds., ACM, 2018, pp. 37–48. DOI: 10.1145/3239332.3242765.

[65] N. V. Rajendrakumar and A. Bieniusa, "Bidirectional typing for Erlang," in *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2021, Virtual Event, Korea, August 26, 2021*, S. Aronis and A. Bieniusa, Eds., ACM, 2021, pp. 54–63. DOI: 10.1145/3471871.3472966.

[66] S. Marlow and P. Wadler, "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*, S. L. P. Jones, M. Tofte, and A. M. Berman, Eds., ACM, 1997, pp. 136–149. DOI: 10.1145/258948.258962.

[67] N. Valliappan and J. Hughes, "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*, N. Chechina and A. Francalanza, Eds., ACM, 2018, pp. 49–60. DOI: 10.1145/3239332.3242766.

[68] M. Dezani, R. Kuhn, S. Lindley, and A. Scalas, "Behavioural Types: Bridging Theory and Practice (Dagstuhl Seminar 21372)," *Dagstuhl Reports*, vol. 11, no. 8, M. Dezani, R. Kuhn, S. Lindley, and A. Scalas, Eds., pp. 52–75, 2022, ISSN: 2192-5283. DOI: 10.4230/DagRep.11.8.52.

[69] *Behavioural types: From Theory to Tools*. River Publishers, 2017. DOI: 10.13052/rp-9788793519817.

[70] "BehAPI Workshop @ ETAPS 2019," 2019, Accessed: 18-March-2022. [Online]. Available: https://www.um.edu.mt/projects/behapi/behapi-workshop-etaps-2019/.

[71] "STARDUST: Session Types for Reliable Distributed Systems," Accessed: 15-December-2021. [Online]. Available: https://epsrc-stardust.github.io/.

[72] P. Letouzey, "A New Extraction for Coq," in *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, H. Geuvers and F. Wiedijk, Eds., ser. Lecture Notes in Computer Science, vol. 2646, Springer, 2002, pp. 200–219. DOI: 10.1007/3-540-39185-1_12.

[73] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda - A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., ser. Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

[74] S. Nyström, "A soft-typing system for Erlang," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, B. Däcker and T. Arts, Eds., ACM, 2003, pp. 56–71. DOI: 10.1145/940880.940888.

Appendix

A. Additional Definitions

In this section, we formalise some auxiliary definitions that were used in Chapters 3 and 4.

Definition A.1 (*Free Variables*)**.** The set of free variables is defined inductively as:

$$\mathbf{fv}(e) = \begin{cases} \{x\} & e = x \\ \emptyset & e = b \\ \mathbf{fv}(e_1) \cup \mathbf{fv}(e_2) & e = e_1 \diamond e_2 \text{ or } e = [\, e_1 \mid e_2 \,] \\ \mathbf{fv}(e') & e = \texttt{not}\ e' \\ \cup_{i \in 1..n} \mathbf{fv}(e_i) & e = \{e_1, \ldots, e_n\} \end{cases}$$

$$\mathbf{fv}(t) = \begin{cases} \mathbf{fv}(t_1) \cup (\mathbf{fv}(t_2) \setminus \{x\}) & t = (x = t_1;\ t_2) \\ \cup_{i \in 1..n} \mathbf{fv}(e_i) \cup \mathbf{fv}(w) & t = \texttt{send}\,(w, \{\texttt{:l}, e_1, \ldots, e_n\}) \\ \cup_{i \in I} [\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i})] & t = \texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I}\texttt{end} \\ \cup_{i \in 2..n} \mathbf{fv}(e_i) \cup \mathbf{fv}(w) & t = f\,(w, e_2, \ldots, e_n) \\ \cup_{i \in I} [\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i)] \cup \mathbf{fv}(e) & t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} \end{cases}$$

■

Definition A.2 (*Bound Variables*)**.**

$$\mathbf{bv}(t) = \begin{cases} \emptyset & t = e \text{ or } t = \texttt{send}\,(w, \{\texttt{:l}, \widetilde{e}\}) \text{ or } t = f\,(\widetilde{e}) \\ \{x\} \cup \mathbf{bv}(t_1) \cup \mathbf{bv}(t_2) & t = (x = t_1;\ t_2) \\ \cup_{i \in I}[\mathbf{bv}(t_i) \cup \mathbf{vars}(\widetilde{p_i})] & t = \texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I}\texttt{end} \\ \cup_{i \in I}[\mathbf{bv}(t_i) \cup \mathbf{vars}(p_i)] & t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end} \end{cases}$$ ■

Definition A.3 (*Agree Function*)**.** The boolean function **agree**, returns true if an action α performs the actions allowed by the session type S. The function, denoted as $\mathbf{agree}(S, \alpha)$, is defined for the following cases:

$$\begin{aligned} &\mathbf{agree}(S, \tau) \\ &\mathbf{agree}(S, f_n) \\ &\mathbf{agree}(\oplus\{!\texttt{l}_i(\widetilde{T_i}).S_i\}_{i \in I}, \iota!\,\{\texttt{l}_j, \widetilde{v}\}) \quad \text{where } j \in I \\ &\mathbf{agree}(\&\{?\texttt{l}_i(\widetilde{T_i}).S_i\}_{i \in I}, ?\,\{\texttt{l}_j, \widetilde{v}\}) \quad \text{where } j \in I \end{aligned}$$ ■

Definition A.4 (*Functions Details*)**.** We can extract details (*i.e.,* `params`, `body`, `param_types`, `return_type`, `dual`) from private functions ($\widetilde{P}$) and build a mapping, using set-comprehension, as follows.

$$\texttt{details}(\widetilde{P}) = \left\{ f_n : \begin{bmatrix} \texttt{dual} = y,\ \texttt{params} = \widetilde{x}, \\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{return\_type} = T,\ \texttt{body} = t \end{bmatrix} \;\middle|\; \begin{bmatrix} \texttt{@spec}\ f\left(\mathsf{pid}, \widetilde{T}\right)\ ::\ T \\ \texttt{defp}\ f(y, \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{P} \right\}$$

where f_n is the function name and arity. The function information for public functions, *i.e.,* Σ_{f_n} (as used in [TMODULE]), although implicit, is analogous to the above definition, but instead we consider a single mapping $(f_n : \Omega)$ at a time, rather than multiple mappings. ■

Definition A.5 (*Functions Names and Arity*)**.** Using `functions`(), we can get a set containing all public function names (and arity). This function, is defined using set-comprehension, as follows.

$$\texttt{functions}(\widetilde{D}) = \left\{ \; f_n \;\middle|\; \begin{bmatrix} \texttt{@session}\ldots,\ \texttt{@spec}\ldots \\ \texttt{def}\ f(y,\ \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{D} \; \right\}$$

where $\widetilde{D}$ contains all public functions. ■

Definition A.6 (*All Session Types*)**.** In the context of a module, `defmodule` m `do` $\widetilde{P}$ $\widetilde{D}$ `end`, the function `session` returns the session type corresponding to a function name (and arity). We used an intermediary environment, Θ, which maps all the public function names (and arity) to their session type.

$$\Theta = \left\{ \; f_n : S \;\middle|\; \begin{bmatrix} \texttt{@session}\ \text{“}S\text{”},\ \texttt{@spec}\ldots \\ \texttt{def}\ f(y,\ x_2,\ \ldots,\ x_n)\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{D} \; \right\}$$

Then, using Θ, we can define the function `session` which returns the session type for a single function name f_n:

$$\texttt{session}(f_n) = \Theta(f_n)$$ ■

Definition A.7 (*Variable Substitution*)**.**

$$e\left[{}^{v}/_{x}\right] = \begin{cases} v & e = x \\ y & e = y,\ y \neq x \\ b & e = b \\ e_1\left[{}^{v}/_{x}\right] \diamond e_2\left[{}^{v}/_{x}\right] & e = e_1 \diamond e_2 \\ \texttt{not}\ \left(e'\left[{}^{v}/_{x}\right]\right) & e = \texttt{not}\ e' \\ \left[\, e_1\left[{}^{v}/_{x}\right] \mid e_2\left[{}^{v}/_{x}\right] \,\right] & e = \left[\, e_1 \mid e_2 \,\right] \\ \left\{e_1\left[{}^{v}/_{x}\right],\ \ldots,\ e_n\left[{}^{v}/_{x}\right]\right\} & e = \left\{e_1,\ \ldots,\ e_n\right\} \end{cases}$$

$$t\left[{}^{v}/_{x}\right] = \begin{cases} \texttt{send}\left(w\left[{}^{v}/_{x}\right], \left\{\texttt{:l},\ e_1\left[{}^{v}/_{x}\right],\ \ldots,\ e_n\left[{}^{v}/_{x}\right]\right\}\right) & t = \texttt{send}\left(w, \left\{\texttt{:l},\ e_1,\ \ldots,\ e_n\right\}\right) \\ \texttt{receive do}\ \left(\left\{\texttt{l}_i, \widetilde{p_i}\right\} \to t_i\left[{}^{v}/_{x}\right]\right)_{i \in I}\texttt{end} & t = \texttt{receive do}\ \left(\left\{\texttt{l}_i, \widetilde{p_i}\right\} \to t_i\right)_{i \in I}\texttt{end} \\ f\left(e_1\left[{}^{v}/_{x}\right],\ \ldots,\ e_n\left[{}^{v}/_{x}\right]\right) & t = f\left(e_1,\ \ldots,\ e_n\right) \\ \texttt{case}\ e\left[{}^{v}/_{x}\right]\ \texttt{do}\ \left(p_i \to t_i\left[{}^{v}/_{x}\right]\right)_{i \in I}\texttt{end} & t = \texttt{case}\ e\ \texttt{do}\ \left(p_i \to t_i\right)_{i \in I}\texttt{end} \\ y = t_1\left[{}^{v}/_{x}\right];\ t_2\left[{}^{v}/_{x}\right] & t = \left(y = t_1;\ t_2\right),\ x \neq y,\ y \neq v \end{cases}$$

■

B. Proofs

The following are the proofs for Proposition 6, Corollary 16, Lemmata 9, 10 and 15 from Chapter 4.

Proposition 6 (Closed Term)**.** *If* $\boldsymbol{fv}(t) = \emptyset$ *and* $t \xrightarrow{\alpha} t'$*, then* $\boldsymbol{fv}(t') = \emptyset$

Proof. By induction on the structure of t.

$[\boldsymbol{t = e}]$ Holds immediately by the rule [rExpression] and the Closed Expression Lemma.

$[\boldsymbol{t = (x = t_1;\ t_2)}]$ Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [$\textbf{rLet}_1$] From the rule, $t' = (x = t_1';\ t_2)$ and

$$t_1 \xrightarrow{\alpha} t_1' \tag{22a}$$

From the premise, $\mathbf{fv}(t) = \emptyset$, so by the **fv** definition, $\mathbf{fv}(t_1) \cup (\mathbf{fv}(t_2) \setminus \{x\}) = \emptyset$, or equivalently

$$\mathbf{fv}(t_1) = \emptyset \tag{22b}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{22c}$$

If we apply the inductive hypothesis to eqs. (22a) and (22b), we get

$$\mathbf{fv}(t_1') = \emptyset \tag{22d}$$

So, by eqs. (22c) and (22d) and the definition of $\mathbf{fv}$, we get $\mathbf{fv}(x = t_1';\ t_2) = \emptyset$ as required.

2. [$\textsc{rLet}_2$] From the rule, $t = (x = v;\ t_2)$ and $t' = t_2\,[^v/_x]$. Since $\mathbf{fv}(t) = \emptyset$, by the *Free Variables* Definition, $\mathbf{fv}(v) \cup (\mathbf{fv}(t_2) \setminus \{x\}) = \emptyset$, or equivalently

$$\mathbf{fv}(v) = \emptyset \tag{22e}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{22f}$$

We need to show that $\mathbf{fv}(t') = \emptyset$, or $\mathbf{fv}(t_2\,[^v/_x]) = \emptyset$, so we consider two sub-cases:

a. If $x \notin \mathbf{fv}(t_2)$, then by Corollary 2, $t_2 = t_2\,[^v/_x]$. Substituting this in eq. (22f), results in $\mathbf{fv}(t_2\,[^v/_x]) = \emptyset$, as required.

b. If $x \in \mathbf{fv}(t_2)$, then by Lemma 3, we get $\mathbf{fv}(t_2\,[^v/_x]) = \mathbf{fv}(t_2) \setminus \{x\}$. If we substitute this in eq. (22f), the case holds.

[$t = \mathtt{send}\,(w, \{\mathtt{:l}, e_1, \ \ldots, \ e_n\})$] Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [$\textsc{rChoice}_1$] From this rule, we know that $\alpha = \tau$ and

$$t' = \mathtt{send}\,(\iota, \{\mathtt{:l}, v_1, \ \ldots, \ v_{k-1}, \ e_k', \ \ldots, \ e_n\})$$

$$e_k \to e_k' \tag{23a}$$

Since $\mathbf{fv}(t) = \emptyset$, then by the $\mathbf{fv}$ definition

$$\mathbf{fv}(\iota) = \emptyset \tag{23b}$$

$$\mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..k-1 \tag{23c}$$

$$\mathbf{fv}(e_i) = \emptyset \text{ for } i \in k-1..n \tag{23d}$$

Applying the Closed Expression Lemma to eqs. (23a) and (23d), results in $\mathbf{fv}(e_k) = \emptyset$. Using this information along with eqs. (23b–d) and the $\mathbf{fv}$ definition, results in $\mathbf{fv}(t') = \emptyset$ as required.

2. [$\textsc{rChoice}_2$] In this case $t = \{\mathtt{:l}, v_1, \ \ldots, \ v_n\}$ and $t' = \{\mathtt{:l}_\mu, v_1, \ \ldots, \ v_n\}$.

Since from the premise $\mathbf{fv}(t) = \emptyset$, then using the **fv** definition,

$$\mathbf{fv}(\iota) = \emptyset, \quad \mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..n \tag{23e}$$

To show that $\mathbf{fv}(\{\texttt{:l}_\mu, v_1, \ldots, v_n\}) = \emptyset$, we can apply eq. (23e) to the **fv** definition.

[$t = \texttt{receive do } (\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i \in I} \texttt{ end}$] From the premise, we know that $\mathbf{fv}(t) = \emptyset$, so by the **fv** definition,

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i}) = \emptyset \quad \text{for all } i \in I \tag{24a}$$

Given that current structure of t, we can deduce $t \xrightarrow{\alpha} t'$ using [rBranch], where $\alpha = ?\{\texttt{:l}_j, v_1, \ldots, v_n\}$ for some $j \in I$, and

$$\mathbf{match}(\widetilde{p_j}, \widetilde{v}) = \sigma \text{ where } \sigma = [v'_1, \ldots, v'_k/x_1, \ldots, x_k] \tag{24b}$$

$$t' = t_j\sigma$$

From eq. (24b), we can apply Lemma 4 to get

$$\mathbf{vars}(\widetilde{p_j}) = \{x_1, \ldots, x_k\} \tag{24c}$$

Substituting eq. (24c) in eq. (24a) (for $i = j$), we get $\mathbf{fv}(t_j) \setminus \{x_1, \ldots, x_k\} = \emptyset$. Our aim is to get $t_j\sigma = \emptyset$, so we check if $x \in \mathbf{fv}(t_j)$. If this is valid, then by Lemma 3, we can conclude that $\mathbf{fv}(t_j\,[v'_1/x_1]) \setminus \{x_2, \ldots, x_k\} = \emptyset$. In case when $x \notin \mathbf{fv}(t_j)$, the same can be concluded by Corollary 2. Applying the same procedure for a total of k times, results in $\mathbf{fv}(t_j\,[v'_1, \ldots, v'_k/x_1, \ldots, x_k]) = \emptyset$, as required.

[$t = f\,(w, e_2, , \ldots, e_n)$] Given the current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [**rCall$_1$**] From this rule, we know that $\alpha = \tau$, $t = f\,(v_1, \ldots, v_{k-1}, e_k, \ldots, e_n)$, $t' = f\,(v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n)$ and

   $$e_k \to e'_k \tag{25a}$$

   Since $\mathbf{fv}(t) = \emptyset$, then by the **fv** definition,

   $$\mathbf{fv}(e_i) = \emptyset \quad \text{for all } i \in 1..n \tag{25b}$$

Applying the Closed Expression Lemma to eqs. (25a) and (25b) (for $i = k$), we get

$$\mathbf{fv}(e_k) = \emptyset \tag{25c}$$

So, using the **fv** definition with eqs. (25b) and (25c), result $\mathbf{fv}(t') = \emptyset$ holds as expected.

2. [**rCall$_2$**] From the rule, we know that $\alpha = f_n$ and

$$t = f\,(\iota, v_2,\ \ldots,\ v_n) \tag{25d}$$

$$t' = \bar{t}\,[\iota/y]\,[v_2,\ \ldots,\ v_n/x_2,\ \ldots,\ x_n]$$

$$\Sigma\,(f_n) = \Omega \qquad \Omega.\texttt{body} = t \qquad \Omega.\texttt{params} = x_2,\ \ldots,\ x_n \qquad \Omega.\texttt{dual} = y \tag{25e}$$

Since term reduction can only happen with respect to a well-formed *function information* environment Σ (*i.e.,* Definition 3.4), we can assume that the only free variables in a function body are the parameter types, or formally, for all $f_n \in dom(\Sigma)$, we have

$$\mathbf{fv}(\Sigma(f_n).\texttt{body}) \setminus \Sigma(f_n).\texttt{params} \setminus \Sigma(f_n).\texttt{dual} = \emptyset$$

Thus, using this information and substituting the information from eq. (25e), we get

$$\mathbf{fv}(\bar{t}) \setminus \{y, x_2,\ \ldots,\ x_n\} = \emptyset \tag{25f}$$

To obtain the expected result (*i.e.,* $\mathbf{fv}(t') = \emptyset$), we check if $y \in \mathbf{fv}(\bar{t})$. If this is true, then by Lemma 3, we can conclude that $\mathbf{fv}(\bar{t}\,[\iota/y]) \setminus \{x_2,\ \ldots,\ x_n\} = \emptyset$. In case when $x \notin \mathbf{fv}(\bar{t})$, the same can be concluded by Corollary 2. Applying the same procedure for the remaining *free* variables (*i.e.,* $x_2,\ \ldots,\ x_n$), we get $\mathbf{fv}(t_j\,[v'_1,\ \ldots,\ v'_k/x_1,\ \ldots,\ x_k]) = \emptyset$, as expected.

[$t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I}\ \texttt{end}$] Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [**rCase$_1$**] From the rule we know that $t' = \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i \in I}\ \texttt{end}$, and

from the premise we know that

$$e \to e' \tag{26a}$$

Since $\mathbf{fv}(t) = \emptyset$, by the **fv** definition, we know that

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i) = \emptyset \quad \text{for all } i \in I \tag{26b}$$

$$\mathbf{fv}(e) = \emptyset \tag{26c}$$

Applying Closed Expression Lemma to eqs. (26a) and (26c), results in $\mathbf{fv}(e') = \emptyset$. Thus, using this information, along with eq. (26b) and the **fv** definition, we get $\mathbf{fv}(t') = \emptyset$ as needed.

2. [$\textsc{rCase}_2$] From the rule, we know that $t = \texttt{case}\ v'\ \texttt{do}\ (p_i \to t_i)_{i \in I}\texttt{end}$, $e = v'$ and for some $j \in I$,

$$\mathbf{match}(p_j, v') = \sigma \text{ where } \sigma = [{}^{v_1, \ldots, v_n}/_{x_1, \ldots, x_n}] \tag{26d}$$

$$t' = t_j\sigma \tag{26e}$$

From the premise, we know that $\mathbf{fv}(t) = \emptyset$, so by the **fv** definition, $\mathbf{fv}(v') = \emptyset$ and

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i}) = \emptyset \quad \text{for all } i \in I \tag{26f}$$

From eq. (26d), we can apply Lemma 4, to get

$$\mathbf{vars}(p_j) = \{x_1, \ldots, x_k\} \tag{26g}$$

Substituting eq. (26g) in eq. (26f) (for $i = j$), we get $\mathbf{fv}(t_j) \setminus \{x_1, \ldots, x_k\} = \emptyset$. By similar reasoning from previous cases, we get $\mathbf{fv}(t') = \emptyset$, as required. □

Lemma 9 (Δ-Weakening)**.** *If* $\Delta \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$*, then* $(\Delta, \Delta') \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$

Proof. Follows by induction on the derivation of $\Delta \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$. We analyse the significant cases:

[tRecUnknownCall] From the rule, we know that

$$(\Delta, f_n : S) \cdot (\Gamma, \Gamma') \vdash^{y} S \rhd \bar{t} : T \lhd S' \tag{27a}$$

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{27b}$$

Applying the inductive hypothesis to eq. (27a) results in $(\Delta, \Delta', f_n : S) \cdot (\Gamma, \Gamma') \vdash^{y} S \rhd t : T \lhd S'$, where we assume that $f_n \notin dom(\Delta')$. So, using the latter result, eq. (27b) and [tRecUnknownCall] results in $(\Delta, \Delta') \cdot \Gamma \vdash^{w} S \rhd t : T \lhd S'$, as required.

[tRecKnownCall] From the rule, we know that

$$\Delta\,(f_n) = S \tag{28a}$$

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{28b}$$

If we extend Δ by Δ', then $(\Delta, \Delta')(f_n) = S$ remains valid. So, using this information, along with eq. (28b) in [tRecKnownCall], we get $(\Delta, \Delta') \cdot \Gamma \vdash^{w} S \rhd t : T \lhd \mathsf{end}$, as required.

Cases [tChoice] and [tExpression] hold immediately since Δ is unused. The remaining cases hold effortlessly by the inductive hypothesis. □

Lemma 10. *Given some pattern p, such that $\vdash^{w}_{pat} p : T \rhd \Gamma$, then $\boldsymbol{vars}(p) = dom(\Gamma)$*

Proof. Follows by induction on $\vdash^{w}_{\text{pat}} p : T \rhd \Gamma$.

[tpLiteral] From the rule (*i.e.*, $\vdash^{w}_{\text{pat}} b : T \rhd \emptyset$), $p = b$ and $\Gamma = \emptyset$, so the domain of $\Gamma = \emptyset$. By the *Variable Patterns* Definition, **vars**(b) also returns $\emptyset$, so case holds.

[tpEList] Analogous to previous case.

[tpVariable] From the rule, $p = x$ and $\Gamma = x : T$, so the domain of Γ contains just the variable x. By the **vars** definition, **vars**(x) also contains just the variable x, so the case is valid.

[TPTUPLE] From the rule, $p = \{w_1, \ldots, w_n\}$, $\Gamma = \Gamma_1, \ldots, \Gamma_n$, and

$$\vdash^{w}_{\text{pat}} w_i : T_i \rhd \Gamma_i \quad \text{for all } i \in 1..n \tag{29a}$$

By eq. (29a) and the inductive hypothesis, we know that

$$\mathbf{vars}(w_i) = dom(\Gamma_i) \quad \text{for all } i \in 1..n \tag{29b}$$

By the **vars** definition, we know that $\mathbf{vars}(p) = \mathbf{vars}(w_1) \cup \cdots \cup \mathbf{vars}(w_n)$. Thus, substituting eq. (29b) in this information, results in $\mathbf{vars}(p) = dom(\Gamma_1) \cup \cdots \cup dom(\Gamma_n)$, which is equal to $dom(\Gamma)$, as required.

[TPLIST] From the rule, $p = [\, w_1 \mid w_2 \,]$, $\Gamma = (\Gamma_1, \Gamma_2)$, and

$$\vdash^{w}_{\text{pat}} w_1 : T \rhd \Gamma_1 \tag{30a}$$

$$\vdash^{w}_{\text{pat}} w_2 : [T] \rhd \Gamma_2 \tag{30b}$$

By eqs. (30a) and (30b) and the inductive hypothesis, we know that

$$\mathbf{vars}(w_1) = dom(\Gamma_1) \quad \text{and} \quad \mathbf{vars}(w_2) = dom(\Gamma_2) \tag{30c}$$

By the **vars** definition, we know that $\mathbf{vars}(p) = \mathbf{vars}(w_1) \cup \mathbf{vars}(w_2)$. Thus, substituting eq. (30c) in this information, results in $\mathbf{vars}(p) = dom(\Gamma_1) \cup dom(\Gamma_2)$, which is equal to $dom(\Gamma)$, as required. □

Lemma 15. *For all patterns p and values v,*

$$\left.\begin{array}{r} \boldsymbol{match}(p, v) = [v_1, \ldots, v_n/x_1, \ldots, x_n] \\ \vdash^{w}_{pat} p : T \rhd \Gamma \\ \emptyset \vdash_{exp} v : T \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \Gamma = x_1 : T_1, \ldots, x_n : T_n \\ \emptyset \vdash_{exp} v_i : T_i \;\; \textit{for } i \in 1..n \end{array}\right.$$

Proof. By induction on the definition $\mathbf{match}(p, v)$. We proceed by case analysis:

[$p = b, v = b$] By the definition, $\mathbf{match}(b, b) = [\,]$, so no substitutions are expected. By $\vdash^{w}_{\text{pat}} b : T \rhd \Gamma$ and [TPLITERAL], the *variable binding* environment (*i.e.,* Γ) must be empty, so case holds immediately.

[$p = x$] By definition, $\mathbf{match}(x, v) = [v/x]$, and from the premise we know that

$$\emptyset \vdash_{\text{exp}} v : T. \tag{31a}$$

From $\vdash^{w}_{\text{pat}} x : T \rhd \Gamma$ and [tpVariable], we know that Γ must contain $x : T$ only. Therefore, case holds by eq. (31a).

$[p = [\, w_1 \mid w_2 \,], v = [\, v_1 \mid v_2 \,]]$

Using the **match** definition, $\mathbf{match}([\, w_1 \mid w_2 \,], [\, v_1 \mid v_2 \,]) =$ $\mathbf{match}(w_1, v_1), \mathbf{match}(w_2, v_2)$, or equivalently

$$\mathbf{match}(w_1, v_1) = [v'_1, \ldots, v'_j / x_1, \ldots, x_j] \tag{32a}$$

$$\mathbf{match}(w_2, v_2) = [v'_k, \ldots, v'_n / x_k, \ldots, x_n] \text{ where } k = j + 1 \tag{32b}$$

From the premise, applying [tList] to $\emptyset \vdash_{\text{exp}} [\, v_1 \mid v_2 \,] : [\, T \,]$, results in

$$\emptyset \vdash_{\text{exp}} v_1 : T \text{ and } \emptyset \vdash_{\text{exp}} v_2 : [T] \tag{32c}$$

Applying also [tpList] to $\vdash^{w}_{\text{pat}} [\, w_1 \mid w_2 \,] : [T] \rhd \Gamma$, results in

$$\vdash^{w}_{\text{pat}} w_1 : T \rhd \Gamma' \text{ and } \vdash^{w}_{\text{pat}} w_2 : [T] \rhd \Gamma'' \tag{32d}$$

Applying the inductive hypothesis twice to eqs. (32a–d) results in

$$\Gamma' = x_1 : T_1, \ldots, x_j : T_j \text{ and } \Gamma'' = x_k : T_k, \ldots, x_n : T_n \tag{32e}$$

$$\emptyset \vdash_{\text{exp}} v'_i : T_i \text{ for all } i \in 1..n \tag{32f}$$

Therefore, case holds by eqs. (32e) and (32f), since $\Gamma = \Gamma', \Gamma''$.

$[p = \{w_1, \ldots, w_m\}, v = \{v_1, \ldots, v_m\}]$

Using the **match** definition, $\mathbf{match}(\{w_1, \ldots, w_m\}, \{v_1, \ldots, v_m\}) =$ $\mathbf{match}(w_1, v_1), \ldots, \mathbf{match}(w_m, v_m) = \sigma$, or equivalently, for $i \in 1..m$,

$$\mathbf{match}(w_i, v_i) = \sigma_i \text{ given that } \sigma = \sigma_1, \ldots, \sigma_m \tag{33a}$$

From $\emptyset \vdash_{\text{exp}} \{v_1, \ldots, v_2\} : \{T_1, \ldots, T_m\}$, by [tTuple], we know that

$$\emptyset \vdash_{\text{exp}} v_i : T_i \tag{33b}$$

Applying also [tpTuple] to $\vdash^{w}_{\text{pat}} \{w_1, \ldots, w_m\} : \{T_1, \ldots, T_m\} \rhd \Gamma_1, \ldots, \Gamma_m$, results in

$$\vdash^{w}_{\text{pat}} w_i : T_i \rhd \Gamma_i \tag{33c}$$

Applying the inductive hypothesis m times to eqs. (33a–c) results in

$$\Gamma = \Gamma_1,\ \ldots,\ \Gamma_m = x_1 : T_1,\ \ldots,\ x_n : T_n$$

$$\emptyset \vdash_{\text{exp}} v_j : T_j \text{ for all } j \in 1..n$$

as required. □

Corollary 16. *For all patterns* $\widetilde{p} = p^1,\ \ldots,\ p^n$*, values* $\widetilde{v} = v_1,\ \ldots,\ v_n$ *and* $\forall j \in 1..n$*, then the following implication holds.*

$$\left.\begin{aligned} \textbf{\textit{match}}(\widetilde{p}, \widetilde{v}) &= [v'_1,\ \ldots,\ v'_k/x_1,\ \ldots,\ x_k] \\ \vdash^{y}_{pat} p^j : T^j &\ \triangleright\ \Gamma^j \\ \emptyset \vdash_{exp} v_j &: T^j \end{aligned}\right\} \implies \left\{\begin{aligned} &\widetilde{\Gamma} = \Gamma^1,\ \ldots,\ \Gamma^j = x_1 : T_1,\ \ldots,\ x_k : T_k \\ &\emptyset \vdash_{exp} v'_i : T_i \ \ \textit{for } i \in 1..k \end{aligned}\right.$$

Proof. Take $j = 1$, where we know that $\textbf{match}(p^1, v_1) = \sigma_1, \vdash^{y}_{\text{pat}} p^1 : T^1 \ \triangleright\ \Gamma^1$ and $\emptyset \vdash_{\text{exp}} v_1 : T^1$. Then, applying this information to Lemma 15, we get

$$\Gamma^1 = x^1_1 : T^1_1,\ \ldots,\ x^1_m : T^1_m \tag{34a}$$

$$\emptyset \vdash_{\text{exp}} v^1_i : T^1_i \text{ for } i \in 1..m \tag{34b}$$

Generalising for $j \in 1..n$, then $\widetilde{\Gamma} = \Gamma^1,\ \ldots,\ \Gamma^n$ holds by generalising eq. (34a). Also, $\emptyset \vdash_{\text{exp}} v'_i : T_i$ for $i \in 1..k$ holds by eq. (34b). Thus, Corollary 16 holds by applying Lemma 15 n times. □

C. Complete Example

In Section 5.5 we presented an example with only one side of an interaction verified. In this chapter, we present a simple example where both sides of an interaction are verified statically by ElixirST using session types.

Consider a simple *counter system* [74] which allows a server process to keep a running total. This total can be incremented by an interactive client process, or else terminated by the same client process.

The interaction between the client and a server is depicted in the `Counter` module shown in Listing C.1, which is made up of two public functions, called `server` and `client`, and a private function called `terminate`. The `server` function (lines 6–11) takes two parameters: the `client`'s *pid* and an initial total. This function is able to receive a message labelled, either `:incr` or `:stop`. If it receives `:incr`, along with some payload called `val`, it recurses back to the beginning, while incrementing the running total. If it receives a `:stop` label, it calls the private function `terminate`. The function `terminate` (lines 14–17) sends the total value back to the client, in a message labelled `:value`.

On the other interacting end, there is the `client` function, defined in lines 21–28, which takes the server's *pid* as a parameter. The client sends a request to the server to increase the total by five (line 22) and then by six (line 23), in messages labelled `:incr`. Afterwords, the client sends a request to `:stop` the interaction, before it receives back the final total counter (lines 25–27). The interaction can be

```
1  defmodule Counter do
2    use ElixirST
3    @session "counter = &{?incr(number).counter,
4                          ?stop().!value(number).end}"
5    @spec server(pid, number) :: atom
6    def server(client, total) do
7      receive do
8        {:incr, val} -> server(client, total + val)
9        {:stop} -> terminate(client, total)
10     end
11   end
12
13   @spec terminate(pid, number) :: atom
14   defp terminate(client, total) do
15       send(client, {:value, total})
16       :ok
17   end
18
19   @dual  "counter"
20   @spec client(pid) :: number
21   def client(server) do
22     send(server, {:incr, 5})
23     send(server, {:incr, 6})
24     send(server, {:stop})
25     receive do
26       {:value, val} -> val
27     end
28   end
29 end
```

server

client

Listing C.1: Counter annotated with *session types*

initiated using ElixirST's `spawn` function, as follows.

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

The two parties follow the protocol defined in Figure C.1. This protocol can be formalised as a session type called *counter*, which defines the interaction from the server's point-of-view:

$$counter = \& \left\{ \begin{array}{l} ?\texttt{incr}(\mathsf{number}).counter, \\ \\ ?\texttt{stop}().!\texttt{value}(\mathsf{number}).\mathsf{end} \end{array} \right\}$$

This *counter* session type dictates that the server must accept two forms of messages. It must be able to handle a sequence of messages labelled `incr` containing a payload of type number. Finally, it must also be able to handle a message `stop`,

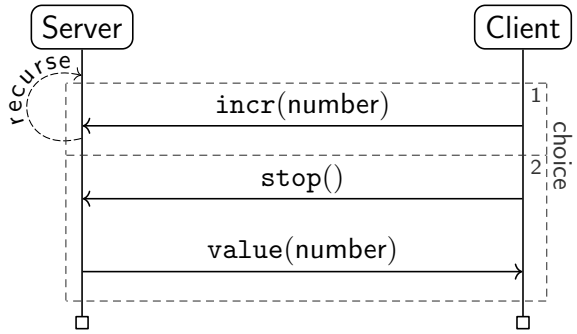


Figure C.1: Counter protocol [29]

where it must send a label `value` and a payload of type number (!`value`(number)), before terminating the session (end).

The other interacting party (*i.e.,* the client) must follow the dual session type of *counter*, called $\overline{counter}$:

$$\overline{counter} = \oplus \left\{ \begin{array}{l} !\texttt{incr}(\text{number}).\overline{counter}, \\ \\ !\texttt{stop}().?\texttt{value}(\text{number}).\text{end} \end{array} \right\}$$

The $\overline{counter}$ type dictates that the client can choose to send a message labelled `incr` with some number, where it recurses back to the beginning ($\overline{counter}$). It can also choose to send a `stop` message, where it receives the total value (?`value`(number)). Then, the interaction terminates (end).

In Listing C.1, the session type *counter* is being enforced via the `@session` annotation in line 3. Conversely, the session type $\overline{counter}$ is enforced using `@dual` in line 19.