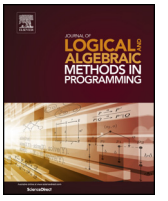
Journal of Logical and Algebraic Methods in Programming 135 (2023) 100891

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



ElixirST: A session-based type system for Elixir modules ☆

Adrian Francalanza, Gerard Tabone *

Dept. of Computer Science, University of Malta, Msida, Malta

ARTICLE INFO

Article history:
Received 5 January 2023
Received in revised form 28 April 2023
Accepted 16 June 2023
Available online 4 July 2023

Keywords:
Session types
Type systems
Elixir
Functional programming

ABSTRACT

This paper investigates the adaptation of session types to provide behavioural information about Elixir modules. We devise a type system, called ElixirST, which statically determines whether functions in an Elixir module observe their endpoint specifications, expressed as session types; a corresponding tool automating this typechecking has also been constructed. In this paper we also formally validate this type system. An LTS-based operational semantics for the language fragment supported by the type system is developed, modelling its runtime behaviour when interacting with the module client. This operational semantics is then used to prove a form of session fidelity and progress for ElixirST.

1. Introduction

Modern programming languages offer a variety of abstractions for the construction of concurrent programs. In the case of message-passing functional programs, concurrency manifests itself as spawned computation that exhibits *communication as a side-effect*, potentially influencing the execution of other (concurrent) computation. Such side-effects inevitably increase the complexity of the programs produced and lead to new sources of errors. As a consequence, correctness becomes harder to verify and language support for detecting errors statically, can substantially decrease the number of concurrency errors.

Elixir [1] is one such example of a functional programming language which supports concurrency based on the actor model [2,3]. As depicted in Fig. 1, Elixir programs are structured as a collection of *modules* that contain *functions*, the basic unit of code decomposition in the language. A module only exposes a subset of these functions to external invocations by defining them as *public*; these functions act as the only entry points to the functionality encapsulated by a module. Internally, the bodies of these public functions may then invoke other functions, which can either be the *public* ones already exposed or the *private* functions that can only be invoked from within the same module. For instance, Fig. 1 depicts a module m which contains several public functions (*i.e.*, $f_1, \ldots, f_n$) and private functions (*i.e.*, $g_1, \ldots, g_j$). The public function f_1 delegates part of its computation by calling the private functions g_1 and g_j, whereas the body of the public function f_n invokes the other public function f_1 when executed. Internally, the body of the private function g_1 calls the other private function g_2, which in turn can call g_1 again, whereas the private function g_j can recursively call itself.

☆ This work has been supported by the MoVeMnt project (No: 217987) of the Icelandic Research Fund and the BehAPI project funded by the EU H2020 RISE of the Marie Skłodowska-Curie action (No: 778233). This work is also supported by the Security Behavioural APIs project (No: I22LU01-01) funded by the UM Research Excellence Funds 2021 and the Tertiary Education Scholarships Scheme (Malta).

* Corresponding author.
E-mail addresses: adrian.francalanza@um.edu.mt (A. Francalanza), gerard.tabone@um.edu.mt (G. Tabone).

https://doi.org/10.1016/j.jlamp.2023.100891
2352-2208/

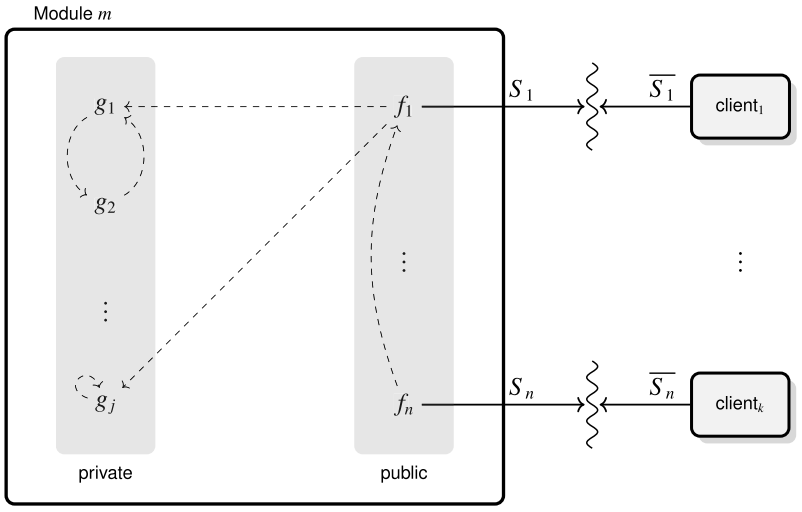


Fig. 1. An Elixir module consisting of public and private functions, interacting with client processes.

A prevalent Elixir design pattern is that of a server listening for client requests, which we refer to as the *service handler design pattern*.[1] For each request, the server spawns a (public) function to execute independently and act as a dedicated client handler: after the respective process IDs of the client and the spawned handler are made known to each other, a session of interaction commences between the two concurrent entities (via message-passing). For instance, in Fig. 1, a handler process running public function f_1 is assigned to the session with client client_1 whereas the request from client_k is assigned a dedicated handler running function f_n. Although traditional interface elements such as function parameters (used to instantiate the executing function body with values such as the client process ID) and the function return value (reporting the eventual outcome of handled request) are important, the messages exchanged between the two concurrent parties within a session are equally or more important for software correctness. More specifically, communication incompatibilities between the interacting parties could lead to various runtime errors. For example, if in a session a message is sent with an unexpected payload, it could cause the receiver's subsequent computation depending on it to crash (*e.g.* multiplying by a string when a number should have been received instead). Also, if messages are exchanged in an incorrect order, they may cause *deadlocks*, *e.g.* Elixir uses blocking receive constructs that wait until a suitable message arrives, so if a process sends an invalid message, then the other process ends up waiting forever for a proper message to arrive.

In many cases, the expected protocol of interactions within a session can be statically determined from the respective endpoint implementations, namely the function bodies; for simplicity, our discussion assumes that endpoint interaction protocols are dual, *e.g.* S_1 and $\overline{S_1}$ in Fig. 1. Although Elixir provides mechanisms for specifying (and checking) the parameters and return values of a function within a module, it does not provide any further static guarantees for programs that adhere to this design pattern due to two major obstacles. Firstly, Elixir does *not* provide any support for describing (and verifying) the interaction protocol of a function in terms of its communication side-effects. Secondly, in open settings, it is often the case that only one side of the code is available, so it is difficult to obtain static guarantees without the full codebase.

Contribution We present a type checker to assist Elixir module construction (following the service handler pattern) in two ways: (a) it allows module designers to formalise the session endpoint protocol as a session type, and ascribe it to a public function; (b) it statically verifies whether the body of a function respects the ascribed session type protocol specification. The type-checker analyses one side of an interaction, *i.e.,* the module side, without requiring access to the code invoking the module public functions. This analysis assumes that the invoking code is well-behaved (even though it may or may not have been verified against the session type protocol specification), *e.g.* client_1 follows $\overline{S_1}$ throughout the whole session in Fig. 1. The code for the type-checker, called ElixirST, is available at:

https://github.com/gertab/ElixirST

In this paper we present the underlying type system (Section 3) on which the ElixirST type-checker is built and discuss its implementation details (Section 6). We also validate the type system; more concretely, in Section 4 we formalise the runtime semantics of the Elixir language fragment supported by ElixirST as a labelled transition system (LTS), modelling the execution of a spawned handler interacting with a client within a session (left implicit). This operational semantics then

[1] Some projects which build on this *service handler design pattern* include *etorrent* [4] and *cowboy* [5]. The latter is an HTTP server which spawns a handler process for each new request; this is scalable since spawning (and maintaining) actors is extremely efficient in Elixir. In literature, this pattern is also called *thread-per-session* [6].

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

Listing 1: Counter written in Elixir.

allows us to prove a conditional form of the *session fidelity* and *progress* properties for the ElixirST type system (Section 5). Intuitively they state that, if the interacting processes (that are left implicit) follow the prescribed protocol correctly, then the module code being typechecked is guaranteed to behave correctly w.r.t. the protocol (expressed in terms of the adapted session fidelity and progress properties).

In this paper we merge and extend the work presented in [7,8]. In addition to [8], we include the full typing rules which were originally relegated to the appendix. We also present two additional sections (adapted from [7]): (a) Section 2 introduces a motivating example; and (b) Section 6 describes the implementation details of the ElixirST tool, along with a case study. The validation part of the paper is also expanded, by including all proofs. We also add a new result, showing that ElixirST observes a form of the *progress* property.

2. Motivating example

Consider a simple *counter* system, adapted from [9], whereby a (server) process stores a counter total which can be increased by a (client) interacting process or else terminated by this same (client) process. A sample Elixir `Counter` module is shown in Listing 1. It offers one public function called `server` on lines 3–8 taking two arguments: the *pid* of the client, `client`, and the initial counter total, `total`. A process executing this function waits to receive client requests as messages in its mailbox using the **`receive do ... end`** statement; this construct is blocking, meaning that the process stops until a message with the expected format is received. The `server` function accepts two types of messages, namely, increment requests with label `:incr` carrying payload `val`, or termination requests denoted by the label `:stop`. This function branches accordingly: for increment requests, it recurses while updating the running total to `total+value` on line 5, whereas termination requests on line 6 are handled by calling the *private* function `terminate`. Private functions, defined using **`defp`**, are only visible from *within* a module. In this case, the function `terminate` (defined on lines 11–14) sends a `:value` message carrying the final total value `total` to the `client` process and terminates with the atom value `:ok`. Assuming that a client process carrying a *pid* bound to variable `cid` already exists, a counter server linked to `cid` initialised with a running total of `0` can be launched using the statement:

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

Elixir conducts dynamic typechecking to catch runtime errors. In addition, `@spec` annotations such as those on lines 2 and 10 can help with detecting potential errors at compile-time. However, the language offers limited support to assist the static detection of errors relating to the concurrent messaging. For instance, it might not be immediately apparent that the payload carried by a `:incr` request should be a `number` value. Similarly, the code in Listing 1 does not necessarily convey enough information that the intended interaction with a `server` process should follow the protocol depicted in Fig. 2. This abstract specification states that a server can be incremented an arbitrary number of times, followed by a single termination request (*i.e.,* no further increment or termination requests can succeed it).

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

$$counter = \& \left\{ \begin{array}{l} ?\texttt{incr}(\mathsf{number}).counter, \\ ?\texttt{stop}().!\texttt{value}(\mathsf{number}).\mathsf{end} \end{array} \right\} \tag{1}$$

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

$$\overline{counter} = \oplus \left\{ \begin{array}{l} !\texttt{incr}(\mathsf{number}).\overline{counter}, \\ !\texttt{stop}().?\texttt{value}(\mathsf{number}).\mathsf{end} \end{array} \right\} \tag{2}$$

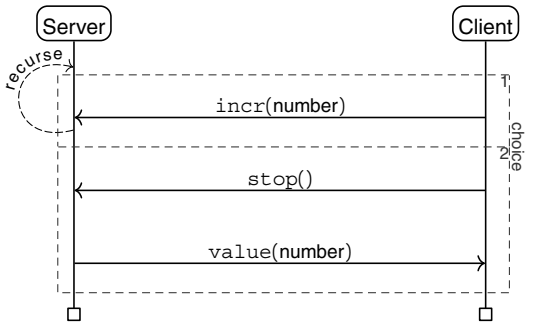


Fig. 2. Counter protocol.

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

Listing 2: Counter annotated with *session types*.

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

This paper proposes an approach whereby module definitions are augmented with a `@session` annotation for functions, as shown in Listing 2. Whereas line 4 requires the *public* function `server` to adhere to the session type *counter*, no session annotation is required for the *private* function `terminate` on line 15. Lines 22–24 present a case in which the client code is defined as a public function within the same module; in such a case, we can annotate it with the `@dual` information on line 20.

Our proposed session type annotations serve two important purposes. On the one hand, they provide a high-level (yet formal) specification as to how a public function is to be interacted with, without the need to look inside its implementation, as in the case of line 4. For instance, by inspecting the *counter* session type on line 4 of Listing 2, one can immediately tell that a process running function `server` accepts two types of messages with labels `incr` or `stop`. On the other hand, they allow function implementations to be typechecked against such specifications. *E.g.*, we are able to statically check that the function `server` (and its ancillary function `terminate`) adheres to the protocol dictated by session type *counter* on line 4. We can also reject the problematic `client` implementation given in Listing 3 at compile-time, on the grounds that it violates the dual session type of *counter*. Concretely, the client selects an illegal choice `decr` on line 23, since the server cannot handle an incoming message labelled `decr`. The client also expects to receive a value with a number (line 26) after 'forgetting' to send a termination request (*i.e.*, a message with a `stop` label). Both cases breach the $\overline{counter}$ protocol.

3. A formal analysis

We introduce a core Elixir subset and define the typing rules for the ElixirST type system.

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

Listing 3: Counter client with issues.

3.1. The actor model

Elixir uses the actor concurrency model [2,3]. It describes computation as a group of concurrent processes, called *actors*, which do *not* share any memory and interact exclusively via asynchronous messages. Every actor is identified via a unique process identifier (*pid*) which is used as the address when sending messages to a specific actor. Messages are communicated asynchronously, and stored in the mailbox of the addressee actor. An actor is the only entity that can fetch messages from its mailbox, using pattern matching. This allows us to provide a static behavioural abstraction for public functions used for the service handler design pattern. Apart from sending and reading messages, an actor can also spawn other actors and obtain their fresh *pid* as a result; this *pid* can be communicated as a value to other actors via messaging, which allows for a dynamically linked structure amongst active actors.

3.2. Session types

The ElixirST type system assumes the standard expression types, including basic types, such as boolean, number, atom and pid, and inductively defined types, such as tuples, $\{T_1, \ldots, T_n\}$, and lists, $[T]$; these already exist in the Elixir language and they are dynamically checked. The type system extends these with (binary) session types, which are used to statically check the message-passing interactions.

$$\text{Expression types} \qquad T ::= \mathsf{boolean} \mid \mathsf{number} \mid \mathsf{atom} \mid \mathsf{pid} \mid \{T_1, \ldots, T_n\} \mid [T]$$

$$\begin{array}{lll} \text{Session types} & S ::= \&\left\{?\mathtt{l}_i(\widetilde{T}_i).S_i\right\}_{i\in I} & \textit{Branch} \\ & \mid \oplus\left\{!\mathtt{l}_i(\widetilde{T}_i).S_i\right\}_{i\in I} & \textit{Choice} \\ & \mid \mathsf{rec}\,\mathsf{X}\,.\,S & \textit{Recursion} \\ & \mid \mathsf{X} & \textit{Variable} \\ & \mid \mathsf{end} & \textit{Termination} \end{array}$$

The *branching* construct, $\&\left\{?\mathtt{l}_i(\widetilde{T}_i).S_i\right\}_{i\in I}$, requires the code to be able to receive a message that is labelled by any one of the labels $\mathtt{l}_i$, with the respective list of values of type $\widetilde{T}_i$ (where $\widetilde{T}$ stands for $T^1, \ldots, T^k$ for some $k \geq 0$), and then adhere to the continuation session type S_i. The *choice* construct is its dual and describes the range and format of outputs the code is allowed to perform at that point of execution. In both cases, the labels need to be pairwise distinct. Recursive types are treated equi-recursively [10], and used interchangeably with their unfolded counterparts. For brevity, the symbols & and $\oplus$ are occasionally omitted for singleton options, *e.g.*, $\oplus\left\{!\mathtt{l}(\mathsf{number}).S_1\right\}$ is written as $!\mathtt{l}(\mathsf{number}).S_1$; similarly end may be omitted as well, *e.g.*, $?\mathtt{l}()$ stands for $?\mathtt{l}().\mathsf{end}$. The *dual* of a session type S is denoted as $\overline{S}$ (shown in Definition 3.1).

Definition 3.1 *(Duality).*

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

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

3.3. Elixir syntax

Elixir programs are organised as modules, *i.e.*, $\mathtt{defmodule}\ m\ \mathtt{do}\ \widetilde{P}\ \widetilde{D}\ \mathtt{end}$. Modules are defined by their name, m, and contain two sets of public $\widetilde{D}$ and private $\widetilde{P}$ functions, declared as sequences. Public functions, $\mathtt{def}\ f(y,\widetilde{x})\ \mathtt{do}\ t\ \mathtt{end}$, are defined by the **def** keyword, and can be called from any module. In contrast, private functions, $\mathtt{defp}\ f(y,\widetilde{x})\ \mathtt{do}\ t\ \mathtt{end}$, can only be called from within the defining module. Functions are defined by their name, f, and their body, t, and parametrised by a sequence of *distinct* variables, $y,\widetilde{x}$, the length of which, $|y,\widetilde{x}|$, is called the *arity*. The first parameter (y), is reserved for

Modules	$M ::= \texttt{defmodule}\ m\ \texttt{do}\ \widetilde{P}\ \widetilde{D}\ \texttt{end}$
Public functions	$D ::= K \quad B \quad \texttt{def}\ f(y, \widetilde{x})\ \texttt{do}\ t\ \texttt{end}$
Private functions	$P ::= B \quad \texttt{defp}\ f(y, \widetilde{x})\ \texttt{do}\ t\ \texttt{end}$
Type annotations	$B ::= \texttt{@spec}\ f(\widetilde{T})\ ::\ T$
Session annotations	$K ::= \texttt{@session}\ \text{“}\mathsf{X} = S\text{”} \mid \texttt{@dual}\ \text{“}\mathsf{X}\text{”}$
Expressions	$e ::= w \mid \texttt{not}\ e \mid e_1 \diamond e_2 \mid [\, e_1 \mid e_2 \,] \mid \{e_1, \ldots, e_n\}$
Operators	$\diamond ::= < \mid > \mid <= \mid >= \mid == \mid != \mid + \mid - \mid * \mid / \mid \texttt{and} \mid \texttt{or}$
Basic values	$b ::= \mathit{boolean} \mid \mathit{number} \mid \mathit{atom} \mid \mathit{pid} \mid [\,]$
Values	$v ::= b \mid [\, v_1 \mid v_2 \,] \mid \{v_1, \ldots, v_n\}$
Identifiers	$w ::= b \mid x$
Patterns	$p ::= w \mid [\, w_1 \mid w_2 \,] \mid \{w_1, \ldots, w_n\}$
Terms	$t ::= e$
	$\mid x = t_1;\ t_2$
	$\mid \texttt{send}(w, \{\texttt{:l}, e_1, \ldots, e_n\})$
	$\mid \texttt{receive do}\ \left(\{\texttt{:l}_i, p_i^1, \ldots, p_i^n\} \to t_i\right)_{i \in I} \texttt{end}$
	$\mid f(w, e_1, \ldots, e_n)$
	$\mid \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i \in I} \texttt{end}$

Fig. 3. Elixir syntax.

the *pid* of the dual process. Although a module may contain functions with the same name, their arity must be different, so we refer to a function with name f and arity n as f/n.

In Elixir, function parameters and return values can be assigned a type using the `@spec` annotation, $f(\widetilde{T}) :: T$, describing the parameter types, $\widetilde{T}$, and the return type, T. This information is then used by the Dialyzer to perform static checking using success typing [11]. In our formalisation, we require that functions are annotated by their type specification, denoted by B in Fig. 3. In addition to this, we decorate public functions with session types, defined in Section 3.2, to describe their side-effect protocol. Public functions can be annotated directly using `@session` “$\mathsf{X} = S$”, or indirectly using the dual session type, `@dual` “X”, where $\mathsf{X} = S$ is shorthand for $\mathsf{rec}\,\mathsf{X}.\,S$.

The body of a function consists of a term, t, which can take the form of an expression, a `let` statement, a send or receive construct, a case statement or a function call; see Fig. 3. In the case of the `let` construct, $x = t_1;\ t_2$, the variable x is a *binder* for the variables in t_2, acting as a placeholder for the value that the subterm t_1 evaluates to. We write $t_1; t_2$, as *syntactic sugar* for $x = t_1;\ t_2$ whenever x is not used in t_2. 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. The *receive* construct, $\texttt{receive do}\ \left(\{\texttt{:l}_i, p_i^1, \ldots, p_i^n\} \to t_i\right)_{i \in I} \texttt{end}$, allows a process to receive a message tagged with a label that matches one of the labels $\texttt{:l}_i$ and a list of payloads that match the patterns $p_i^1, \ldots, p_i^n$, branching to continue executing as t_i. Patterns, p, defined in Fig. 3, can take the form of a variable, a basic value, a tuple or a list (*e.g.* $[\, x \mid y \,]$, where x is the head and y is the tail of the list). The remaining constructs are fairly standard. Variables in patterns $p_i^1, \ldots, p_i^n$ employed by the `receive` and `case` statements are binders for the respective continuation branches t_i. In our formalisation, the `case` construct is assumed to have a catch-all pattern which acts as a fail-safe mechanism for unmatchable values. This provides stronger static guarantees, however, it is more conservative when compared to Elixir's common *let it crash* philosophy, which allows unmatched values to crash the whole process.

Terms can also take the form of an expression, e. An expression can be a variable, basic value (*e.g. boolean*), list, tuple, or other operations (*e.g.* $+$, $<$, *and*). Note that in the original Elixir syntax, there is no separation between terms (t) and expressions (e). However, with this distinction we are able to keep the type system in Section 3.4 more manageable and concise. We assume standard notions of open (*i.e.,* $\mathbf{fv}(t) \neq \emptyset$) and closed (*i.e.,* $\mathbf{fv}(t) = \emptyset$) terms and work up to alpha-conversion of bound variables.

3.4. Type system

Our session type system statically verifies that public functions within a module observe the communication protocols ascribed to them. It uses three environments:

$$\text{Variable binding env.} \quad \Gamma ::= \emptyset \mid \Gamma,\ x : T$$

$$\text{Session typing env.} \quad \Delta ::= \emptyset \mid \Delta,\ f/n : S$$

$$\text{Function information env.} \quad \Sigma ::= \emptyset \mid \Sigma,\ f/n : \left\{ \begin{array}{l} \texttt{params} = \widetilde{x},\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{body} = t,\ \texttt{return\_type} = T,\ \texttt{dual} = y \end{array} \right\}$$

The *variable binding* environment, Γ, maps (data) variables to basic types ($x : T$). We write $\Gamma, x : T$ to extend Γ with the new mapping $x : T$, where $x \notin \mathbf{dom}(\Gamma)$. The *session typing* environment, Δ, maps function names and arity pairs to their session type ($f/n : S$); this will be used when recursing (*i.e.,* function calls) to check whether a function is already mapped to a session type. If a function f/n has a *known* session type, then it can be found in Δ, *i.e.,* $\Delta(f/n) = S$. Each module has a static *function information* environment, Σ, that holds information related to the function definitions. 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 static variable name that represents the interacting process' *pid* (`dual`). We assume that function information environment, Σ, is *well-formed*, meaning that all functions mapped ($f/n \in \mathbf{dom}(\Sigma)$) observe the following condition requiring that the body of function f/n is *closed*, *i.e.,* for $\Sigma(f/n) = \Omega$:

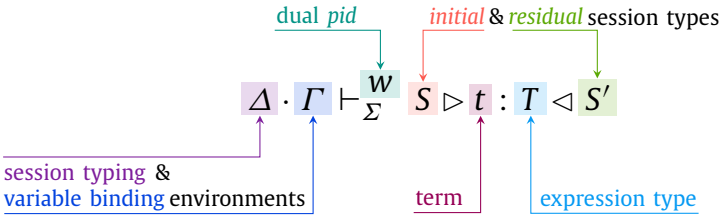
$$\mathbf{fv}\big(\Omega.\texttt{body}\big) \setminus \big(\Omega.\texttt{params} \cup \{\Omega.\texttt{dual}\}\big) = \emptyset$$

Intuitively, if we look at a function's body, its only free variables will be its parameters (*i.e.,* `params` and `dual`).

Session typechecking is initiated by analysing an Elixir module, rule [tModule]. A module is typechecked by inspecting each of its public functions, ascertaining that they correspond and fully consume the session types ascribed to them. The rule uses three helper functions. The function $\mathbf{functions}(\widetilde{D})$ returns a list of all function names (and arity) of the public functions ($\widetilde{D}$) to be checked individually. The function $\mathbf{sessions}(\widetilde{D})$ obtains a mapping of all the public functions to their expected session types stored in Δ. This ensures that when a function f with arity n executes, it adheres to the session type associated with it using either the `@session` or `@dual` annotations. The helper function $\mathbf{details}(-)$ populates the function information environment (Σ) with details about all the *public* ($\widetilde{D}$) and *private* functions ($\widetilde{P}$) within the module.

$$[\textsc{tModule}]\ \frac{\Delta = \mathbf{sessions}(\widetilde{D}) \qquad \Sigma = \mathbf{details}(\widetilde{P}\ \widetilde{D}) \qquad \forall f/n \in \mathbf{functions}(\widetilde{D}) \cdot \left\{ \begin{array}{l} \Delta(f/n) = S \qquad \Sigma\,(f/n) = \Omega \\ \Omega\,.\texttt{params} = \widetilde{x} \qquad \Omega\,.\texttt{param\_types} = \widetilde{T} \\ \Omega.\texttt{body} = t \qquad \Omega.\texttt{dual} = y \\ \Omega.\texttt{return\_type} = T \\ \Delta \cdot \big(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\big) \vdash^{y}_{\Sigma} S \rhd t : T \lhd \texttt{end} \end{array} \right.}{\vdash \texttt{defmodule}\ m\ \texttt{do}\ \widetilde{P}\ \widetilde{D}\ \texttt{end}}$$

For every public function f/n in $\mathbf{functions}(\widetilde{D})$, [tModule] checks that its body adheres to it session type using the highlighted *term typing* judgement detailed below:



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' (akin to parameterised monads [12]), 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 term typing rules. We consider each rule in detail.

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

The `receive` construct is typechecked using the [TBranch] rule. It expects an (external) branching session type $\&\{\dots\}$, where 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 ($\widetilde{T}_i$) correspond. Each `receive` branch is checked w.r.t. the common type T and a common residual session type S'. The types within each `receive` branch are computed using the pattern typing judgement, $\vdash^{w}_{\text{pat}} p : T \triangleright \Gamma$, which assigns types to variables present in patterns (explained later in Fig. 6).

To ensure that all possible patterns are exhausted, we use *Simple Patterns* to filter out pattern structures such as lists and basic values. As a result, if a `receive` construct (contains exclusively simple patterns) is well-typed against the branching session type, then we are certain that any valid incoming messages can be matched with one of the available branches. This is further discussed later in Proposition 3. Simple patterns do not limit the expressivity of our language: complex patterns can still be expressed using a combination of (simple) `receives` and inner `case` constructs, which allow us to use patterns that do not block receives while postponing more specific patterns to the subsequent case matching.

Definition 3.2 *(Simple Patterns).* The predicate **simplepat**(p, T) only accepts patterns that exclusively contain variables and top-level tuples, eliminating the possibility of patterns with structures such as lists or basic values. It is defined as follows:

$$\textbf{simplepat}(p, T) \stackrel{\text{def}}{=} \big((p = \{x_1, \dots, x_n\} \text{ and } T = \{T_1, \dots, T_n\}) \text{ or } p = x\big) \quad \blacksquare$$

Example 3.1. Consider the following `receive` construct (following the `?incr`(number)$\dots$ session type) which does not satisfy **simplepat**:

```
receive do  {:incr, 5} → ...  end
```

This can only accept messages of the form {`:incr`, 5}, so other messages that are *valid* for the `?incr`(number)$\dots$ session type, *e.g.* {`:incr`, 9}, cannot be processed, since not all valid patterns are exhausted. In contrast, **simplepat** limits us to branches such as the one below where the variable (called `value`) can accept any value:

```
receive do  {:incr, value} → ...  end
```

■

Another crucial typing rule is [TChoice], which typechecks the sending of messages.

$$[\text{TChoice}]\ \frac{\mu \in I \qquad \forall j \in 1..n \cdot \left\{\Gamma \vdash_{\text{exp}} e_j : T^j_\mu\right\}}{\Delta \cdot \Gamma \vdash^{w} \oplus\left\{!\mathtt{l}_i(\widetilde{T}_i).S_i\right\}_{i\in I} \triangleright \mathtt{send}\left(w, \left\{:\mathtt{l}_\mu, e_1, \dots, e_n\right\}\right) : \{\mathsf{atom}, \widetilde{T}_\mu\} \triangleleft S_\mu}$$

This rule requires an internal choice session type $\oplus\{\dots\}$, where the label tagging the message to be sent must match with one of the labels ($\mathtt{l}_\mu$) offered by the session choice. The message payloads must also match with the corresponding types associated with the label ($\widetilde{T}_\mu$ of $\mathtt{l}_\mu$) stated via the expression typing judgement $\Gamma \vdash_{\text{exp}} e : T$ (see Fig. 5). The resulting expression type of the `send` construct is equivalent to the type of message being sent, *i.e.,* $\left\{:\mathtt{l}_\mu, \widetilde{e_i}\right\}$ has type $\{\mathsf{atom}, \widetilde{T}_\mu\}$. The typing rule also checks the *pid* of the addressee of the `send` statement which must match with the dual *pid* (w) in the judgment itself to ensure that messages are only sent to the correct addressee.

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

Since public functions are decorated with a session type explicitly using the `@session` (or `@dual`) annotation, they are listed in **dom**(Δ). Calls to public functions are typechecked using the [TKnownCall] rule, which verifies that the expected initial session type is equivalent to the function's *known* session type (S) obtained from the *session typing* environment, *i.e.,* $\Delta(f/n) = S$. Without typechecking the function's body, which is done in rule [TModule], this rule ensures that the parameters have the correct types (using the expression typing rules). From the check performed in rule [TModule], it can also safely assume that this session type S is fully consumed, thus the residual type becomes end. Rule [TKnownCall] also ensures that the *pid* (w) is preserved during a function call, by requiring it to be passed as a parameter and comparing it to the expected dual *pid* (*i.e.,* $\Delta \cdot \Gamma \vdash^{\boxed{w}} S \triangleright f(\boxed{w}, \dots) : T \triangleleft \mathsf{end}$).

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

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

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

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

Fig. 4. Remaining term typing rules.

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

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

$$[\textsc{tBasic}]\ \frac{\textbf{type}(b) = T}{\Gamma \vdash_{\text{exp}} b : T} \qquad [\textsc{tVariable}]\ \frac{\Gamma(x) = T}{\Gamma \vdash_{\text{exp}} x : T}$$

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

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

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

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

Fig. 5. Expression typing rules.

In contrast, a call to a (private) function, f/n, with an *unknown* session type associated to it is typechecked using the [tUnknownCall] rule. As in the other rule, it ensures that parameters have the correct types ($\Gamma \vdash_{\text{exp}} e_i : T_i$). In addition, it also analyses the function's body t (obtained from Σ) with respect to the session type S inherited from the initial session type of the call, This session type is appended to the session typing environment Δ for future reference, *i.e.,* $\Delta' = (\Delta, f/n : S)$ which allows it to handle recursive calls to itself; should the function be called again, rule [tKnownCall] is used thus bypassing the need to re-analyse its body.

The remaining rules (Fig. 4) make up the functional aspect of the language. The *let* statement $x = t_1;\ t_2$ is typechecked using the rule [tLet]. The initial session type S is first transformed to S' due to some actions in t_1 and finally becomes S'' after the actions in t_2. The rule [tCase] checks the `case` construct, where each case has to match the corresponding type T and session type S. The final pattern (x_{all}) of the `case` construct acts as a catch-all alternative to ensure that all values are matched to at least one case. The catch-all case is sometimes omitted for ease of readability. Finally, [tExpression] checks all expressions e using *expression typing*. Expressions do not have a side effect, so the continuation session type S remains unchanged.

Expression typing Expressions are typechecked using the judgement $\Gamma \vdash_{\text{exp}} e : T$, where expression e has type T subject to the variable environment Γ. The expressions typing rules (Fig. 5) are adapted from [13]. Rule [tBasic] checks the type of basic values using the function `type`, which returns the type for basic values, *e.g.,* $\textbf{type}(true) = \text{boolean}$. Rule [tVariable] checks that variables have the correct type, as specified in Γ. Rules [tTuple], [tEList] and [tList] check the types of tuples, empty lists and lists, respectively. Rule [tArithmetic] ensures that numbers are used when doing arithmetic operations. The remaining rules, [tBoolean], [tNot] and [tComparisons], are analogous.

Pattern typing In the term typing rules [tBranch] and [tCase], *new* variables are introduced as a result of pattern matching. These need to be assigned to their respective type for typechecking purposes. This is obtained via the judgement $\vdash^{p}_{\text{pat}} T : \Gamma \rhd$ defined in Fig. 6, which states that all variables in a pattern p are collected (with their type) in Γ. Basic values are checked in [tpBasic], and new variables are introduced in [tpVariable]. The latter also ensures that the *pid* of the dual process remains unchanged (*i.e.,* $x \neq w$), which allows to statically locate the destination *pid* of the messages. Each element in a tuple is checked individually for either values or variables ([tpTuple]). Lists are checked using [tpEList] and [tpList]. Multiple variable environments Γ and Γ' are joined together as Γ, Γ' (their domains must be distinct).

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

$$[\text{TPBASIC}]\ \frac{\emptyset \vdash_{\text{exp}} b : T \qquad b \neq [\,]}{\vdash^{w}_{\text{pat}} b : T \rhd \emptyset} \qquad [\text{TPVARIABLE}]\ \frac{x \neq w}{\vdash^{w}_{\text{pat}} x : T \rhd x : T}$$

$$[\text{TPTUPLE}]\ \frac{\forall i \in 1..n \cdot \left\{\vdash^{w}_{\text{pat}} w_i : T_i \rhd \Gamma_i\right\}}{\vdash^{w}_{\text{pat}} \{w_1, \ldots, w_n\} : \{T_1, \ldots, T_n\} \rhd \Gamma_1, \ldots, \Gamma_n}$$

$$[\text{TPLIST}]\ \frac{\vdash^{w}_{\text{pat}} w_1 : T \rhd \Gamma_1 \qquad \vdash^{w}_{\text{pat}} w_2 : [T] \rhd \Gamma_2}{\vdash^{w}_{\text{pat}} [\,w_1 \mid w_2\,] : [T] \rhd \Gamma_1, \Gamma_2} \qquad [\text{TPELIST}]\ \frac{}{\vdash^{w}_{\text{pat}} [\,] : [T] \rhd \emptyset}$$

Fig. 6. Pattern typing rules.

3.5. Typing in action

Recall the *counter* system from Listing 2, which contains two public functions, `server` and `client`, annotated with the session types, *counter* and $\overline{\textit{counter}}$, respectively defined in eqs. (1) and (2). We discuss briefly how the type system of Section 3.4 can be used to statically analyse this Elixir module.

Typechecking starts from the [TMODULE] rule ($\vdash M$), and the judgement:

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

where $\widetilde{D}$ contains the functions `server` and `client`, while $\widetilde{P}$ contains the private function `terminate`. The premise of [TMODULE] requires that all public functions are checked individually using the behavioural typing judgement: $\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$ (Section 3.4). Starting with $f_n = \texttt{server}_2$, the initial session type, S, is set to *counter* and the expected residual session type, S', is end, since functions are only well-typed if they *fully consume* the session type. For the `client` function, the initial session type S is computed to get the dual type of *counter*, given in eq. (2). We focus on the behavioural typing of the `server` function. The function body, t, of `server` consists of the following:

$$t = \left\{\begin{array}{l}\texttt{receive do}\\ \quad \texttt{\{:incr, value\} -> server(client, total + value)}\\ \quad \texttt{\{:stop\} -> terminate(client, total)}\\ \texttt{end}\end{array}\right.$$

This `receive` statement is typechecked as the judgement below, using the [TBRANCH] rule:

$$\Delta \cdot \Gamma \vdash^{w} \& \left\{\begin{array}{l}?\texttt{incr}(\text{number}).\textit{counter},\\ ?\texttt{stop}().S_1\end{array}\right\} \rhd t : \text{atom} \lhd \text{end}$$

where $S_1 = !\texttt{value}(\text{number}).\text{end}$ and $w = \texttt{client}$.

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

$$\Delta \cdot \Gamma' \vdash^{w} \textit{counter} \rhd \texttt{server(client, total+value)} : \text{atom} \lhd \text{end}$$

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

$$[\text{TUNKNOWNCALL}]\ \frac{[\text{TLET}]\ \dfrac{\vdots}{(\Delta, \texttt{terminate}_1 : S_1) \cdot \Gamma'' \vdash^{w} S_1 \rhd \left[\begin{array}{l}\texttt{send(w, \{:value, total\})}\\ \texttt{:ok}\end{array}\right] : \text{atom} \lhd \text{end}}}{\Delta \cdot \Gamma \vdash^{w} S_1 \rhd \texttt{terminate(client, total)} : \text{atom} \lhd \text{end}}$$

Note that Γ'' contains the type information for the `client` and `total` variable names, and the session typing judgement (Δ) is extended to contain the session typing information for the function $\texttt{terminate}_1$. To finish our typing analysis, we have to consider the two premises of the [TLET] rule. This rule checks two sub-terms in succession, as follows:

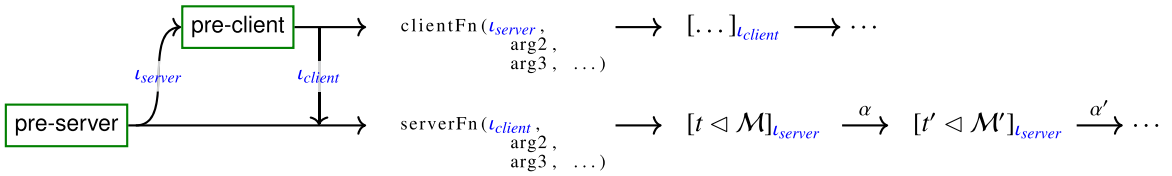


Fig. 7. Spawning two processes (green boxes represent *spawned* concurrent processes). (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

$$(\Delta, \texttt{terminate}_1 : S_1) \cdot \Gamma'' \vdash^w S_1 \rhd \texttt{send(}w\texttt{, \{:value, total\})} : U \lhd \mathsf{end} \tag{3}$$

$$(\Delta, \texttt{terminate}_1 : S_1) \cdot \Gamma'' \vdash^w S_1 \rhd \texttt{:ok} : \mathsf{atom} \lhd \mathsf{end} \tag{4}$$

Equation (3) contains a `send` construct that sends a message labelled `value`. This matches with the labels offered in S_1, which is checked using [TCHOICE]. Finally, Equation (4) is typechecked using [TEXPRESSION], which finalises our analysis.

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

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

The first `send` statement is checked successfully using the [TLET] and the [TCHOICE] rules. The next `send` statement also needs to be checked using [TCHOICE]:

$$\Delta' \cdot \Gamma''' \vdash^w \oplus \left\{ \begin{array}{l} !\texttt{incr}(\mathsf{number}).\overline{counter}, \\ !\texttt{stop}().\overline{S_1} \end{array} \right\} \rhd \texttt{send(server, \{:decr, 2\})} : \mathsf{number} \lhd \mathsf{end}$$

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

3.6. Elixir system

Natively, Elixir can create actors using its `spawn` functions (*e.g.* `spawn/3`), which take a function (and its arguments), spawns it and returns its *pid*. ElixirST extends this to provide a bespoke spawning function called `session/4` which allows the initiation of two concurrent processes executing in tandem as part of a session. This `session/4` function takes two pairs of arguments: two references of function names (that will be spawned), along with their list of arguments. Its participant creation flow is shown in Fig. 7. Initially the actor (pre-server) is spawned, passing its *pid* (ι_{server}) to the second spawned actor (pre-client). Then, pre-client relays back its *pid* (ι_{client}) to pre-server. In this way, both actors participating in a session become aware of each other's *pids*. From this point onwards, the two actors execute their respective function to behave as the participants in the binary session; the first argument of each running function is initiated to the respective *pid* of the other participant. Fig. 7 shows that the server process executes the body t, where it has access to the mailbox, which we denote as $\mathcal{M}$. As it executes, messages may be sent or received (shown by the action α) and stored in the (modified) mailbox $\mathcal{M}'$. The specific working of these transitions is explained in the following section.

4. Operational semantics

We describe the operational semantics of the Elixir language subset of Fig. 3 as a *labelled transition system* (LTS) [14] describing how a handler process within a session executes while interacting with the session client (left implicit), as outlined in Fig. 1. The transition $t \xrightarrow{\alpha} t'$ describes the fact that a handler process in state t performs an execution step to transition to the new state t', while possibly interacting with the client via the action α as a side-effect. External actions are visible by, and bear an effect on the client, whereas internal actions do not. In our case, an action α can take the following forms:

$$\begin{array}{rlr} \alpha \in \text{Act} ::= & \iota!\{\texttt{:l}, \widetilde{v}\} & \text{Output message to } \iota \text{ tagged as } \texttt{:l} \text{ with payload } \widetilde{v} \\ | & ?\{\texttt{:l}, \widetilde{v}\} & \text{Input message tagged as } \texttt{:l} \text{ with payload } \widetilde{v} \\ | & f/n & \text{Call function } f \text{ with arity } n \\ | & \tau & \text{Internal reduction step} \end{array} \quad \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \begin{array}{l} \text{external action} \\ \\ \text{internal action} \end{array}$$

Both output and input actions constitute external actions that affect either party in a session; the type system from Section 3.4 disciplines these external actions. Internal actions, include *silent* transition (τ) and function calls (f/n); although

$\boxed{t \xrightarrow[\Sigma]{\alpha} t'}$

$$[\text{RLET}_1]\ \frac{t_1 \xrightarrow{\alpha} t_1'}{x = t_1;\ t_2 \xrightarrow{\alpha} x = t_1';\ t_2} \qquad [\text{RLET}_2]\ \frac{}{x = v;\ t \xrightarrow{\tau} t\,[^{v}/_{x}]}$$

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

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

$$[\text{RBRANCH}]\ \frac{\exists j \in I \qquad \texttt{l}_j = \texttt{l} \qquad \mathbf{match}(\widetilde{p_j}, v_1, \ldots, v_n) = \sigma}{\texttt{receive do}\,(\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i\in I}\texttt{end} \xrightarrow{?\{\texttt{:l}, v_1, \ldots, v_n\}} t_j\sigma}$$

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

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

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

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

Fig. 8. Term transition semantic rules.

the latter may be denoted as a silent action, the decoration facilitates our technical development. We note that, function calls can only transition subject to a well-formed function information environment (Σ), which contains details about all the functions available in the module. Since Σ remains static during transitions, we leave it implicit in the transitions rules.

The transitions are defined by the *term* transition rules listed in Fig. 8. Rules [RLET$_1$] and [RLET$_2$] deal with the evaluation of a *let* statement, $x = t_1;\ t_2$ modelling a *call-by-value* semantic, where the first term t_1 has to transition fully to a value before being substituted for x in t_2 denoted as $[^{v}/_{x}]$ (or $[^{v_1, v_2}/_{x_1, x_2}]$ for multiple 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 from left to right. This is carried out via rule [RCHOICE$_1$] which produces no observable side-effects. When the whole message is reduced to a tuple of values $\{\texttt{:l}, v_1, \ldots, v_n\}$, rule [RCHOICE$_2$] performs the actual message sending operation. This transition produces an action $\iota!\{\texttt{:l}, v_1, \ldots, v_n\}$, where the message $\{\texttt{:l}, v_1, \ldots, v_n\}$ is sent to the interacting process with a *pid* value of ι. The operational semantics of the *receive* construct, $\texttt{receive do}\,(\{\texttt{:l}_i, \widetilde{p_i}\} \to t_i)_{i\in I}\texttt{end}$, is defined by rule [RBRANCH]. When a message is received (*i.e.,* $\alpha = ?\{\texttt{:l}, \widetilde{v}\}$), it is matched with a valid branch from the *receive* construct, using the label :l. Should one of the labels match ($\exists j \in I$ such that $\texttt{:l}_j = \texttt{:l}$), the payload of the message ($\widetilde{v}$) is compared to the corresponding patterns in the selected branch ($\widetilde{p_j}$) using $\mathbf{match}(\widetilde{p_j}, \widetilde{v})$. If the values match with the pattern, the **match** function (Definition 4.1) produces the substitutions σ, mapping the matched variables in the pattern $\widetilde{p_j}$ to values from $\widetilde{v}$. This substitution σ is then used to instantiate the free variables in continuation branch t_j.

Definition 4.1 *(Pattern Matching).* The **match** function pairs patterns with a corresponding value, resulting in a sequence of substitutions (called σ), *e.g.*, $\mathbf{match}(p, v) = [^{v_1}/_{x_1}][^{v_2}/_{x_2}] = [^{v_1, v_2}/_{x_1, x_2}]$. The match function builds a meta-level list of substitutions, which should not be confused with the lists defined by the Elixir syntax in Fig. 3.

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

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

Example 4.1. For the pattern $p_1 = \{x, 2, y\}$ and the value tuple $v_1 = \{8, 2, \mathit{true}\}$, $\mathbf{match}(p_1, v_1) = \sigma$ where $\sigma = [^{8}/_{x}][^{\mathit{true}}/_{y}]$ (written also as $\sigma = [^{8, \mathit{true}}/_{x, y}]$). However for pattern $p_2 = \{x, 2, \mathit{false}\}$, the operation $\mathbf{match}(p_2, v_1)$ fails, since p_2 expects a *false* value as the third element, but finds a *true* value instead. ■

Using rule [RCALL$_1$] from Fig. 8, a function call is evaluated by first reducing all of its parameters to a value, using the expression reduction rules in Fig. 9; again this models a call-by-value semantics. Once all arguments have been fully

$\boxed{e \to e'}$

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

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

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

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

Fig. 9. Expression reduction rules.

reduced, [rCall$_2$], the implicit environment Σ is queried for function f with arity n to fetch the function's parameter names and body. This results in a transition to the function body with its parameters instantiated accordingly, $t\,[{}^{\iota}/_{y}]\,[{}^{v_2,\ \ldots,\ v_n}/_{x_2,\ \ldots,\ x_n}]$, decorated by the function name, *i.e.,* $\alpha = f/n$. Along the same lines a case construct first reduces the expression which is being matched using rule [rCase$_1$]. Then, rule [rCase$_2$] matches the value with the correct branch, using the **match** function, akin to [rBranch]. Whenever a term consists solely of an expression, it silently reduces using [rExpression] using the expression reduction rules $e \to e'$ of Fig. 9. These are fairly standard.

5. Validation of the ElixirST type system

We validate the static properties imposed by the ElixirST type system, overviewed in Section 3, by establishing a relation with the runtime behaviour of a typechecked Elixir program, using the transition semantics defined in Section 4. Broadly, we establish a form of *type preservation*, which states that if a well-typed term transitions, the resulting term then remains well-typed [10]. However, our notion of type preservation, needs to be stronger to also take into account *(i)* the side-effects produced by the execution; and *(ii)* the progression of the execution with respect to protocol expressed as a session type. Following the long-standing tradition in the session type community, these two aspects are captured by the refined preservation property called *session fidelity* [15,16]. This property ensures that: *(i)* the communication action produced as a result of the execution of the typed process is one of the actions allowed by the current stage of the protocol; and that *(ii)* the resultant process following the transition is still well-typed w.r.t. the remaining part of the protocol that is still outstanding. We also establish a conditional form of *progress*, where well-typed processes are either a value, or else they can safely transition to a new term, producing an internal (or external) action.

Before embarking on the proofs for session fidelity and progress, we prove an auxiliary proposition that acts as a sanity check for our operational semantics of Section 4. We note that the operational semantics assume that only *closed* programs are executed; an *open* program (*i.e.,* a program containing free variables) is seen as an incomplete program that cannot execute correctly due to missing information. To this end, Proposition 1 ensures that a closed term *remains closed* even after transitioning.

Proposition 1 *(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. Refer to Appendix B.1 for details. □

The statement of the session fidelity property relies on the definition of a partial function called **after** (Definition 5.1), which takes a session type and an action as arguments and returns another session type as a result. This function serves two purposes: (i) the function $\textbf{after}(S, \alpha)$ is only defined for actions α that are (immediately) permitted by the protocol S, which allows us to verify whether a term transition step violated a protocol or not; and (ii) since S describes the current stage of the protocol to be followed, we need a way to evolve this protocol to the next stage should α be a permitted action, and this is precisely S', the continuation session type returned where $\textbf{after}(S, \alpha) = S'$.

Definition 5.1 *(After Function).* The **after** function is partial function defined for the following cases:

$$\begin{aligned}
\textbf{after}(S, \tau) &\stackrel{\text{def}}{=} S \\
\textbf{after}(S, f/n) &\stackrel{\text{def}}{=} S \\
\textbf{after}(\oplus\{!\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}, \iota!\{\mathtt{l}_j, \widetilde{v}\}) &\stackrel{\text{def}}{=} S_j \quad \text{where } j \in I \\
\textbf{after}(\&\{?\mathtt{l}_i(\widetilde{T}_i).S_i\}_{i\in I}, ?\{\mathtt{l}_j, \widetilde{v}\}) &\stackrel{\text{def}}{=} S_j \quad \text{where } j \in I
\end{aligned}$$

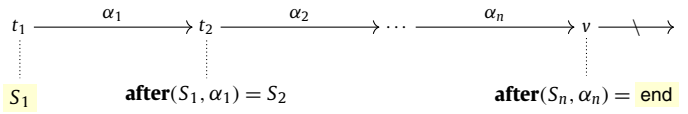


Fig. 10. Repeated applications of *session fidelity*.

This function is undefined for all other cases. The **after** function is overloaded to range over session typing environments (Δ) in order to compute a new session typing environment given some action α and session type S:

$$\mathbf{after}(\Delta,\, f/n,\, S) \stackrel{\text{def}}{=} \Delta,\, f/n : S$$

$$\mathbf{after}(\Delta, \alpha,\, S) \stackrel{\text{def}}{=} \Delta \qquad \text{if } \alpha \neq f/n$$

Intuitively, when the action produced by the transition is f/n, the session typing environment is extended by the new mapping $f/n : S$. For all other actions, the session typing environment remains unchanged. ■

Recall that module typechecking using rule [TMODULE] entails typechecking the bodies of all the public functions w.r.t. their ascribed session type, $\Delta \cdot \left(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\right) \vdash^{y}_{\Sigma} S \rhd t : T \lhd S'$ (where $S' = \mathtt{end}$ for this specific case). At runtime, a spawned client handler process in a session starts running the function body term t where the parameter variables $y, \widetilde{x}$ are instantiated with the *pid* of the client, say ι, and the function parameter values, say $\widetilde{v}$, respectively, $t\,[\iota/y]\,[\widetilde{v}/\widetilde{x}]$, as modelled in rule [RCALL$_2$] from Fig. 8. The instantiated function body is thus closed and can be typed w.r.t. an empty variable binding environment, $\Gamma = \emptyset$. Session fidelity thus states that if a closed term t is well-typed, *i.e.,*

$$\Delta \cdot \emptyset \vdash^{w} S \rhd t : \boxed{T} \lhd S' \tag{5}$$

(where S and S' are initial and residual session types, respectively, and T is the basic expression type) and this term t transitions to a new term t' with action α, *i.e.,*

$$t \xrightarrow{\alpha} t' \tag{6}$$

the new term t' is expected to remain well-typed, *i.e.,*

$$\Delta' \cdot \emptyset \vdash^{w} S'' \rhd t' : \boxed{T} \lhd S' \tag{7}$$

where the expanded Δ' is computed as $\mathbf{after}(\Delta, \alpha, S) = \Delta'$ and the base type of the term is preserved, as described by the constant type $\boxed{T}$ in eqs. (5) and (7). To understand how the evolved initial session type (S'') is computed or assumed (in eq. (7)), we have to consider the context of our analysis. Our type-checker analyses one side of an interaction that does not interleave with other sessions. We also assume that the interacting dual processes are well-behaved and thus follow a compatible protocol (*e.g.* the dual session type) during execution. Despite these restrictions, we obtain a certain degree of flexibility, where we can statically analyse processes interacting with external processes which we may not have access to their source.

To this end, we make a distinction between the transition actions in Equation (6). If the action α depends on the context process left implicit (*i.e.,* α is an incoming message), we only require the guarantee stated in Equation (7) whenever α is permitted by the protocol S *i.e.,* $S'' = \mathbf{after}(S, \alpha)$. More specifically, the condition $S'' = \mathbf{after}(S, \alpha)$ is part of the antecedent of the first clause in Session Fidelity Theorem and captures our conditional guarantees that apply only when the context process is well-behaved. On the other hand, if the action α depends solely on the process being typechecked (*i.e.,* α is an outputted message or an internal action), then we *require* that α is permitted by the protocol S. Concretely, the second clause of Session Fidelity Theorem is stronger and the condition $\mathbf{after}(S, \alpha)$ forms part of the succedent.

Theorem 2 *(Session Fidelity). If* $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$ *and* $t \xrightarrow[\Sigma]{\alpha} t'$

- *for* $\alpha = ?\{\mathtt{:}\iota,\, \widetilde{v}\}$ *and some session type* $S'' = \boldsymbol{after}(S, \alpha)$*, then there exists some* Δ'*, such that* $\Delta' \cdot \emptyset \vdash^{w}_{\Sigma} S'' \rhd t' : T \lhd S'$ *and* $\boldsymbol{after}(\Delta, \alpha, S) = \Delta'$
- *for* $\alpha \in \{f/n, \tau, \iota!\{\mathtt{:}\iota,\, \widetilde{v}\}\}$*, then there exists some* S'' *and* Δ'*, such that* $\Delta' \cdot \emptyset \vdash^{w}_{\Sigma} S'' \rhd t' : T \lhd S'$ *for* $\boldsymbol{after}(S, \alpha) = S''$ *and* $\boldsymbol{after}(\Delta, \alpha, S) = \Delta'$

Proof. By induction on the typing derivation $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$. Refer to Appendix B.2. □

As shown in Fig. 10, by repeatedly applying Theorem 2, we can therefore conclude that all the (external) actions generated as a result of a typed computation (*i.e.,* sequence of transition steps) must all be actions that follow the protocol

```
1  @dual "counter"
2  def client(server) do
3     x = send(server, {:stop})
4
5     receive do
6        {:value, num} -> num
7     end
8  end
```

Listing 4: Counter client obeying the protocol $\overline{counter}$.

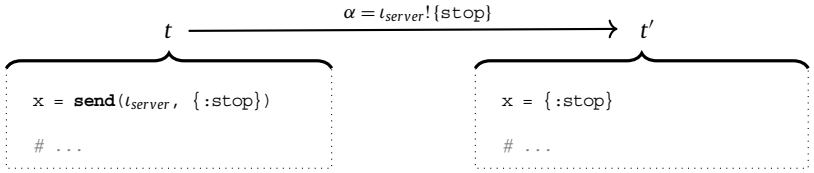
described by the initial session type. Since public functions are always typed with a residual session type end, certain executions could reach the case where the outstanding session is updated to end as well, *i.e.,* $\mathbf{after}(S_n, \alpha_n) = \mathsf{end}$. In such a case, we are guaranteed that the term will not produce further side-effects, as in the case of Fig. 10 where the term is reduced all the way down to some value, v.

Example 5.1. We consider a concrete example to show the importance of session fidelity. The function called `client`/1 sends a `stop` label and awaits a reply with the total value.

This function adheres to the following protocol (from eq. (2)):

$$\overline{counter} = \oplus \left\{ \begin{array}{l} !\texttt{incr}\ldots, \\ !\texttt{stop}().?\texttt{value}(\mathsf{number}).\mathsf{end} \end{array} \right\}$$

A process evaluating the function `client` executes by first sending a message containing a `stop` label to the interacting processes' *pid* (ι_{server}), as shown below.



As the process evaluates, the initial term t transitions to t', where it sends a message as a side-effect. This side-effect is denoted as an action α, where $\alpha = \iota_{server}![\texttt{stop}]$. By the After Function Definition, $\overline{counter}$ evolves to a new session type X:

$$\mathsf{X} = \mathbf{after}(\overline{counter}, \alpha) = ?\texttt{value}(\mathsf{number}).\mathsf{end}$$

For t' to remain well-typed, it must now match with the evolved session type X, where it has to be able to receive a message labelled `value`, before terminating. As a result, by the session fidelity property, we know that each step of execution will be in line with the original protocol. ■

We will also show that ElixirST observes a conditional form of the *progress* property, which describes how well-typed terms transition within the aforementioned analysis context. However, we consider the most problematic scenario first, where a process depends on external factors to progress. When a process executes a term that is following the branching session type, the process has to be able to handle (and pattern match) any valid incoming message. Concretely, a term t following the session type $\&\{\ldots\}$ has to be able to accept incoming messages, by (successfully) pattern matching them to one of the branches. This holds due to the [tBranch] rule (from Section 3.4) where we restrict all branching patterns to simple ones. In Proposition 3 we show that valid messages can always be matched to one of the branches.

Proposition 3. *Any well-typed term* $(\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S')$ *following the branching session type is able to process any* valid *incoming message, i.e.,*

$$\left.\begin{array}{r} t = \texttt{receive do}\ \left(\{:l_i, p_i^1, \ldots, p_i^n\} \to t_i\right)_{i\in I}\texttt{end} \\ S = \&\left\{?l_i\left(T_i^1, \ldots, T_i^n\right).S_i\right\}_{i\in I} \\ \mathbf{after}(S, ?\{l_k, v^1, \ldots, v^n\}) = S_k \ \textit{for some}\ k \in I \end{array}\right\} \implies \mathbf{match}(p_k^j, v^j)_{i\in 1..n}\ \textit{is defined}$$

Proof. Note that $\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$ and $t = \texttt{receive do}\ \left(\{:l_i, p_i^1, \ldots, p_i^n\} \to t_i\right)_{i\in I}\texttt{end}$ imply $\mathbf{simplepat}(p_i^j, T_i^j)_{i\in I, j\in 1..n}$. Refer to Appendix B.3. □

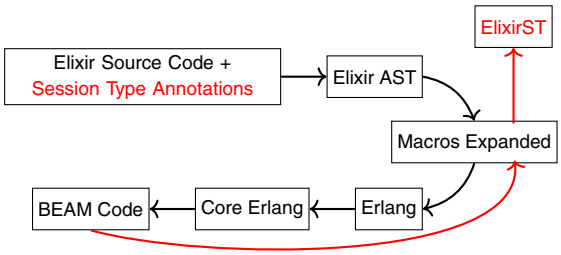


Fig. 11. Stages of Elixir compilation along with the session type implementation (in red).

By Proposition 3 we know that if a process depends on external factors (*i.e.,* a term expects an incoming message), then it should not have a problem to be able to handle these messages. From this, we can infer that a well-typed `receive` construct is able to transition to a new term, when a message is received. If we extend this to all terms, we can form the Progress Theorem, where we establish that a process can always transition from one form to another. If no further transitions are possible (see Fig. 10), then the process must be itself a (final) value. In this theorem we also establish that the action produced during the transition is always allowed by the protocol; this is checked using the **after** function.

Theorem 4 *(Progress). If* $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$*, then either* t *is a value, or else there exists some term* t' *and action* α *such that* $t \xrightarrow[\Sigma]{\alpha} t'$ *and* $\textbf{\textit{after}}(S, \alpha)$ *is defined*

Proof. By induction on the typing derivation $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$. Refer to Appendix B.3. □

6. Implementation

This section describes how the type system of Section 3 is implemented as the session type checker tool called ElixirST. This tool is integrated in Elixir with minimal changes to the syntax of the surface language. The source code is written in Elixir and is available open-source.

6.1. Elixir compilation with session types

The Elixir source code is compiled in several steps (see Fig. 11). The original Elixir source code is initially parsed into an Abstract Syntax Tree (AST). Then, Elixir's powerful macro system expands all non-*special form*[2] into *special form* macros (*e.g.*, `if`/`unless` statements are converted into `case` constructs) [17]. Afterwords, the expanded Elixir AST is converted into Erlang abstract format and Core Erlang. Finally, it is compiled into BEAM code which can be executed on the Erlang Virtual Machine (BEAM).

Our implementation integrates seamlessly within this compilation pipeline (see Fig. 11, red). Inside the Elixir source code, processes are described with a specific session type using annotations (starting with `@`). Annotations are able to hold information about a module during compile-time. We provide normal labelled session types (`@session`) and their dual (`@dual`, referenced by a label):

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

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

6.2. Bridging between Elixir and our model

Every construct in Fig. 3 maps directly to a corresponding construct in the actual Elixir language. The `@spec` annotation which decorates functions with types is already present in the latest distribution of the language. It is typically used for

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

code documentation and to statically analyse programs using the *Dialyzer* [11], a tool that detects potential (type) errors in Core Erlang programs using success typing [9]. We use the `@spec` information to specify the types for the parameters and the return type of the functions, supplementing our session typechecking analysis. A similar approach to ours was adopted by Cassola et al. [13] for a gradual static type system for Elixir.

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

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

The typing rules of Section 3 are also designed in a way to minimally alter common coding patterns in the language. For instance, branches in session types might have common continuations, such as !C().end in the type &{?A().!C().end, ?B().!C().end}. Many type systems force programs to structure their code as shown in the left-hand side code snippet above (which performs the same `send` action twice). However, in Elixir it is common to express this as a fork-join pattern, with a single continuation performing the common action once as shown in the right-hand side code snippet. Our type system can typecheck *both* code snippets.

The only aspect left to discuss is the mechanism used by our implementation to guarantee an interaction between the two processes implementing the respective endpoints of binary session type. To achieve this end, we implemented a bespoke spawning function (refer to Section 3.6) that takes the code of the respective endpoints and returns a tuple with the *pids* of the two processes that are already linked. The implementation of our session initiation is given below:

```
1  def session(serverFn, server_args, clientFn, client_args)
2     when is_function(serverFn) and is_function(clientFn) do
3     server_pid = spawn(fn ->
4                        receive do
5                           {:pid, client_pid} ->
6                                  apply(serverFn, [client_pid | server_args])
7                        end
8                     end)
9
10    client_pid = spawn(fn ->
11                       send(server_pid, {:pid, self()})
12                       apply(clientFn, [server_pid | client_args])
13                    end)
14
15     {server_pid, client_pid}
16 end
```

This modified `session/4` function takes two pairs of arguments: two references of function names (that should be spawned) and their list of arguments. The code implements the initialisation protocol depicted in Fig. 7. It first spawns one process (pre-server in Fig. 7 for line 3) and passes its *pid* to the second spawned process (pre-client in Fig. 7, as the variable `server_pid` on lines 11 and 12). Then, the pre-client process sends its *pid* to the pre-server process (line 11 and lines 4 and 5). At this point, both processes execute their respective functions to transform into the *actual* first and second processes participating in the session, passing the respective *pids* as the first argument of the executing functions (lines 6 and 12).

The current implementation of the `session/4` function can only launch two processes at a time, in line with the binary sessions. This can however, be extended to handle more than two processes in the case of hierarchical processes, where a process may interact in several separate binary sessions, similar to the notion of intuitionistic session types [18].

Our implementation still allows spawned processes to receive messages from *any* other process. Unfortunately, unsolicited messages can interfere with a session-typed process, since the receiver is not able to distinguish where the message is originating from in the present implementation. An improvement would be exploiting Elixir's ability to cherry-pick messages out-of-order from the queue using pattern matching. As soon as a session is launched, a unique session ID would be shared with the two parties, and each message exchanged between them would use this ID to identify the source (and destination). This would enable selective reads to filter unsolicited messages. Mostrous and Vasconcelos [19] created a similar system to distinguish messages by attaching a unique reference to each message.

6.3. Case study

We use ElixirST to verify an Elixir (CLI) application which interacts with a third-party service (*i.e.*, Duffel [20]). Duffel offers a real-time flight selling service, where flights can be fetched and booked via a REST application programming interface (API).

Our application is built as an Elixir module, called `FlightSystem`, which interacts with the Duffel Flight Server, as shown in Fig. 12. The module consists of a client which can request to book flights from the Duffel Server. This server can

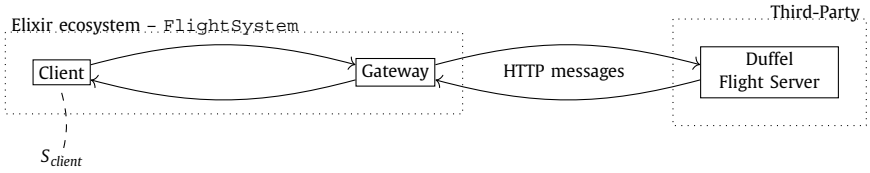


Fig. 12. Interactions with Duffel API.

only accept HTTP messages (*e.g.*, GET or POST requests), so we use a gateway which acts as a middleman between the client and the third-party server.

From the Duffel API documentation,[5] we can get a list of API calls that can be made, along with their details. Thus, we can *infer* an order (*i.e.,* a protocol) of calls that need to be invoked in the correct order, to achieve what we want. For instance, we consider a client that searches and books a flight. Concretely, the client has to interact with several endnodes, including `\offer_requests`, `\offers` and `\orders`, in a correct order. This order of interaction is formalised in the S_{client} protocol which is then used by ElixirST to ensure that a client process follows it, thus interacting safely (via a gateway process which converts the requests to HTTP messages) with the Duffel server.

$$S_{client} = \oplus\begin{cases} !\texttt{request}\big(\text{origin: binary, destination: binary,} \\ \qquad\qquad \text{dep\_date: binary, class: atom, pass\_no: number}\big).S_{offers}, \\ !\texttt{cancel}() \end{cases}$$

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

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

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)`).

This behaviour formalised by S_{client} is applied in the module `FlightSystem`, which is shown partly in Listing 5. `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 (line 4). 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 a compatible protocol, such as the dual session type, $\overline{S_{client}}$. This is not enforced, since the `gateway` function goes beyond the syntax defined in Section 3.3 (*e.g.*, uses other modules to parse JSON responses, or uses dynamic types) which are not defined by our type system in Section 3.4. This function may also be inaccessible for static analysis (*e.g.* proprietary source code), so we assume that at runtime it obeys $\overline{S_{client}}$. Nevertheless, we can initiate a session executing both public functions using our `session` function, as follows:

```
ElixirST.session(&FlightSystem.client/6,
                 ["MLA", "CDG", "2023-11-24", :economy, 2],
                 &FlightSystem.gateway/1,
                 [])
```

[5] https://duffel.com/docs/api/overview/welcome.

```
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: Session-typed snippet of a flight system written in Elixir.

ElixirST ensures that the function *client*/6 follows the pattern described by S_{client} which mirrors the implicit order of API requests by Duffel. For instance, if a client tries to make a booking before making a request containing the flight details (*i.e.,* skipping line 7), then the order will fail; this will be flagged by our type system earlier on instead of being rejected by Duffel at runtime.

This case study shows that ElixirST is flexible and practical enough to be integrated in real-world applications. In addition to verifying statically the individual functions, by explicitly adding information about the interactions, we provide a source of documentation within the source code itself.

In this case study we analysed the client side of the interaction, leaving the gateway process *unverified*, thus susceptible to behavioural issues. To fix this, we can use existing techniques, such as synthesising runtime monitors [21–23] for the unverified parts, similar to the work by Bartolo Burlò et al. [24].

7. Related work

In this section, we compare ElixirST with other type systems and implementations.

7.1. Type systems for Elixir

Cassola et al. [13,25] presented a gradual type system for Elixir. It statically typechecks the functional part of Elixir modules, using a gradual approach, where some terms may be left with an unknown expression type. In contrast to ElixirST, Cassola et al. analyse directly the unexpanded Elixir code which results in more explicit typechecking rules. Also, they focus on the static type system without formulating the operational semantics.

Another static type-checker for Elixir is *Gradient* [26]. It is a wrapper for its Erlang counterpart tool and takes a similar approach to [13], where gradual types are used. Another project, *TypeCheck* [27], adds dynamic type validations to Elixir programs. *TypeCheck* performs runtime typechecking by wrapping checks around existing functions. *Gradient* and *TypeCheck* are provided as an implementation only, without any formal analysis. In contrast to ElixirST, the discussed type-checkers [13, 26,27] analyse the sequential part of the Elixir language omitting any checks related to message-passing between processes.

Some implementations aim to check issues related to message-passing. Harrison [28] statically checks Core Erlang for such issues. For instance, it detects orphan messages (*i.e.,* messages that will never be received) and unreachable receive branches. In separate work, Christakis and Sagonas [29] analyse Core Erlang code to construct a communication graph which depicts the message flow between different process. This is then used to detect errors of similar kind, *e.g.* `receive` constructs that never receive any messages. This work was implemented as part of the Dialyzer. Harrison [30] extends [28] to analyse Erlang/OTP behaviours (*e.g.*, `gen_server`, which structures processes in a hierarchical manner) by injecting

runtime checks in the code. Compared to our work, [28,30,29] perform automatic analysis of the implementation, where they analyse send and receive primitives against each other. They analyse messages on a fine-grained level, which contrast with our work that uses a general protocol (*e.g.*, session types) describing the full interaction within a session.

Another type system for Erlang was presented by Svensson et al. [31]. Their body of work covers a larger subset of Erlang to what would be its equivalent in Elixir covered by our work. Moreover, its multi-tiered semantics captures an LTS defined over systems of concurrent actors. Although we opted for a smaller subset, we go beyond the pattern matching described by Svensson et al. since we perform a degree of typechecking for base types (*e.g.* in the premise of [tBranch]).

7.2. Session type systems

Closest to our work is [19], where Mostrous and Vasconcelos introduced session types to a fragment of Core Erlang, a dynamically typed language linked to Elixir. Their type system tags each message exchanged with a unique reference. This allows multiple sessions to coexist, since different messages could be matched to the corresponding session, using correlation sets. Mostrous and Vasconcelos take a more theoretical approach, so there is no implementation for [19]. Their type system guarantees *session fidelity* by inspecting the processes' mailboxes where, at termination, no messages should be left unprocessed in their mailboxes. Our work takes a more restrictive but pragmatic approach, where we introduce session types for functions within a module. We offer additional features, including variable binding (*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 session-based runtime monitoring tool for Python was initially presented by Neykova and Yoshida [32]. They use the Scribble [33] language to write *multiparty* session type (MPST) [16] protocols, which are then used to monitor the processes' actions. Different processes are ascribed a role (defined in the MPST protocol) using function decorators (*e.g.* `@role`), which is similar to how we annotate functions with protocols (*e.g.* using `@session`). Similar to [32], Fowler [34] presented an MPST implementation for Erlang. This implementation uses Erlang/OTP behaviours (*e.g.*, `gen_server`), which take into account Erlang's *let it crash* philosophy, where processes may fail while executing. Neykova and Yoshida [35] extend process monitoring in Erlang to provide a recovery strategy for the failed processes, ensuring that all of the failed (or affected) processes are restarted safely. All of these Erlang tools accept a more flexible language than the one allowed by our work. This is done at an added runtime cost, since they flag issues at runtime, whereas our work provides static guarantees that flags issues at pre-deployment stages. Moreover, our work is able to statically analyse part of the code (and give static guarantees for it) without requiring access to the entire codebase.

Scalas and Yoshida [36] applied binary session types to the Scala language, where session types are abstracted as Scala classes. Session fidelity is ensured using Scala's compiler, which complains if an implementation does not follow its ascribed protocol. Bartolo Burlò et al. [24] extended the aforementioned work, to monitor one side of an interaction statically and the other side dynamically using runtime monitors. These works relegate linearity checks to runtime. In contrast, ElixirST statically ensures that annotated implementations fully exhaust their associated protocol *once*. Another implementation was done by Scalas et al. [37,38], where session types were added in Scala 3. This design utilises dependent function types and model checking to verify programs at compile-time.

Harvey et al. [39] presented a new actor-based language, called EnsembleS, which offers session types as a native feature of 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 obey a known protocol. Thus, actors can be terminated and discovered at runtime, while still maintaining static correctness.

There have been several binary [40,41] and multiparty [42,43] session type implementations for Rust. These implementations exploit Rust's affine type system to guarantee that channels mirror the actions prescribed by a session type. Padovani [44] created a binary session type library for OCaml to provide static communication guarantees. This project was extended [45] to include dynamic contract monitoring which flags violations at runtime. The approaches used in the Rust and OCaml implementations rely heavily on type-level features of the language, which do not readily translate to the dynamically typed Elixir language. When we compare our work to the aforementioned work, we notice several limitations. ElixirST only supports a limited form of spawning (discussed in Section 6.2), where we constrain the number of processes in a single session to two processes. This contrasts to the unbounded number of parallel processes that are allowed in the π-calculus, where session types were first introduced [46]. Another aspect that we have not discussed is the lack of *session delegation*. ElixirST does not allow processes to hand over the remaining session to other processes. This stems from the approach that we use; our tool typechecks actors directly, whereas the aforementioned works [40–45] typecheck channel endpoints that can easily be transferred between different processes.

Actor-like techniques are also used in Active Objects (AO) based languages to combine the concept of process separation, with asynchronous method calls in object-oriented languages. Session types are utilised to structure method calls in such languages. For instance, Kamburjan et al. [47,48] added session types to ABS, which is an AO language that uses futures to resolve the results derived from method invocations. Kamburjan et al. use global protocols (stemming from MPSTs) to define the order of method invocation that originate from object instances. Although these global protocols are used for dynamic checking, they are also used to check each method statically on a more localised scale, using local types. ABS uses (abstracted) Erlang processes to structure their concurrent backend, similar to the backend structure used in our work.

Actor systems are notoriously hard to analyse statically. The main reason for this is that actors are open to receive any kind of messages, which makes it difficult to predict or analyse their behaviour. Our work takes a lenient view of this, where we simply ignore malformed messages using Elixir's selective `receive` construct. On the other hand, well-formed but *unsolicited* messages can cause behavioural issues. Other works handle this by typing the actors' mailboxes directly rather than their behaviour. De'Liguoro and Padovani [49] introduced a mailbox calculus which considers mailboxes as first-class citizens. This calculus adds types to mailboxes, thus ensuring that processes are free from behavioural issues, such as deadlocks. Fowler et al. [50] built on this to implement mailbox types within a practical concurrent programming language. These works [47,48,50] are presented with an implementation for purpose-built bespoke languages. The aims of our work are different since we start with an industry strength language and try to retrofit session type mechanisms so as to support the existing design patterns of the language.

8. Conclusion

In this work we established a correspondence between the ElixirST type system [7] and the runtime behaviour of a client handler running an Elixir module function that has been typechecked w.r.t. its session type protocol. In particular, we showed that this session-based type system observes the standard *session fidelity* property, meaning that processes executing a typed function *always* follow their ascribed protocols at runtime. This property provides the necessary underlying guarantees to attain various forms of communication safety, whereby should two processes following mutually compatible protocols (*e.g.* S and its dual $\overline{S}$), they avoid certain communication errors (*e.g.*, a send statement without a corresponding receive construct).

Future work There are a number of avenues we intend to pursue. One line of investigation is the augmentation of protocols that talk about multiple entry points to a module perhaps from the point of view of a client that is engaged in multiple sessions at one time, possibly involving multiple modules. The obvious starting points to look at here are the well-established notions of multiparty session types [16,38] or the body of work on intuitionistic session types organising processes hierarchically [51,18]. Another natural extension to our work would be to augment our session type protocol in such a way to account for process failure and supervisors, which is a core part of the Elixir programming model. For this, we will look at previous work on process/session type extensions that account for failure [52,53,39,54–57]. We also plan to augment our session typed protocols to account for resource usage and cost, along the lines of [58,59]. It should also be relatively straightforward to integrate the elaborate expression typechecking mechanisms developed by Castagna et al. [60] to replace our (limited) expression typing in Fig. 5. This would considerably enhance the expressivity of our framework to handle a wider range of Elixir programs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Additional definitions

In this appendix, we formalise some auxiliary definitions that are used in Sections 3–5 and Appendix B.

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

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

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

Definition A.2 *(Bound Variables).*

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

Definition A.3 *(Variable Substitution).*

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

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

Definition A.4 *(Type).* This partial function defines the types of basic values.

$$\mathbf{type}(\mathit{boolean}) \stackrel{\text{def}}{=} \mathsf{boolean} \qquad \mathbf{type}(\mathit{number}) \stackrel{\text{def}}{=} \mathsf{number}$$

$$\mathbf{type}(\mathit{atom}) \stackrel{\text{def}}{=} \mathsf{atom} \qquad \mathbf{type}(\iota) \stackrel{\text{def}}{=} \mathsf{pid},\ \text{where } \iota \text{ is a } \mathit{pid} \text{ instance}$$ ■

Definition A.5 *(Variable Patterns).* Computes an ordered set of variables from a given pattern p.

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

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

Definition A.6 *(Function Details).* We can extract function details (*i.e.,* `params`, `body`, `param_types`, `return_type`, `dual`) from a list of functions $(\widetilde{Q})$ and build a mapping, using set-comprehension, as follows. The list of functions $(\widetilde{Q})$ may consist of public (D) and private (P) functions.

$$\mathbf{details}(\widetilde{Q}) \stackrel{\text{def}}{=} \left\{ f/n : \begin{bmatrix} \texttt{dual} = y,\ \texttt{params} = \widetilde{x}, \\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{return\_type} = T,\ \texttt{body} = t \end{bmatrix} \;\middle|\; \begin{bmatrix} \texttt{[@session "}S\texttt{"]} \\ \texttt{@spec}\ f\,(\mathsf{pid}, \widetilde{T})\ ::\ T \\ \texttt{def[p]}\ f\,(y, \widetilde{x})\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{Q} \right\}$$ ■

Definition A.7 *(Functions Names and Arity).* This definition takes the set of all public function $(\widetilde{D})$ as input, and returns a set of all public function names and their arity.

$$\mathbf{functions}(\widetilde{D}) \stackrel{\text{def}}{=} \left\{ f/n \;\middle|\; \begin{bmatrix} \texttt{@session...;}\ \texttt{@spec...} \\ \texttt{def}\ f\,(y, x_2,\ \ldots,\ x_n)\ \texttt{do}\ t\ \texttt{end} \end{bmatrix} \in \widetilde{D} \right\}$$ ■

Definition A.8 *(All Session Types).* The function **sessions**$(\widetilde{D})$, returns the session type corresponding to each annotated public function.

$$\textbf{sessions}(\widetilde{D}) \stackrel{\text{def}}{=} \left\{ f/n : S \;\middle|\; \begin{bmatrix} \texttt{@session "}S\texttt{"; @spec...} \\ \texttt{def } f(y, x_2, \ldots, x_n) \texttt{ do } t \texttt{ end} \end{bmatrix} \in \widetilde{D} \right\}$$

In case the `@dual` annotation is used instead of `@session`, the dual session type is computed automatically. ■

Appendix B. Proofs

In this appendix, we present the complete proofs of Proposition 1, Theorems 2 and 4, which were omitted from the main text.

B.1. Proofs for Proposition 1

Before proving Proposition 1, we must analyse some properties related to closed terms, where we see how they affect variable substitutions (Definition A.3). Lemma 5 states that if a variable x does not exist inside a term t, then, if we initiate x with some value, term t must remain unaffected, *i.e.,* $t\,[^{v}/_{x}] = t$. Restricting this statement, Corollary 6 states that, if x is not a free variable in t, then the same result should hold. Lemma 7 consists of two statements that compare the free variables in terms (or expressions) with those that include a substitution.

Lemma 5. *$x \notin \boldsymbol{fv}(t) \cup \boldsymbol{bv}(t)$ implies $t\,[^{v}/_{x}] = t$*

Proof. By induction on the structure of t. □

Corollary 6. *$x \notin \boldsymbol{fv}(t)$ implies $t\,[^{v}/_{x}] = t$*

Proof. A consequence of Lemma 5. □

Lemma 7.

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

Proof. By induction on the structure of p.

[$p = b$] The function $\textbf{match}(b, v)$ succeeds only when $v = b$. So, by the **match** definition, when $v = b$,

$$\textbf{match}(b, b) = [\,] \tag{B.1a}$$

By the **vars** definition, $\textbf{vars}(b) = \emptyset$, which matches the result from eq. (B.1a) since no variables where substituted.

[$p = x$] By the **match** definition, for any v,

$$\textbf{match}(x, v) = [^{v}/_{x}] \tag{B.1b}$$

By the **vars** definition, $\textbf{vars}(x) = \{x\}$, which matches the variable in the substitution of eq. (B.1b).

[$p = [\,w_1 \mid w_2\,]$] By the **match** definition, for $v = [\,v_1 \mid v_2\,]$,

$$\textbf{match}(p, v) = \textbf{match}(w_1, v_1), \textbf{match}(w_2, v_2) = [^{\widetilde{v_1}}/_{\widetilde{x_1}}]\,[^{\widetilde{v_2}}/_{\widetilde{x_2}}] \text{ where} \tag{B.1c}$$

$$\textbf{match}(w_1, v_1) = [^{\widetilde{v_1}}/_{\widetilde{x_1}}] \tag{B.1d}$$

$$\textbf{match}(w_2, v_2) = [^{\widetilde{v_2}}/_{\widetilde{x_2}}] \tag{B.1e}$$

By case analysis of w_1 and w_2 from eqs. (B.1d) and (B.1e), we conclude that

$$\textbf{vars}(w_1) = \{\widetilde{x_1}\} \tag{B.1f}$$

$$\textbf{vars}(w_2) = \{\widetilde{x_2}\} \tag{B.1g}$$

We need to show that $\textbf{vars}([\,w_1 \mid w_2\,]) = \{\widetilde{x_1}, \widetilde{x_2}\}$. By the **vars** definition and eqs. (B.1f) and (B.1g), $\textbf{vars}([\,w_1 \mid w_2\,]) = \textbf{vars}(\widetilde{x_1}) \cup \textbf{vars}(\widetilde{x_2}) = \{\widetilde{x_1}\} \cup \{\widetilde{x_2}\}$. This result matches the variables in the substitutions of eq. (B.1c).

[$p = \{w_1, \ldots, w_n\}$] Similar to the previous case. □

Lemma 9 *(Closed Expression).* $\mathbf{fv}(e) = \emptyset$ *and* $e \to e'$ *implies* $\mathbf{fv}(e') = \emptyset$

Proof. By induction on the structure of e. □

Lemmata 5–9 allow us to prove Closed Term Proposition (Proposition 1). 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 9 is analogous; it states that expressions remain closed after reductions.

Proposition 1 *(Closed Term).* *If* $\mathbf{fv}(t) = \emptyset$ *and* $t \xrightarrow{\alpha} t'$, *then* $\mathbf{fv}(t') = \emptyset$

Proof. By induction on the structure of t.

[$t = e$] Holds immediately by the rule [rExpression] and the losed Expression] Lemma.
[$t = (x = t_1;\ t_2)$] Given that current structure of t, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

1. [**rLet$_1$**] From the rule, $t' = (x = t_1';\ t_2)$ and

$$t_1 \xrightarrow{\alpha} t_1' \tag{B.2a}$$

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

$$\mathbf{fv}(t_1) = \emptyset \tag{B.2b}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{B.2c}$$

If we apply the inductive hypothesis to eqs. (B.2a) and (B.2b), we get

$$\mathbf{fv}(t_1') = \emptyset \tag{B.2d}$$

So, by eqs. (B.2c) and (B.2d) and the definition of **fv**, we get $\mathbf{fv}(x = t_1';\ t_2) = \emptyset$ as required.

2. [**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

$$\mathbf{fv}(v) = \emptyset \tag{B.2e}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{B.2f}$$

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 6, $t_2 = t_2\,[{}^{v}/_{x}]$. Substituting this in eq. (B.2f), results in $\mathbf{fv}(t_2\,[{}^{v}/_{x}]) = \emptyset$, as required.
   b. If $x \in \mathbf{fv}(t_2)$, then by Lemma 7, we get $\mathbf{fv}(t_2\,[{}^{v}/_{x}]) = \mathbf{fv}(t_2) \setminus \{x\}$. If we substitute this in eq. (B.2f), 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. [**rChoice$_1$**] From this rule, we know that $\alpha = \tau$ and

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

$$e_k \to e_k' \tag{B.3a}$$

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

$$\mathbf{fv}(\iota) = \emptyset \tag{B.3b}$$

$$\mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..k-1 \tag{B.3c}$$

$$\mathbf{fv}(e_i) = \emptyset \text{ for } i \in k..n \tag{B.3d}$$

Applying the Closed Expression Lemma to eqs. (B.3a) and (B.3d), results in $\mathbf{fv}(e_k) = \emptyset$. Using this information along with eqs. (B.3b–d) and the **fv** definition, results in $\mathbf{fv}(t') = \emptyset$ as required.

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

$$\mathbf{fv}(\iota) = \emptyset, \quad \mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..n \tag{B.3e}$$

To show that $\mathbf{fv}(\{\texttt{:l}_\mu, v_1, \ldots, v_n\}) = \emptyset$, we can apply eq. (B.3e) to the **fv** definition.

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

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p}_i) = \emptyset \quad \text{for all } i \in I \tag{B.4a}$$

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

$$\mathbf{match}(\widetilde{p}_j, \widetilde{v}) = \sigma \text{ where } \sigma = [{}^{v'_1, \ldots, v'_k}/_{x_1, \ldots, x_k}] \tag{B.4b}$$
$$t' = t_j\sigma$$

From eq. (B.4b), we can apply Lemma 8 to get

$$\mathbf{vars}(\widetilde{p}_j) = \{x_1, \ldots, x_k\} \tag{B.4c}$$

Substituting eq. (B.4c) in eq. (B.4a) (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 7, 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 6. 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

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

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

$$\mathbf{fv}(v_i) = \emptyset \quad \text{for all } i \in 1..k-1 \tag{B.5b}$$
$$\mathbf{fv}(e_i) = \emptyset \quad \text{for all } i \in k..n \tag{B.5c}$$

Applying the Closed Expression Lemma to eqs. (B.5a) and (B.5c) (for $i = k$), we get

$$\mathbf{fv}(e_k) = \emptyset \tag{B.5d}$$

So, using the **fv** definition with eqs. (B.5b–d), result $\mathbf{fv}(t') = \emptyset$ holds as expected.

2. [**RCall$_2$**] From the rule, we know that $\alpha = f/n$ and

$$t = f(\iota, v_2, \ldots, v_n) \tag{B.5e}$$
$$t' = \bar{t}[{}^{\iota}/_{y}][{}^{v_2, \ldots, v_n}/_{x_2, \ldots, x_n}]$$
$$\Sigma(f/n) = \Omega \qquad \Omega.\texttt{body} = t \qquad \Omega.\texttt{params} = x_2, \ldots, x_n \qquad \Omega.\texttt{dual} = y \tag{B.5f}$$

Since term reduction can only happen with respect to a well-formed function information environment Σ, we can assume that the only free variables in a function body are the parameter types, or formally, for all $f/n \in \mathbf{dom}(\Sigma)$, we have

$$\mathbf{fv}\big(\Sigma(f/n).\texttt{body}\big) \setminus \big(\Sigma(f/n).\texttt{params} \cup \{\Sigma(f/n).\texttt{dual}\}\big) = \emptyset$$

Thus, using this information and substituting the information from eq. (B.5f), we get

$$\mathbf{fv}(\bar{t}) \setminus \{y, x_2, \ldots, x_n\} = \emptyset \tag{B.5g}$$

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 7, 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 6. 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.

[$t = \mathbf{case}\ e\ \mathbf{do}\ (p_i \to t_i)_{i\in I}\ \mathbf{end}$] Given that current structure of t, which implicitly contains the catch-all pattern, 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

$$e \to e' \tag{B.6a}$$

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

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i) = \emptyset \quad \text{for all } i \in I \tag{B.6b}$$

$$\mathbf{fv}(e) = \emptyset \tag{B.6c}$$

Applying Closed Expression Lemma to eqs. (B.6a) and (B.6c), results in $\mathbf{fv}(e') = \emptyset$. Thus, using this information, along with eq. (B.6b) and the **fv** definition, we get $\mathbf{fv}(t') = \emptyset$ as needed.

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 $j \in I$,

$$\mathbf{match}(p_j, v') = \sigma \text{ where } \sigma = [^{v_1,\ \ldots,\ v_n}/_{x_1,\ \ldots,\ x_n}] \tag{B.6d}$$

$$t' = t_j\sigma \tag{B.6e}$$

From the premise, we know that $\mathbf{fv}(t) = \emptyset$, so by the **fv** definition, $\mathbf{fv}(v') = \emptyset$ and

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i}) = \emptyset \quad \text{for all } i \in I \tag{B.6f}$$

From eq. (B.6d), we can apply Lemma 8, to get

$$\mathbf{vars}(p_j) = \{x_1,\ \ldots,\ x_k\} \tag{B.6g}$$

Substituting eq. (B.6g) in eq. (B.6f) (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. □

B.2. Proofs for Theorem 2

Before proving Theorem 2, we consider some other necessary lemmata. The Δ-Weakening Lemma weakens (*i.e.,* extends) the session typing environment (Δ) without affecting the overall typing result.

Lemma 10 *(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:

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

$$(\Delta,\ f/n : S) \cdot \Gamma' \vdash^y S \rhd \bar{t} : T \lhd S' \tag{B.7a}$$

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{B.7b}$$

Applying the inductive hypothesis to eq. (B.7a) results in $\big(\Delta, \Delta',\ f/n : S\big) \cdot \Gamma' \vdash^y S \rhd t : T \lhd S'$, where we assume that $f/n \notin \mathbf{dom}(\Delta')$. So, using the latter result, eq. (B.7b) and [tUnknownCall] results in $(\Delta, \Delta') \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$, as required.

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

$$\Delta\,(f/n) = S \tag{B.8a}$$

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{B.8b}$$

If we extend Δ by Δ', then $(\Delta, \Delta')(f/n) = S$ remains valid. So, using this information, along with eq. (B.8b) in [tKnownCall], 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. □

The type system observes the session fidelity property if well-typed terms remain 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 11) ensures that when free variables inside of terms and expressions are substituted with a closed value,

the resulting terms and expressions remain well-typed. As a result, the substituted variables become redundant in *variable binding* environment (Γ), and thus can be removed from Γ. This lemma consists of two statements, where substitution is performed in *(i)* terms, and *(ii)* expressions.

Lemma 11 *(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. We show the main cases for Item i:

[TLET] From the rule, we know that $t = (x' = t_1;\ t_2)$, and

$$x' \neq w \tag{B.9a}$$
$$\Gamma \vdash_{\text{exp}} v : T' \tag{B.9b}$$
$$\Delta \cdot (\Gamma, x : T') \vdash^{w} S \triangleright t_1 : T'' \triangleleft S'' \tag{B.9c}$$
$$\Delta \cdot (\Gamma, x : T', x' : T'') \vdash^{w} S'' \triangleright t_2 : T \triangleleft S' \tag{B.9d}$$

The *variable binding* environment of eq. (B.9d) can be reordered to

$$\Delta \cdot (\Gamma, x' : T'', x : T') \vdash^{w} S'' \triangleright t_2 : T \triangleleft S' \tag{B.9e}$$

We need to show that $\Delta \cdot \Gamma \vdash^{w[v/x]} S \triangleright (x' = t_1;\ t_2)\,[v/x] : T \triangleleft S'$, which by the Variable Substitution Definition, is equivalent to

$$\Delta \cdot \Gamma \vdash^{w[v/x]} S \triangleright x' = t_1\,[v/x];\ t_2\,[v/x] : T \triangleleft S' \tag{B.9f}$$

for $x \neq x'$ and $x' \neq v$. To obtain eq. (B.9f), we need some preliminary results. Applying the inductive hypothesis to eqs. (B.9b) and (B.9c), and similarly to eqs. (B.9b) and (B.9e), results in

$$\Delta \cdot \Gamma \vdash^{w[v/x]} S \triangleright t_1\,[v/x] : T'' \triangleleft S'' \tag{B.9g}$$
$$\Delta \cdot (\Gamma, x' : T'') \vdash^{w[v/x]} S'' \triangleright t_2\,[v/x] : T \triangleleft S' \tag{B.9h}$$

From eq. (B.9a) and the Variable Substitution Definition we know that $x \neq w\,[v/x]$. Applying this information, along with eqs. (B.9g) and (B.9h) to the premise of [TLET] results in eq. (B.9f), as required.

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

$$\Gamma \vdash_{\text{exp}} v : T' \tag{B.10a}$$
$$S = \&\left\{?\mathtt{l}_i\left(T_i^1,\ \ldots,\ T_i^n\right).S_i\right\}_{i \in I}$$
$$t = \mathtt{receive\ do}\left(\left\{:\mathtt{l}_i, p_i^1,\ \ldots,\ p_i^n\right\} \to t_i\right)_{i \in I}\mathtt{end} \tag{B.10b}$$

From the premise, we also know that, for all $i \in I$ and $j \in 1..n$:

$$\mathbf{simplepat}(p_i^j, T_i^j) \tag{B.10c}$$
$$\vdash^{w}_{\text{pat}} p_i^j : T_i^j \ \triangleright\ \Gamma_i^j \tag{B.10d}$$
$$\Delta \cdot \left(\Gamma, x : T', \Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash^{w} S_i \triangleright t_i : T \triangleleft S' \tag{B.10e}$$

This case holds if the following statement is obtained:

$$\Delta \cdot \Gamma \vdash^{w[v/x]} S \triangleright t\,[v/x] : T \triangleleft S' \tag{B.10f}$$

where $t\,[v/x] = \mathtt{receive\ do}\left(\left\{:\mathtt{l}_i, p_i^1,\ \ldots,\ p_i^n\right\} \to\right)_{i \in I}\mathtt{end}$. To obtain eq. (B.10f) we need to use the [TBRANCH] rule which requires multiple premises. Applying the inductive hypothesis to eqs. (B.10a) and (B.10e) results in

$$\Delta \cdot \left(\Gamma, \Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash^{w[v/x]} S_i \triangleright t_i\,[v/x] : T \triangleleft S' \quad \text{for all } i \in I \tag{B.10g}$$

If $w \neq x$, then eq. (B.10d)

$$\vdash^{w[v/x]}_{\text{pat}} p_i^j : T_i^j \ \triangleright\ \Gamma_i^j \quad \text{for all } j \in 1..n \tag{B.10h}$$

since by the Variable Substitution Definition, $w = w[^{v}/_{x}]$. Therefore, eqs. (B.11c), (B.10g) and (B.10h) can be applied to the premise of [TBRANCH] to obtain eq. (B.10f):

$$\Delta \cdot \Gamma \vdash^{w[^{v}/_{x}]} S \rhd t[^{v}/_{x}] : T \lhd S'$$

which is the required result. In case when $w = x$, then an additional mapping may be obtained from the pattern type rule which maps the dual *pid* to some type. However, since in this case x would be substituted to a variable, then the extra mapping does not affect the result, obtaining eq. (B.10f) as required.

[**TCHOICE**] From the rule, we know that for some $\mu \in I$, $T = \{\mathsf{atom}, T^1_\mu, \ldots, T^n_\mu\}$, $S = \oplus\{!\mathtt{l}_\mu(\widetilde{T_\mu}).S_\mu\}_{i\in I}$ and

$$t = \mathtt{send}\left(\iota, \{:\mathtt{l}_\mu, e_1, \ldots, e_n\}\right) \tag{B.11a}$$

$$\Gamma, x : T' \vdash_{\mathsf{exp}} e_j : T^j_\mu \quad \text{for all } j \in 1..n \tag{B.11b}$$

$$\Gamma \vdash_{\mathsf{exp}} v : T' \tag{B.11c}$$

Applying eqs. (B.11b) and (B.11c) to Item ii of Lemma 11 results in $\Gamma \vdash_{\mathsf{exp}} e_j[^{v}/_{x}] : T^j_\mu$ for all $j \in 1..n$. Applying this result to [TCHOICE] results in

$$\Delta \cdot \Gamma \vdash^{w[^{v}/_{x}]} S \rhd t[^{v}/_{x}] : T \lhd S'$$

which is the required result, since $t[^{v}/_{x}] = \mathtt{send}\left(w[^{v}/_{x}], \{:\mathtt{l}_\mu, e_1[^{v}/_{x}], \ldots, e_n[^{v}/_{x}]\}\right)$. □

Lemma 12 links expression types to the basic values (and vice versa), *e.g.* the value 5 has type number.

Lemma 12 *(Value Typing).*

i. $\Gamma \vdash_{exp} v : boolean$ *iff* $v = boolean$
ii. $\Gamma \vdash_{exp} v : number$ *iff* $v = number$
iii. $\Gamma \vdash_{exp} v : atom$ *iff* $v = atom$
iv. $\Gamma \vdash_{exp} v : pid$ *iff* $v = \iota$
iiv. $\Gamma \vdash_{exp} v : [T]$ *iff* $v = [v_1 \mid v_2]$ *or* $v = [\,]$
iiiv. $\Gamma \vdash_{exp} v : \{\widetilde{T}\}$ *iff* $v = \{\widetilde{v'}\}$

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

Lemma 13 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 14 provides the same guarantees but for a *sequence* of patterns and values.

Lemma 13. *For all patterns p and values v,*

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

Proof. By induction on the definition $\textbf{match}(p, v)$. We proceed by case analysis:

[$p = b, v = b$] By the definition, $\textbf{match}(b, b) = [\,]$, so no substitutions are expected. By $\vdash^{w}_{\mathsf{pat}} b : T \rhd \Gamma$ and [TPBASIC], the *variable binding* environment (*i.e.,* Γ) must be empty, so case holds immediately.

[$p = x$] By definition, $\textbf{match}(x, v) = [^{v}/_{x}]$, and from the premise we know that

$$\emptyset \vdash_{\mathsf{exp}} v : T. \tag{B.12a}$$

From $\vdash^{w}_{\mathsf{pat}} x : T \rhd \Gamma$ and [TPVARIABLE], we know that Γ must contain $x : T$ only. Therefore, case holds by eq. (B.12a).

[$p = [\,w_1 \mid w_2\,], v = [\,v_1 \mid v_2\,]$] Using the **match** definition, $\textbf{match}([\,w_1 \mid w_2\,], [\,v_1 \mid v_2\,]) = \textbf{match}(w_1, v_1), \textbf{match}(w_2, v_2)$, or equivalently

$$\textbf{match}(w_1, v_1) = [^{v'_1, \ldots, v'_j}/_{x_1, \ldots, x_j}] \tag{B.13a}$$

$$\textbf{match}(w_2, v_2) = [^{v'_k, \ldots, v'_n}/_{x_k, \ldots, x_n}] \text{ where } k = j + 1 \tag{B.13b}$$

From the premise, applying [TLIST] to $\emptyset \vdash_{\mathsf{exp}} [\,v_1 \mid v_2\,] : [\,T\,]$, results in

$$\emptyset \vdash_{\mathsf{exp}} v_1 : T \text{ and } \emptyset \vdash_{\mathsf{exp}} v_2 : [\,T\,] \tag{B.13c}$$

Applying also [TPLIST] to $\vdash^{w}_{\mathsf{pat}} [\,w_1 \mid w_2\,] : [\,T\,] \rhd \Gamma$, results in

$$\vdash^{w}_{\text{pat}} w_1 : T \ \triangleright\ \Gamma' \text{ and } \vdash^{w}_{\text{pat}} w_2 : [\,T\,] \ \triangleright\ \Gamma'' \tag{B.13d}$$

Applying the inductive hypothesis twice to eqs. (B.13a–d) results in

$$\Gamma' = x_1 : T_1,\ \ldots,\ x_j : T_j \text{ and } \Gamma'' = x_k : T_k,\ \ldots,\ x_n : T_n \tag{B.13e}$$

$$\emptyset \vdash_{\text{exp}} v'_i : T_i \text{ for all } i \in 1..n \tag{B.13f}$$

Therefore, case holds by eqs. (B.13e) and (B.13f), since $\Gamma = \Gamma', \Gamma''$.

$[p = \{w_1, \ldots, w_m\}, v = \{v_1, \ldots, v_m\}]$ Using the **match** definition, $\textbf{match}(\{w_1, \ldots, w_m\}, \{v_1, \ldots, v_m\}) = \textbf{match}(w_1, v_1), \ldots, \textbf{match}(w_m, v_m) = \sigma$, or equivalently, for $i \in 1..m$,

$$\textbf{match}(w_i, v_i) = \sigma_i \text{ given that } \sigma = \sigma_1, \ldots, \sigma_m \tag{B.14a}$$

From $\emptyset \vdash_{\text{exp}} \{v_1, \ldots, v_2\} : \{T_1, \ldots, T_m\}$, by [TTUPLE], we know that

$$\emptyset \vdash_{\text{exp}} v_i : T_i \tag{B.14b}$$

Applying also [TPTUPLE] to $\vdash^{w}_{\text{pat}} \{w_1, \ldots, w_m\} : \{T_1, \ldots, T_m\} \ \triangleright\ \Gamma_1, \ldots, \Gamma_m$, results in

$$\vdash^{w}_{\text{pat}} w_i : T_i \ \triangleright\ \Gamma_i \tag{B.14c}$$

Applying the inductive hypothesis m times to eqs. (B.14a–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 14. *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.*

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

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 13, we get

$$\Gamma^1 = x^1_1 : T^1_1, \ \ldots, \ x^1_m : T^1_m \tag{B.15a}$$

$$\emptyset \vdash_{\text{exp}} v^1_i : T^1_i \text{ for } i \in 1..m \tag{B.15b}$$

Generalising for $j \in 1..n$, then $\widetilde{\Gamma} = \Gamma^1, \ldots, \Gamma^n$ holds by generalising eq. (B.15a). Also, $\emptyset \vdash_{\text{exp}} v'_i : T_i$ for $i \in 1..k$ holds by eq. (B.15b). Thus, Corollary 14 holds by applying Lemma 13 n times. □

Lemma 15 shows that the type of expressions remains unchanged (or preserved) after an expression is reduced. This means that expressions have a constant type in all steps of reductions, until the expression cannot be reduced further.

Lemma 15 *(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

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 1..n \tag{B.16a}$$

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

$$e_k \to e'_k \tag{B.16b}$$

Applying eqs. (B.16a) and (B.16b) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_k : T_k$. By the latter, eq. (B.16a) 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

$$\emptyset \vdash_{\mathrm{exp}} e_1 : \mathsf{number} \tag{B.17a}$$

$$\emptyset \vdash_{\mathrm{exp}} e_2 : \mathsf{number} \tag{B.17b}$$

$e \to e'$ 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

$$e_1 \to e'_1 \tag{B.17c}$$

Applying eqs. (B.17a) and (B.17c) to the inductive hypothesis results in $\emptyset \vdash_{\mathrm{exp}} e'_1 : \mathsf{number}$. Using this information, along with eq. (B.17b) in [tArithmetic], results in $\emptyset \vdash_{\mathrm{exp}} e' : \mathsf{number}$, 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 $\emptyset \vdash_{\mathrm{exp}} e : T$, or $\emptyset \vdash_{\mathrm{exp}} v_1 \diamond v_2 : T$, then $\emptyset \vdash_{\mathrm{exp}} e' : T$ follows immediately given that $e' = v = v_1 \diamond v_2$.

Regarding the remaining cases: Cases [tBasic], [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]. □

Lemmata 10–15 allow us to prove the Session Fidelity Theorem. This is one of the main results of Section 5.

Theorem 2 *(Session Fidelity). If* $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \triangleright t : T \lhd S'$ *and* $t \xrightarrow[\Sigma]{\alpha} t'$

- *for* $\alpha = ?\{\mathtt{:l}, \widetilde{v}\}$ *and some session type* $S'' = \textbf{\textit{after}}(S, \alpha)$*, then there exists some* Δ'*, such that* $\Delta' \cdot \emptyset \vdash^{w}_{\Sigma} S'' \triangleright t' : T \lhd S'$ *and* $\textbf{\textit{after}}(\Delta, \alpha, S) = \Delta'$
- *for* $\alpha \in \{f/n, \tau, \iota!\{\mathtt{:l}, \widetilde{v}\}\}$*, then there exists some* S'' *and* Δ'*, such that* $\Delta' \cdot \emptyset \vdash^{w}_{\Sigma} S'' \triangleright t' : T \lhd S'$ *for* $\textbf{\textit{after}}(S, \alpha) = S''$ *and* $\textbf{\textit{after}}(\Delta, \alpha, S) = \Delta'$

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

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

$$t = (x = t_1;\ t_2) \tag{B.18a}$$

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright t_1 : T' \lhd S''' \tag{B.18b}$$

$$\Delta \cdot (x : T') \vdash^{w} S''' \triangleright t_2 : T \lhd S' \tag{B.18c}$$

From the structure of t (eq. (B.18a)), term transitions ($t \xrightarrow{\alpha} t'$) 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

$$t_1 \xrightarrow{\alpha} t'_1 \tag{B.18d}$$

If $\alpha = ?\{\mathtt{:l}, \widetilde{v}\}$, then by α, eqs. (B.18b) and (B.18d) and the inductive hypothesis we obtain

$$\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright t'_1 : T' \lhd S''' \tag{B.18e}$$

where $\textbf{after}(\Delta, \alpha, S) = \Delta'$. In case that $\alpha \neq ?\{\mathtt{:l}, \widetilde{v}\}$, eq. (B.18e) can be obtained as well, however we also require $\textbf{after}(S, \alpha) = S''$, which holds immediately. 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. (B.18c) to get

$$\Delta' \cdot (x : T') \vdash^{w} S''' \triangleright t_2 : T \lhd S' \tag{B.18f}$$

Using eqs. (B.18e) and (B.18f) as the premise for rule [tLet], we obtain:

$$[\text{tLet}]\ \frac{\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright t'_1 : T' \lhd S''' \qquad \Delta' \cdot (x : T') \vdash^{w} S''' \triangleright t_2 : T \lhd S' \qquad x \neq w}{\Delta' \cdot \emptyset \vdash^{w} S'' \triangleright x = t'_1;\ t_2 : T \lhd S'}$$

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 $\alpha = \tau$, therefore $\mathbf{after}(S, \alpha) = S = S'''$. Since $t_1 = v$, by eq. (B.18b) and [tExpression], then $\emptyset \vdash_{\text{exp}} v : T'$ holds. If we apply this latter information and eq. (B.18c) 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, $\Delta' = \Delta$.

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

$$S = \&\left\{?\mathtt{l}_i\left(T_i^1,\ \ldots,\ T_i^n\right).S_i\right\}_{i\in I} \tag{B.19a}$$

$$t = \mathtt{receive\ do}\left(\left\{:\mathtt{l}_i, p_i^1,\ \ldots,\ p_i^n\right\} \to t_i\right)_{i\in I}\mathtt{end} \tag{B.19b}$$

$$S_\mu = \mathbf{after}(\&\left\{?\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i\in I}, \alpha) \tag{B.19c}$$

Since α has to be an incoming message, we focus solely on te first case of the Session Fidelity Theorem. From the premise, we also know that some properties regarding each individual branch from the `receive` construct:

$$\forall i \in I \cdot \begin{cases} \vdash^{w}_{\text{pat}} p_i^j : T_i^j \ \rhd\ \Gamma_i^j \ \text{ for all } j \in 1..n & \text{(B.19d)} \\ \Delta \cdot \left(\Gamma_i^1,\ \ldots,\ \Gamma_i^n\right) \vdash^{w} S_i \rhd t_i : T \lhd S' & \text{(B.19e)} \end{cases}$$

From the structure of t (eq. (B.19b)), 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

$$\alpha = ?\left\{:\mathtt{l}_\mu,\ v_1,\ \ldots,\ v_n\right\} \tag{B.19f}$$

$$\mathbf{match}((p_\mu^1,\ \ldots,\ p_\mu^n), (v_1,\ \ldots,\ v_n)) = [v'_1, \ldots, v'_k / x_1, \ldots, x_k] \tag{B.19g}$$

$$t' = t_\mu\,[v'_1, \ldots, v'_k / x_1, \ldots, x_k]$$

From eq. (B.19f), α refers to the message received from the dual process, which is assumed (from the premise of the theorem) to be valid by the **after** function (eq. (B.19c)). We can compare the contents of this message to the original session type S (eq. (B.19a)), 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 $?\mathtt{l}_\mu(\boldsymbol{T_\mu^1}, \ldots, \boldsymbol{T_\mu^n}).S_\mu$. Formally, this can be written as

$$\emptyset \vdash_{\text{exp}} v_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{B.19h}$$

Applying eqs. (B.20), (B.19g) and (B.19h) into Corollary 14, results in $\widetilde{\Gamma_\mu} = \Gamma_\mu^1,\ \ldots,\ \Gamma_\mu^n = x_1 : T_1,\ \ldots,\ x_k : T_k$ and

$$\emptyset \vdash_{\text{exp}} v'_m : T_m \ \text{ for } m \in 1..k \tag{B.19i}$$

Applying eq. (B.19i) and $\Delta \cdot \widetilde{\Gamma_\mu} \vdash^{w} S_\mu \rhd t_\mu : T \lhd S'$ (from eq. (B.19e) for $i = \mu$) repeatedly to the Substitution Lemma, we get

$$\Delta \cdot \emptyset \vdash^{w} S_\mu \rhd t_\mu\,[v'_1, \ldots, v'_k / x_1, \ldots, x_k] : T \lhd S' \tag{B.19j}$$

Since $\mathbf{after}(\Delta, \alpha, S) = \Delta$, then eq. (B.19j) is the expected result.

[**tChoice**] From the rule, we know that for some $\mu \in I$, $T = \{\mathsf{atom}, T_\mu^1,\ \ldots,\ T_\mu^n\}$ and

$$S = \oplus\left\{!\mathtt{l}_i\left(\widetilde{T_i}\right).S_i\right\}_{i\in I} \tag{B.20a}$$

$$t = \mathtt{send}\left(\iota, \left\{:\mathtt{l}_\mu, e_1,\ \ldots,\ e_n\right\}\right) \tag{B.20b}$$

$$\emptyset \vdash_{\text{exp}} e_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{B.20c}$$

From the structure of t (eq. (B.20b)), 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

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

$$e_k \to e'_k \tag{B.20d}$$

Applying eq. (B.20c) (for $j = k$) and eq. (B.20d) to the Preservation (Expressions) Lemma, we get $\emptyset \vdash_{\text{exp}} e'_k : T_k$. Applying this and eq. (B.20c) to [TCHOICE] results in $\Delta \cdot \emptyset \vdash^{w} S \triangleright t' : T \lhd S_\mu$. Since $\mathbf{after}(S, \tau) = S$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$, this holds.

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

$$t' = \{:\mathtt{l}_\mu, v_1, \ldots, v_n\}$$
$$\alpha = \iota!\{:\mathtt{l}_\mu, v_1, \ldots, v_n\} \tag{B.20e}$$

where α (eq. (B.20e)) is the message being sent to the dual process with *pid* ι – so, we only consider the second case of the Session Fidelity Theorem.

Recall eq. (B.20c), where we have $\emptyset \vdash_{\text{exp}} e_j : \boldsymbol{T^j_\mu}$ for $j \in 1..n$. Notice, that the types $\boldsymbol{T^j_\mu}$ were obtained from the session type S (eq. (B.20a)), 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

$$\emptyset \vdash_{\text{exp}} v_j : T^j_\mu \text{ for all } j \in 1..n \tag{B.20f}$$

By the Value Typing Lemma, we also know that $\emptyset \vdash_{\text{exp}} :\mathtt{l}_\mu : \text{atom}$. Using this latter information and eq. (B.20f) in [TTUPLE] and [TEXPRESSION], we get the required result:

$$\dfrac{\dfrac{\emptyset \vdash_{\text{exp}} :\mathtt{l}_\mu : \text{atom} \qquad \forall j \in 1..n \qquad \emptyset \vdash_{\text{exp}} v_j : T^j_\mu}{\emptyset \vdash_{\text{exp}} \{:\mathtt{l}_\mu, v_1, \ldots, v_n\} : \{\text{atom}, T^1_\mu, \ldots, T^n_\mu\}}\,[\text{TTUPLE}]}{\Delta \cdot \emptyset \vdash^{y} S_\mu \triangleright \{:\mathtt{l}_\mu, v_1, \ldots, v_n\} : T \lhd S_\mu}\,[\text{TEXPRESSION}] \tag{B.20g}$$

Result from eq. (B.20g) holds as required, since $\mathbf{after}(S, \alpha) = S_\mu$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$.

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

$$t = f(w, e_2, \ldots, e_n) \tag{B.21a}$$
$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{B.21b}$$

From the structure of t (eq. (B.21a)), term transitions ($t \xrightarrow{\alpha} t'$) 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

$$t' = f(v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n)$$
$$e_k \to e'_k \tag{B.21c}$$

Applying eq. (B.21b) (for $i = k$) and eq. (B.21c) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e'_k : T_k \tag{B.21d}$$

By eqs. (B.21b) and (B.21d) and [TKNOWNCALL], we get

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright f(v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n) : T \lhd S' \tag{B.21e}$$

eq. (B.21e) holds since $v_1 = w$.

2. [**RCALL$_2$**] From the rule, we know that $\alpha = f/n$, $w = \iota$ and

$$t = f(\iota, v_2, \ldots, v_n) \tag{B.21f}$$
$$t' = \bar{t}\,[\iota/y]\,[v_2, \ldots, v_n/x_2, \ldots, x_n]$$
$$\Sigma(f/n) = \Omega \text{ where } \begin{cases} \Omega.\mathtt{return\_type} = T \\ \Omega.\mathtt{param\_types} = T_2, \ldots, T_n \end{cases} \tag{B.21g}$$
$$\Delta(f/n) = S \tag{B.21h}$$

Since all *known* functions (*i.e.,* $f/n \in \mathbf{dom}(\Delta)$) by eq. (B.21h)) are already typechecked once before, then from the function information environment (*i.e.,* Σ) and eq. (B.21g), we can assume that

$$\Delta \cdot \Gamma' \vdash^{y} S \triangleright \bar{t} : T \lhd \text{end} \tag{B.21i}$$

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. (B.21i) 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 $f/n \in \mathbf{dom}(\Sigma)$, we have

$$\mathbf{fv}\big(\Sigma(f/n).\mathtt{body}\big) \setminus \big(\Sigma(f/n).\mathtt{params} \cup \{\Sigma(f/n).\mathtt{dual}\}\big) = \emptyset$$

By eq. (B.21f) and Value Typing Lemma we know that $\emptyset \vdash_{\mathsf{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (B.21i) to the Substitution Lemma results in

$$\Delta \cdot (x_2 : T_2, \ldots, x_n : T_n) \vdash^{y[\iota/y]} S \triangleright \bar{t}\,[\iota/y] : T \lhd \mathsf{end} \tag{B.21j}$$

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

Applying the Substitution Lemma multiple times to eqs. (B.22b) and (B.21j), results in

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright \bar{t}\,[\iota/y]\,[v_2, \ldots, v_n/x_2, \ldots, x_n] : T \lhd \mathsf{end} \tag{B.21k}$$

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. (B.21h), f/n is already mapped to S in the session typing environment, therefore $(\Delta, f/n : S) = \Delta$, as needed.

[**tUnknownCall**] From the rule, we know

$$t = f\,(w, e_2, \ldots, e_n) \tag{B.22a}$$

$$\emptyset \vdash_{\mathsf{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{B.22b}$$

From the premise we also know that

$$(\Delta, f/n : S) \cdot \big(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\big) \vdash^{y} S \triangleright \bar{t} : T \lhd S' \quad \text{where } \widetilde{x}, \widetilde{T}, \bar{t}, T \text{ and } y \text{ are obtained from the function information environment } (\textit{i.e.}, \Sigma) \tag{B.22c}$$

From the structure of t (eq. (B.22a)), term transitions ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

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

$$t' = f\,\big(v_1, \ldots, v_{k-1}, e'_k, \ldots, e_n\big)$$
$$e_k \to e'_k \tag{B.22d}$$

Applying eq. (B.22b) (for $i = j$) and eq. (B.22d) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\mathsf{exp}} e'_j : T_j \tag{B.22e}$$

Using eq. (B.22b) and eq. (B.22e) in the rule [tUnknownCall], results in

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

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

$$t = f\,(\iota, v_2, \ldots, v_n) \tag{B.22f}$$
$$w = \iota \tag{B.22g}$$
$$t' = \bar{t}\,[\iota/y]\,[v_2, \ldots, v_n/x_2, \ldots, x_n]$$

By eq. (B.22f) and the Value Typing Lemma we know that $\emptyset \vdash_{\mathsf{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (B.22c) to the Substitution Lemma results in

$$(\Delta, f/n : S) \cdot \big(\widetilde{x} : \widetilde{T}\big) \vdash^{y[\iota/y]} S \triangleright \bar{t}\,[\iota/y] : T \lhd S' \tag{B.22h}$$

where by the Variable Substitution Definition and eq. (B.22g), $y\,[\iota/y] = \iota = w$.

Applying the Substitution Lemma repeatedly to eqs. (B.22b) and (B.22h), results in

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

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,

$$t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i\in I}\ (x_{\text{all}} \to t_{\text{all}})\ \texttt{end} \tag{B.23a}$$
$$\Delta \cdot (x_{\text{all}} : U) \vdash^{w} S \rhd t_{\text{all}} : T \lhd S' \tag{B.23b}$$
$$\emptyset \vdash_{\text{exp}} e : U \tag{B.23c}$$
$$\vdash^{w}_{\text{pat}} p_i : U \ \rhd\ \Gamma_i' \qquad \text{for all } i \in I \tag{B.23d}$$
$$\Delta \cdot \Gamma_i' \vdash^{w} S \rhd t_i : T \lhd S' \qquad \text{for all } i \in I \tag{B.23e}$$

By eq. (B.23a), 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 $t' = \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i\in I}\ (x_{\text{all}} \to t_{\text{all}})\ \texttt{end}$. From the premise we know that

   $$e \to e' \tag{B.23f}$$

   By eqs. (B.23c) and (B.23f) and the Preservation (Expressions) Lemma, we get

   $$\emptyset \vdash_{\text{exp}} e' : U \tag{B.23g}$$

   Using eqs. (B.23b), (B.23d), (B.23e), (B.23g), and [tCase], we get

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

   which holds as expected given that $\textbf{after}(S, \tau) = S$ and $\textbf{after}(\Delta, \tau, S) = \Delta$.
2. [**rCase$_2$**] From the rule, we know that $t = \texttt{case}\ v\ \texttt{do}\ (p_i \to t_i)_{i\in I}\ (x_{\text{all}} \to t_{\text{all}})\ \texttt{end}$ and $e = v$. For convenience, we include the catch-all pattern in t with the remaining patterns, so $t = \texttt{case}\ v\ \texttt{do}\ (p_i \to t_i)_{i\in I'}\texttt{end}$. So, for some $j \in I'$,

   $$\textbf{match}(p_j, v) = \sigma \text{ where } \sigma = [^{v_1, \ldots, v_n}/_{x_1, \ldots, x_n}] \tag{B.23h}$$
   $$t' = t_j\sigma \tag{B.23i}$$

   By eqs. (B.24c), (B.23d) and (B.23h) and Lemma 13, we know that $\Gamma_j' = x_1 : T_1, \ \ldots, \ x_n : T_n$ and

   $$\emptyset \vdash_{\text{exp}} v_k : T_k \text{ for all } k \in 1..n \tag{B.23j}$$

   Then, by repeatedly applying the Substitution Lemma to eq. (B.23j), (B.23e for $i = j$), we get

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

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

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

$$t = e \tag{B.24a}$$
$$\emptyset \vdash_{\text{exp}} e : T \tag{B.24b}$$

From the structure of t (eq. (B.24a)), term transition ($t \xrightarrow{\alpha} t'$) can only be derived using [rExpression], resulting in an internal transition (*i.e.*, $\alpha = \tau$)

$$e \to e' \tag{B.24c}$$

By Lemma 15 and eqs. (B.24b) and (B.24c), we know that

$$\emptyset \vdash_{\text{exp}} e' : T \tag{B.24d}$$

Thus, case holds by [tExpression] and eq. (B.24d).

B.3. Proofs for Theorem 4

In this section, we provide the proofs of Proposition 3 which lead to the proofs for *progress* of expressions (Lemma 16) and terms (Theorem 4).

Proposition 3. *Any well-typed term* ($\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$) *following the branching session type is able to process any* valid *incoming message, i.e.,*

$$\left.\begin{array}{r} t = \texttt{receive do}\left(\left\{:l_i, p_i^1, \ldots, p_i^n\right\} \to t_i\right)_{i\in I}\texttt{end} \\ S = \&\left\{?l_i\left(T_i^1, \ldots, T_i^n\right).S_i\right\}_{i\in I} \\ \textbf{\textit{after}}(S, ?\left\{l_k, v^1, \ldots, v^n\right\}) = S_k \text{ for some } k \in I \end{array}\right\} \implies \textbf{\textit{match}}(p_k^j, v^j)_{i\in 1..n} \text{ is defined}$$

Proof. A well-typed term following the branching session type ($\&\{\ldots\}$), has to be precisely a `receive` construct, as shown by term t. Such term can only be typed using the [tBranch] rule, from which we infer that

$$\textbf{simplepat}(p_i^j, T_i^j)_{i\in I, j\in 1..n}$$

The **simplepat** statement indicates that all of the patterns used in the branches are simple ones, *i.e.,* p_i^j has the form of a variable (x_i^j) or a tuple of variables (x_i^j). From the third statement, we know that a message is valid if it matches with one of the labels available in t (*i.e.,* l_k for some $k \in I$) and each payload value ($v^1, \ldots, v^n$) matches with the payload types dictated by the session type S. Therefore we can apply x_i^j and v^j (for all $j \in 1..n$) to the **match** function, to obtain a series of variable substitutions, as required. □

Lemma 16 *(Progress (Expressions)). If* $\emptyset \vdash_{exp} e : T$, *then either* e *is a value, or else there exists some* e' *such that* $e \to e'$

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

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

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 1..n \tag{B.25}$$

By the inductive hypothesis and eq. (B.25), expression e_i can either be a value or reduce to e_i'. We consider both sub-cases:

1. If for any $i \in 1..n$, $e_i \to e_i'$, then case holds by [reTuple], resulting in the reduction $e \to \{v_1, \ldots, v_{i-1}, e_i', \ldots, e_n\}$.
2. Otherwise, if for all $i \in 1..n$, $e_i = v_i$, then case holds since the tuple $\{v_1, \ldots, v_n\}$ is a value.

[tArithmetic] From the rule we know that $e = e_1 \diamond e_2$ and

$$\emptyset \vdash_{\text{exp}} e_1 : \mathsf{number} \tag{B.26a}$$

$$\emptyset \vdash_{\text{exp}} e_2 : \mathsf{number} \tag{B.26b}$$

By the inductive hypothesis and eq. (B.26a), e_1 can either be a value or reduce to e_1'. We consider both sub-cases:

1. If $e_1 \to e_1'$, then case holds by [reOperation$_1$], resulting in the reduction $e \to e_1' \diamond e_2$.
2. Otherwise, if $e_1 = v_1$, we have to consider the behaviour of e_2, which by the inductive hypothesis and eq. (B.26b) results in another two sub-cases:

   2a. If $e_2 \to e_2'$, then case holds by [reOperation$_2$], resulting in the reduction $e \to v_1 \diamond e_2'$.
   2b. Otherwise, if e_2 is a value, then case holds by [reOperation$_3$].

Regarding the remaining cases: Cases [tBoolean], [tComparison], [tList] and [tNot] are analogous to the previous case. Finally, cases [tBasic], [tEList] hold immediately since e in these instances is a value. □

Theorem 4 *(Progress). If* $\Delta \cdot \emptyset \vdash^{w}_{\Sigma} S \triangleright t : T \triangleleft S'$, *then either* t *is a value, or else there exists some term* t' *and action* α *such that* $t \xrightarrow[\Sigma]{\alpha} t'$ *and* $\textbf{\textit{after}}(S, \alpha)$ *is defined*

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

[TLET] From the rule, we know that $t = (x = t_1;\ t_2)$, and

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright t_1 : T' \triangleleft S''' \tag{B.27}$$

By the inductive hypothesis and eq. (B.27), we know that $\mathbf{after}(S, \alpha)$ is defined and that the term t_1 can be a value, or else transition to a new term t_1'. We consider both sub-cases:

1. If $t_1 \xrightarrow{\alpha} t_1'$, then case holds by [RLET$_1$], resulting in $t \xrightarrow{\alpha} t_1'; t_2$.
2. Otherwise, if $t_1 = v_1$, then case holds by [RLET$_2$], resulting in the internal transition $t \xrightarrow{\tau} t_2\,[{}^{v_2}/_{x}]$.

In both cases, $\mathbf{after}(S, \alpha)$ remains defined from the inductive hypothesis.

[TBRANCH] From the rule, we know that

$$t = \texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p}_i\} \to t_i)_{i \in I}\ \texttt{end} \tag{B.28}$$

$$S = \&\left\{?\texttt{l}_i(\widetilde{T}_i).S_i\right\}_{i \in I} \tag{B.29}$$

A `receive` term (eq. (B.28)) transitions to a new subterm (t_i) when a message ($\alpha = ?\{\texttt{:l}_j, v_1, \ldots, v_n\}$) is received. An incoming message is guaranteed to match with one of the available patterns, since (i) we are dealing with a trusted observer which sends valid actions, and (ii) the branches are restricted to simple patterns (using **simplepat**) so a well structured message will match with one of the branches. An incoming message is matched to a branch using its label $\texttt{:l}_j$ (for $j \in I$). So by [RBRANCH], we know that $t \xrightarrow{?\{\texttt{:l}_j, v_1, \ldots, v_n\}} t_j\sigma$, where σ is computed using the Pattern Matching Definition (*i.e.,* $\sigma = \mathbf{match}(\widetilde{p}_j, v_1, \ldots, v_n)$). By the incoming message action and S from eq. (B.29), $\mathbf{after}(S, ?\{\texttt{:l}_j, \widetilde{v}\})$ is defined, as required.

[TCHOICE] From the rule, we know that for some $\mu \in I$, then

$$t = \texttt{send}\left(\iota, \left\{\texttt{:l}_\mu, e_1, \ldots, e_n\right\}\right)$$

$$\emptyset \vdash_{\text{exp}} e_j : T_\mu^j \quad \text{for all } j \in 1..n \tag{B.30}$$

$$S = \oplus\left\{!\texttt{l}_i(\widetilde{T}_i).S_i\right\}_{i \in I} \tag{B.31}$$

By Lemma 16 and eq. (B.30), any expression e_i can be either a value, or else reduce to a new expression, so we consider both sub-cases:

1. If, for any $i \in 1..n$, expression e_i can be reduced (*i.e.,* $e_i \to e_i'$), then case holds by [RCHOICE$_1$], resulting in the internal transition:
$$t \xrightarrow{\tau} \texttt{send}\left(\iota, \left\{\texttt{:l}_\mu, v_1, \ldots, v_{i-1}, e_i', \ldots, e_n\right\}\right)$$
Since this is an internal transition, $\mathbf{after}(S, \tau)$ is defined for any S.
2. Otherwise, if for all $i \in 1..n$, $e_i = v_i$, then case holds by [RCHOICE$_2$], resulting in the transition $t \xrightarrow{\iota!\{\texttt{:l}, v_1, \ldots, v_n\}} \{\texttt{:l}, v_1, \ldots, v_n\}$. In this case, a message containing $\{\texttt{:l}, v_1, \ldots, v_n\}$ is sent to process with *pid* ι. From eq. (B.31), we know that S is an internal choice, thus $\mathbf{after}(S, !\{\texttt{:l}, \widetilde{v}\})$ is defined, as required.

[TKNOWNCALL] From the rule, we know that

$$t = f\ (\iota, e_2, \ldots, e_n)$$

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{B.32}$$

By Lemma 16 and eq. (B.32), any expression e_i can be either a value, or else reduce to a new expression, so we consider both sub-cases:

1. If, for any $i \in 2..n$, expression e_i can be reduced (*i.e.,* $e_i \to e_i'$), then case holds by [RCALL$_1$], resulting in the internal transition: $t \xrightarrow{\tau} f\ (\iota, v_2, \ldots, v_{i-1}, e_i', \ldots, e_n)$.
2. Otherwise, if for all $i \in 2..n$, $e_i = v_i$, then case holds by [RCALL$_2$], resulting in the transition $t \xrightarrow{f/n} t'\,[{}^{\iota}/_{y}]\,[{}^{v_2, \ldots, v_n}/_{x_2, \ldots, x_n}]$. The callee's function body (*i.e.,* t') and the substitution details (*e.g.* y, x_2) are obtained directly from the function information environment Σ using the transition action f/n, *e.g.* $\Sigma\,(f/n)\,.\texttt{body} = t'$.

For both cases, $\mathbf{after}(S, \alpha)$ is defined since **after** accepts and session type S for internal actions (f/n and τ).

[TUNKNOWNCALL] Analogous to previous case.

[TCASE] From the rule, we know that

$$t = \texttt{case}\ e\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}$$
$$\emptyset \vdash_{\text{exp}} e : U \tag{B.33}$$

By Lemma 16 and eq. (B.33), expression e can be either a value, or reduce to a new expression. Consider both sub-cases:

1. If expression e can be reduced (*i.e.,* $e \to e'$), then case holds by [RCASE$_1$], resulting in the internal transition: $t \xrightarrow{\tau} \texttt{case}\ e'\ \texttt{do}\ (p_i \to t_i)_{i\in I}\texttt{end}$.
2. Otherwise, if $e = v$, then case holds by [RCASE$_2$], resulting in the internal transition $t \xrightarrow{\tau} t_j\sigma$, where t_j is the branch which pattern matches with v, and σ contains the substitutions (*i.e.,* $\sigma = \textbf{match}(p_j, v)$).

[TEXPRESSION] From the rule, we know that

$$t = e \tag{B.34a}$$
$$\emptyset \vdash_{\text{exp}} e : T \tag{B.34b}$$

Similar to the previous case, by Lemma 16 and eq. (B.34b), expression e can be either a value, or reduce to a new expression. In case of the former, where e can be reduced (*i.e.,* $e \to e'$), then case holds immediately by [REXPRESSION] (*i.e.,* $t \xrightarrow{\tau} e'$), where $\textbf{after}(S, \tau)$ is defined. In the other case, when e is a value, then t is also a value, since $t = e$ (by eq. (B.34a)). □

References

[1] D. Thomas, Programming Elixir: Functional, Concurrent, Pragmatic, Fun, Pragmatic Bookshelf, 2018.
[2] C. Hewitt, P.B. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: N.J. Nilsson (Ed.), Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Standford, CA, USA, August 20-23, 1973, William Kaufmann, 1973, pp. 235–245.
[3] G.A. Agha, ACTORS - a Model of Concurrent Computation in Distributed Systems, MIT Press Series in Artificial Intelligence, MIT Press, 1990.
[4] J.L. Andersen, etorrent, GitHub repository, https://github.com/jlouis/etorrent, 2013.
[5] ninenines, Cowboy, GitHub repository, https://github.com/ninenines/cowboy, 2022.
[6] D.C. Schmidt, S. Vinoski, Object Interconnections Comparing Alternative Programming Techniques for Multi-Threaded CORBa Servers (Column 7), 1996.
[7] G. Tabone, A. Francalanza, Session types in Elixir, in: E. Castegren, J.D. Koster, S. Fowler (Eds.), 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, ACM, 2021, pp. 12–23.
[8] G. Tabone, A. Francalanza, Session fidelity for ElixirST: a session-based type system for Elixir modules, in: C. Aubert, C. Di Giusto, L. Safina, A. Scalas (Eds.), Proceedings 15th Interaction and Concurrency Experience, ICE 2022, Lucca, Italy, 17 June 2022, in: EPTCS, vol. 365, 2022, pp. 17–36.
[9] S. Nyström, A soft-typing system for Erlang, in: B. Däcker, T. Arts (Eds.), Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003, ACM, 2003, pp. 56–71.
[10] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[11] T. Lindahl, K. Sagonas, Practical type inference based on success typings, in: A. Bossi, M.J. Maher (Eds.), Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy, ACM, 2006, pp. 167–178.
[12] R. Atkey, Parameterised notions of computation, J. Funct. Program. 19 (2009) 335–376, https://doi.org/10.1017/S095679680900728X.
[13] M. Cassola, A. Talagorria, A. Pardo, M. Viera, A gradual type system for Elixir, J. Comput. Lang. 68 (2022) 101077, https://doi.org/10.1016/j.cola.2021.101077.
[14] R.M. Keller, Formal verification of parallel programs, Commun. ACM 19 (1976) 371–384, https://doi.org/10.1145/360248.360251.
[15] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: CONCUR 2010 - Concurrency Theory, 21st International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings, 2010, pp. 222–236.
[16] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, J. ACM 63 (2016) 9, https://doi.org/10.1145/2827695.
[17] C. MacCord, Metaprogramming Elixir - Write Less Code, Get More Done (and Have Fun!), The Pragmatic Programmers, O'Reilly, 2015, http://www.oreilly.de/catalog/9781680500417/index.html.
[18] K. Pruiksma, F. Pfenning, Back to futures, J. Funct. Program. 32 (2022) e6, https://doi.org/10.1017/S0956796822000016.
[19] D. Mostrous, V.T. Vasconcelos, Session typing for a featherweight Erlang, in: W.D. Meuter, G. Roman (Eds.), Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings, in: Lecture Notes in Computer Science, vol. 6721, Springer, 2011, pp. 95–109.
[20] Duffel Technology Ltd, Duffel, https://duffel.com/, 2023.
[21] A. Francalanza, A. Seychell, Synthesising correct concurrent runtime monitors, Form. Methods Syst. Des. 46 (2015) 226–261, https://doi.org/10.1007/s10703-014-0217-9.
[22] A. Francalanza, A theory of monitors, Inf. Comput. 281 (2021) 104704, https://doi.org/10.1016/j.ic.2021.104704.
[23] L. Aceto, I. Cassar, A. Francalanza, A. Ingólfsdóttir, Bidirectional runtime enforcement of first-order branching-time properties, Log. Methods Comput. Sci. 19 (2023), https://doi.org/10.46298/lmcs-19(1:14)2023.
[24] C. Bartolo Burlò, A. Francalanza, A. Scalas, On the monitorability of session types, in theory and practice, in: A. Møller, M. Sridharan (Eds.), 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference), in: LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 20:1–20:30.
[25] M. Cassola, A. Talagorria, A. Pardo, M. Viera, A gradual type system for Elixir, in: Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity, 2020.
[26] Erlang Solutions, Gradient, GitHub repository, https://github.com/esl/gradient, 2022.

[27] W.-M. Wijnja, Typecheck: fast and flexible runtime type-checking for your Elixir projects, GitHub repository, https://github.com/Qqwy/elixir-type_check, 2022.
[28] J. Harrison, Automatic detection of core Erlang message passing errors, in: N. Chechina, A. Francalanza (Eds.), Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, ICFP 2018, St. Louis, MO, USA, September 23-29, 2018, ACM, 2018, pp. 37–48.
[29] M. Christakis, K. Sagonas, Detection of asynchronous message passing errors using static analysis, in: R. Rocha, J. Launchbury (Eds.), Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings, in: Lecture Notes in Computer Science, vol. 6539, Springer, 2011, pp. 5–18.
[30] J. Harrison, Runtime type safety for Erlang/OTP behaviours, in: A. Francalanza, V. Fördós (Eds.), Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2019, Berlin, Germany, August 18, 2019, ACM, 2019, pp. 36–47.
[31] H. Svensson, L. Fredlund, C.B. Earle, A unified semantics for future Erlang, in: S.L. Fritchie, K. Sagonas (Eds.), Proceedings of the 9th ACM SIGPLAN Workshop on Erlang, Baltimore, Maryland, USA, September 30, 2010, ACM, 2010, pp. 23–32.
[32] R. Neykova, N. Yoshida, Multiparty session actors, Log. Methods Comput. Sci. 13 (2017), https://doi.org/10.23638/LMCS-13(1:17)2017.
[33] K. Honda, A. Mukhamedov, G. Brown, T. Chen, N. Yoshida, Scribbling interactions with a formal foundation, in: R. Natarajan, A.K. Ojo (Eds.), Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings, in: Lecture Notes in Computer Science, vol. 6536, Springer, 2011, pp. 55–75.
[34] S. Fowler, An Erlang implementation of multiparty session actors, in: M. Bartoletti, L. Henrio, S. Knight, H.T. Vieira (Eds.), Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016, in: EPTCS, vol. 223, 2016, pp. 36–50.
[35] R. Neykova, N. Yoshida, Let it recover: multiparty protocol-induced recovery, in: Proceedings of the 26th International Conference on Compiler Construction, CC 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 98–108.
[36] A. Scalas, N. Yoshida, Lightweight session programming in Scala, in: S. Krishnamurthi, B.S. Lerner (Eds.), 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy, in: LIPIcs, vol. 56, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 21:1–21:28.
[37] A. Scalas, N. Yoshida, E. Benussi, Effpi: verified message-passing programs in Dotty, in: J.I. Brachthäuser, S. Ryu, N. Nystrom (Eds.), Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019, ACM, 2019, pp. 27–31.
[38] A. Scalas, N. Yoshida, E. Benussi, Verifying message-passing programs with dependent behavioural types, in: K.S. McKinley, K. Fisher (Eds.), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, ACM, 2019, pp. 502–516.
[39] P. Harvey, S. Fowler, O. Dardha, S.J. Gay, Multiparty session types for safe runtime adaptation in an actor language, in: A. Møller, M. Sridharan (Eds.), 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference), in: LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 10:1–10:30.
[40] T.B.L. Jespersen, P. Munksgaard, K.F. Larsen, Session types for rust, in: P. Bahr, S. Erdweg (Eds.), Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015, ACM, 2015, pp. 13–22.
[41] W. Kokke, Rusty variation: deadlock-free sessions with failure in rust, in: M. Bartoletti, L. Henrio, A. Mavridou, A. Scalas (Eds.), Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20–21 June 2019, in: EPTCS, vol. 304, 2019, pp. 48–60.
[42] N. Lagaillardie, R. Neykova, N. Yoshida, Implementing multiparty session types in rust, in: S. Bliudze, L. Bocchi (Eds.), 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, in: Lecture Notes in Computer Science, vol. 12134, Springer, 2020, pp. 127–136.
[43] Z. Cutner, N. Yoshida, Safe session-based asynchronous coordination in rust, in: F. Damiani, O. Dardha (Eds.), Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings, in: Lecture Notes in Computer Science, vol. 12717, Springer, 2021, pp. 80–89.
[44] L. Padovani, A simple library implementation of binary sessions, J. Funct. Program. 27 (2017) e4, https://doi.org/10.1017/S0956796816000289.
[45] H.C. Melgratti, L. Padovani, Chaperone contracts for higher-order sessions, Proc. ACM Program. Lang. 1 (2017) 35, https://doi.org/10.1145/3110279.
[46] K. Honda, V.T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: Programming Languages and Systems: 7th European Symposium on Programming, ESOP'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 7, Springer, 1998, pp. 122–138.
[47] E. Kamburjan, C.C. Din, T. Chen, Session-based compositional analysis for actor-based languages using futures, in: K. Ogata, M. Lawford, S. Liu (Eds.), Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 10009, 2016, pp. 296–312.
[48] R. Hähnle, A.W. Haubner, E. Kamburjan, Locally static, globally dynamic session types for active objects, in: F.S. de Boer, J. Mauro (Eds.), Recent Developments in the Design and Implementation of Programming Languages, Gabbrielli's Festschrift, November 27, 2020, Bologna, Italy, in: OASIcs, vol. 86, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 1:1–1:24.
[49] U. de'Liguoro, L. Padovani, Mailbox types for unordered interactions, in: T.D. Millstein (Ed.), 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, the Netherlands, in: LIPIcs, vol. 109, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 15:1–15:28.
[50] S. Fowler, D.P. Attard, F. Sowul, S.J. Gay, P. Trinder, Special delivery: programming with mailbox types, Proc. ACM Program. Lang. (2023).
[51] S. Balzer, F. Pfenning, Manifest sharing with session types, Proc. ACM Program. Lang. 1 (2017) 37, https://doi.org/10.1145/3110281.
[52] A. Francalanza, M. Hennessy, A theory for observational fault tolerance, J. Log. Algebraic Methods Program. 73 (2007) 22–50.
[53] A. Francalanza, M. Hennessy, A theory of system behaviour in the presence of node and link failure, Inf. Comput. 206 (2008) 711–759.
[54] L. Bocchi, J. Lange, S. Thompson, A.L. Voinea, A model of actors and grey failures, in: COORDINATION, in: Lecture Notes in Computer Science, vol. 13271, Springer, 2022, pp. 140–158.
[55] K. Peters, U. Nestmann, C. Wagner, Fault-tolerant multiparty session types, in: FORTE, in: Lecture Notes in Computer Science, vol. 13273, Springer, 2022, pp. 93–113.
[56] A.D. Barwell, A. Scalas, N. Yoshida, F. Zhou, Generalised multiparty session types with crash-stop failures, in: CONCUR, in: LIPIcs, vol. 243, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 35:1–35:25.
[57] M.A.L. Brun, O. Dardha, Magπ: types for failure-prone communication, in: ESOP, in: Lecture Notes in Computer Science, vol. 13990, Springer, 2023, pp. 363–391.
[58] A. Das, J. Hoffmann, F. Pfenning, Work analysis with resource-aware session types, in: A. Dawar, E. Grädel (Eds.), Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, ACM, 2018, pp. 305–314.
[59] A. Francalanza, E. de Vries, M. Hennessy, Compositional reasoning for explicit resource management in channel-based concurrency, Log. Methods Comput. Sci. 10 (2014), https://doi.org/10.2168/LMCS-10(2:15)2014.
[60] G. Castagna, G. Duboc, J. Valim, The design principles of the Elixir type system, CoRR, arXiv:2306.06391, 2023.