# Session Fidelity for ElixirST: A Session-Based Type System for Elixir Modules

Gerard Tabone Adrian Francalanza

Computer Science Department
University of Malta
Msida, Malta

gerard.tabone.17@um.edu.mt adrian.francalanza@um.edu.mt

This paper builds on prior work investigating the adaptation of session types to provide behavioural information about Elixir modules. A type system called ElixirST has been constructed to statically determine 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 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 invoked by the module client. This operational semantics is then used to prove session fidelity for ElixirST.

## 1 Introduction

In order to better utilise recent advances in microprocessor design and architecture distribution, modern programming languages offer a variety of abstractions for the construction of concurrent programs. In the case of message-passing 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, program correctness becomes harder to verify and language support for detecting errors at the development stage can substantially decrease the number of concurrency errors.

Elixir [34], based on the actor model [1, 14], is one such example of a modern programming language for concurrency. As depicted in Figure 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, Figure 1 depicts a module $m$ which contains several public functions (*i.e.,* $f_1, \ldots, f_n$) and private functions (*i.e.,* $g_1, \ldots, g_j$). For example, 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 $g_2$ whereas the private function $g_j$ can recursively call itself.

A prevalent Elixir design pattern is that of a server listening for client requests. 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 Figure 1, a handler process running public function $f_1$ is assigned to the session with client $\mathsf{client}_1$ whereas

Preliminary Report. Final version to appear in:
ICE 2022

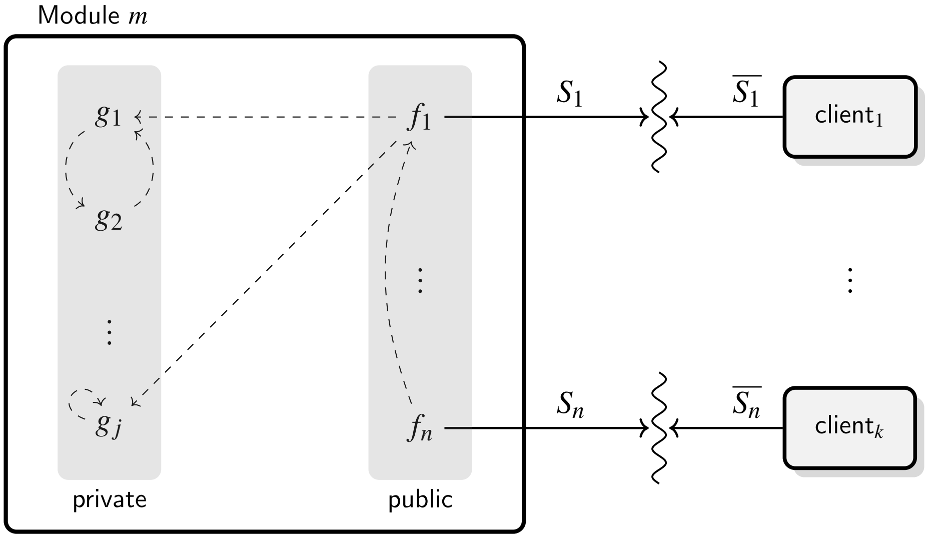


Figure 1: An Elixir module consisting of public and private functions, interacting with client processes

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 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.* two processes waiting forever for one another to send messages of a particular kind when a message of a different kind has been sent instead).

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 Figure 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 support for describing (and verifying) the interaction protocol of a function in terms of its communication side-effects. To this end, in earlier work [32] we devised the tool[1] ElixirST, assisting module construction 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 implements a type-checker that verifies whether the body of a function respects the ascribed session type protocol specification.

**Contribution.** This paper validates the underlying type system on which the ElixirST type-checker is built. More concretely, in Section 3 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. This operational semantics then allows us to prove *session fidelity* for the ElixirST type system in Section 4. In Section 2 we provide the necessary background on the existing session type system from [32] to make the paper self-contained.

[1] ElixirST is available on GitHub: https://github.com/gertab/ElixirST

## 2 Preliminaries

We introduce a core Elixir subset and review the main typing rules for the ElixirST type system [32].

### 2.1 The Actor Model

Elixir uses the actor concurrency model [1, 14]. 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 mechanisms such as pattern matching. 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.

### 2.2 Session Types

The ElixirST type system [32] assumes the standard expression types, including basic types, such as boolean, number, atom and pid, and inductively defined types, such as tuples ($\{T_1,\ \dots,\ T_n\}$) and lists ($[T]$); these already exist in the Elixir language and they are dynamically checked. It extends these with (binary) session types, which are used to statically check the message-passing interactions.

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

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

The *branching* construct, $\&\big\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\big\}_{i\in I}$, 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 $\tilde{T}$ stands for $T^1,\dots,T^k$ 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. In both cases, the labels need to be pairwise distinct. Recursive types are treated equi-recursively [27], and used interchangeably with their unfolded counterparts. For brevity, the symbols & and $\oplus$ are occasionally omitted for singleton options, *e.g.*, $\oplus\big\{!\mathtt{l}(\mathsf{number}).S_1\big\}$ 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 $\overline{S}$.

### 2.3 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*. As an extension to [32], the first parameter

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

Figure 2: Elixir syntax

($y$), is reserved for the *pid* of the dual process. Although a module may contain functions with the same name, their arity must be different.

In our formalisation, Elixir function parameters and return values are assigned a type using the `@spec` annotation, $f(\widetilde{T})\ ::\ T$, describing the parameter types, $\widetilde{T}$, and the return type, $T$. This annotation is already used by Dialyzer for success typing [21]. In addition to this, we decorate public functions with session types, defined in Section 2.2, 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` "$\mathsf{X}$", 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 Figure 2. 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, $\texttt{receive do}\ \big(\{\,\texttt{:l}_i, p_i^1,\ \ldots,\ p_i^n\} \to t_i\big)_{i\in I}\texttt{end}$, 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 Figure 2, 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 $t_i$. 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.

## 2.4 Type System

The session type system from [32] statically verifies that public functions within a module observe the communication protocols ascribed to them. It uses three environments:

$$
\begin{array}{rl}
\text{Variable binding env.} & \Gamma ::= \emptyset \mid \Gamma,\ x : T \\
\text{Session typing env.} & \Delta ::= \emptyset \mid \Delta,\ f/n : S \\
\text{Function inf. env.} & \Sigma ::= \emptyset \mid \Sigma,\ f/n : \left\{ \begin{array}{l} \texttt{params} = \widetilde{x},\ \texttt{param\_types} = \widetilde{T}, \\ \texttt{body} = t,\ \texttt{return\_type} = T,\ \texttt{dual} = y \end{array} \right\}
\end{array}
$$

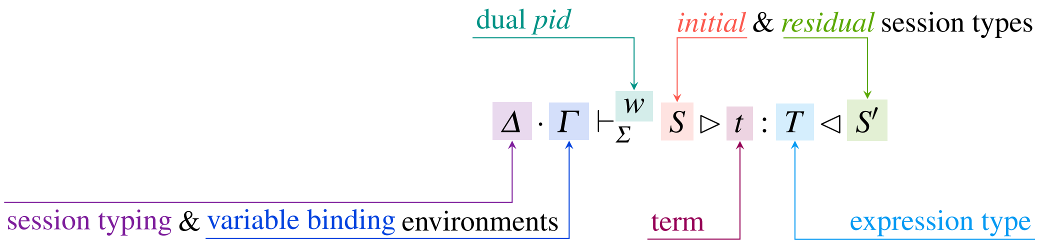
The *variable binding* environment, $\Gamma$, maps (data) variables to basic types ($x : T$). We write $\Gamma, x : T$ to extend $\Gamma$ with the new mapping $x : T$, where $x \notin \mathbf{dom}(\Gamma)$. The *session typing* environment, $\Delta$, maps function names and arity pairs to their session type ($f/n : S$). If a function $f/n$ has a *known* session type, then it can be found in $\Delta$, *i.e.,* $\Delta(f/n) = S$. Each module has a static *function information* environment, $\Sigma$, 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`). In contrast to the original type system from [32], $\Sigma(f/n)$ also returns the variable name that represents the interacting process' *pid*, *i.e.,* the option `dual`. We assume that *function information* environments, $\Sigma$, are *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*:

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

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 **functions**$(\widetilde{D})$ returns a list of all function names (and arity) of the public functions ($\widetilde{D}$) to be checked individually. The function **sessions**$(\widetilde{D})$ obtains a mapping of all the public functions to their expected session types stored in $\Delta$. 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 **details** populates the *function information* environment ($\Sigma$) with details about all the *public* ($\widetilde{D}$) and *private* functions ($\widetilde{P}$) within the module.

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

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


dual *pid*
*initial* & *residual* session types
$\Delta \cdot \Gamma \vdash^{w}_{\Sigma} S \rhd t : T \lhd S'$
session typing & variable binding environments
term
expression type

This judgement states that "the term $t$ can produce a value of type $T$ after following an interaction protocol starting from the initial session type $S$ up to the residual session type $S'$, while interacting with a dual process with *pid* identifier $w$. This typing is valid under some *session typing* environment $\Delta$, *variable binding* environment $\Gamma$ and *function information* environment $\Sigma$." Since the *function information* environment $\Sigma$ is static for the whole module (and by extension, for all sub-terms), it is left implicit in the term typing rules. We consider four main rules, and relegate the rest to Appendix B.1.

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

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. The types within each `receive` branch are computed using the pattern typing judgement, $\vdash^{w}_{\text{pat}} p : T \ \rhd\ \Gamma$, which assigns types to variables present in patterns (see Appendix B.3). Each `receive` branch is then checked w.r.t. the common type $T$ and a common residual session type $S'$.

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

The rule [TCHOICE] typechecks the sending of messages. 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 ($\mathtt{l}_i$) offered by the session choice. The message payloads must also match with the corresponding types associated with the label ($\widetilde{T_i}$ of $\mathtt{l}_i$) stated via the expression typing judgement $\Gamma \vdash_{\text{exp}} e : T$ (see Appendix B.2). The typing rule also checks the *pid* of the addressee of the `send` statement which must match with the dual *pid* ($w$) states in the judgment itself to ensure that messages are only sent to the correct addressee.

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

Since public functions are decorated with a session type explicitly using the `@session` (or `@dual`) annotation, they are listed in $\mathbf{dom}(\Delta)$. Calls to public functions are typechecked using the [TRECKNOWNCALL] 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 [TRECKNOWNCALL] 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.*, $\Delta \cdot \Gamma \vdash^{w} S \rhd f\,(\, w\,, \dots) : T \lhd \mathsf{end}$).

$$\begin{array}{cccc} & \Sigma(f/n)=\Omega & f/n\notin\mathbf{dom}(\Delta) & \Omega.\mathtt{dual}=y \\ \Omega.\mathtt{params}=\widetilde{x} & \Omega.\mathtt{param\_type}=\widetilde{T} & \Omega.\mathtt{body}=t & \Omega.\mathtt{return\_type}=T \end{array}$$

$$[\textsc{tRecUnknownCall}]\ \frac{(\Delta, f/n:S)\cdot\big(\Gamma, y:\mathsf{pid}, \widetilde{x}:\widetilde{T}\big)\vdash^{y} S\rhd t:T\lhd S' \qquad \forall i\in 2..n\cdot\big\{\Gamma\vdash_{\exp} e_i:T_i\big\}}{\Delta\cdot\Gamma\vdash^{w} S\rhd f\,(w,\,e_2,\,\ldots,\,e_n):T\lhd S'}$$

Contrastingly, a call to a (private) function, $f/n$, with an *unknown* session type associated to it is type-checked using the [tRecUnknownCall] rule. As in the other rule, it ensures that parameters have the correct types ($\Gamma\vdash_{\exp} e_i:T_i$). However, it also analyses the function's body $t$ (obtained from $\Sigma$) with respect to the session type $S$ inherited from the initial session type of the call, Furthermore, this session type is appended to the *session typing* environment $\Delta$ 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 [tRecKnownCall] is used thus bypassing the need to re-analyse its body.

### 2.5 Elixir System

The ElixirST provides 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 Figure 3. Initially the actor (pre-server) is spawned, passing its *pid* ($\iota_{server}$) to the second spawned actor (pre-client). Then, pre-client relays back its *pid* ($\iota_{client}$) 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. Figure 3 shows that the server process executes the body $t$, where it has access to the mailbox $\mathscr{M}$. As it executes, messages may be sent or received (shown by the action $\alpha$) and stored in the mailbox $\mathscr{M}'$. The specific working of these transitions is explained in the following section.

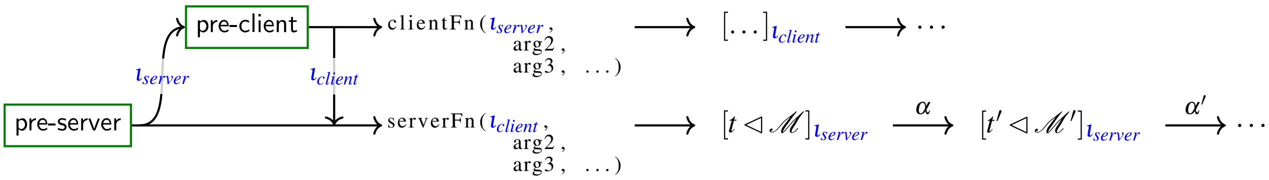

pre-client
clientFn($\iota_{server}$, arg2, arg3, ...)
$[\ldots]_{\iota_{client}}$
$\iota_{server}$
$\iota_{client}$
pre-server
serverFn($\iota_{client}$, arg2, arg3, ...)
$[t\lhd\mathscr{M}]_{\iota_{server}}$
$\alpha$
$[t'\lhd\mathscr{M}']_{\iota_{server}}$
$\alpha'$

Figure 3: Spawning two processes (green boxes represent *spawned* concurrent processes)

## 3 Operational Semantics

We describe the operational semantics of the Elixir language subset of Figure 2 as a *labelled transition system* (LTS) [18] describing how a handler process within a session executes while interacting with the session client, as outlined in Figure 1. The transitions $t\xrightarrow{\alpha}t'$ describes the fact that a handler process in state $t$ performs an execution step to transition to the new state $t'$ while producing action $\alpha$ 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 $\alpha$ can take the following forms:

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

Both output and input actions constitute external actions that affect either party in a session; the type system from Section 2.4 disciplines these external actions. Internal actions, include *silent* transition ($\tau$) and function calls ($f/n$); although the latter may be formalised 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 ($\Sigma$), which contains details about all the functions available in the module. Since $\Sigma$ remains static during transitions, we leave it implicit in the transitions rules.

The transitions are defined by the *term* transition rules listed in Figure 4. 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

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

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

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

$$[\textsc{rBranch}]\ \frac{\exists j \in I \qquad \texttt{l}_j = \texttt{l} \qquad \textbf{match}(\widetilde{p}_j, v_1,\ \ldots,\ v_n) = \sigma}{\texttt{receive do}\ (\{\texttt{:l}_i, \widetilde{p}_i\} \rightarrow t_i)_{i \in I}\texttt{end} \xrightarrow{?\{\texttt{:l},\ v_1,\ \ldots,\ v_n\}} t_j\sigma}$$

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

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

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

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

Figure 4: Term transition semantic rules

$$\boxed{e \to e'}\quad [\textsc{rEOperation}_1]\,\frac{e_1 \to e_1'}{e_1 \,\diamond\, e_2 \to e_1' \diamond e_2} \qquad [\textsc{rEOperation}_2]\,\frac{e_2 \to e_2'}{v_1 \,\diamond\, e_2 \to v_1 \diamond e_2'}$$

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

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

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

Figure 5: Expression reduction rules

part of the message to a value from left to right. This is carried out via rule $[\textsc{rChoice}_1]$ 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 $[\textsc{rChoice}_2]$ 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, which has a *pid* value of $\iota$. 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 $[\textsc{rBranch}]$. 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 $\texttt{:l}$. 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 3.1) produces the substitutions $\sigma$, mapping the matched variables in the pattern $\widetilde{p_j}$ to values from $\widetilde{v}$. This substitution $\sigma$ is then used to instantiate the free variables in continuation branch $t_j$.

**Definition 3.1** *(Pattern Matching)***.** The **match** function pairs patterns with a corresponding value, resulting in a sequence of substitutions (called $\sigma$), *e.g.*, $\mathbf{match}(p,v) = [{}^{v_1}\!/_{x_1}]\,[{}^{v_2}\!/_{x_2}] = [{}^{v_1,v_2}\!/_{x_1,x_2}]$. Note that, a sequence of **match** outputs are combined together, where the empty substitutions (*i.e.,* $[\,]$) are ignored. The match function builds a meta-list of substitutions, which is a different form of lists defined by the Elixir syntax in Figure 2.

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

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

**Example 3.1.** For the pattern $p_1 = \{x, 2, y\}$ and the value tuple $v_1 = \{8, 2, \textit{true}\}$, $\mathbf{match}(p_1, v_1) = \sigma$ where $\sigma = [{}^{8}\!/_{x}]\,[{}^{\textit{true}}\!/_{y}]$ (written also as $\sigma = [{}^{8,\textit{true}}\!/_{x,y}]$). However for pattern $p_2 = \{x, 2, \textit{false}\}$, 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 $[\textsc{rCall}_1]$ from Figure 4, a function call is evaluated by first reducing all of its parameters to a value, using the expression reduction rules in Figure 5; again this models a call-by-value semantics. Once all arguments have been fully reduced, $[\textsc{rCall}_2]$, the implicit environment $\Sigma$ 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 $[\textsc{rCase}_1]$. Then, rule $[\textsc{rCase}_2]$ matches the value with the correct branch, using the **match** function, akin to $[\textsc{rBranch}]$. Whenever a term consists solely of an expression, it silently reduces using $[\textsc{rExpression}]$ using the expression reduction rules $e \rightarrow e'$ of Figure 5. These are fairly standard.

# 4 Session Fidelity

We validate the static properties imposed by the ElixirST type system [32], overviewed in Section 2.4, by establishing a relation with the runtime behaviour of a typechecked Elixir program, using the transition semantics defined in Section 3. Broadly, we establish a form of *type preservation*, which states that if a well-typed term transitions, the resulting term then remains well-typed [27]. 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*. 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.

Before embarking on the proof for session fidelity, we prove an auxiliary proposition that acts as a sanity check for our operational semantics. We note that the operational semantics of Section 3 assumes 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 C.1 for details. □

The statement of the session fidelity property relies on the definition of a partial function called **after** (Definition 4.1), which takes a session type and an action as arguments and returns another session type as a result. This function serves two purposes: (a) the function $\mathbf{after}(S, \alpha)$ is only defined for actions $\alpha$ that are (immediately) permitted by the protocol $S$, which allows us to verify whether a term transition step violated a protocol or not; and (b) 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 $\alpha$ be a permitted action, and this is precisely $S'$, the continuation session type returned where $\mathbf{after}(S, \alpha) = S'$.

**Definition 4.1** *(After Function)***.** The **after** function is partial function defined for the following cases:

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

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

$$\begin{aligned}
\textbf{after}(\Delta,f/n,S) &\stackrel{\text{def}}{=} \Delta, f/n : S \\
\textbf{after}(\Delta,\alpha,S) &\stackrel{\text{def}}{=} \Delta \qquad\qquad \text{if } \alpha\neq f/n
\end{aligned}$$

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 $\iota$, and the function parameter values, say $\widetilde{v}$, respectively, $t\,[{}^{\iota}/_{y}]\,[{}^{\widetilde{v}}/_{\widetilde{x}}]$, as modelled in rule [RCALL$_2$] from Figure 4. 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.,*

$$\Delta\cdot\emptyset\vdash^{w} S\rhd t : T \lhd S' \tag{1}$$

(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 $\alpha$, *i.e.,*

$$t\xrightarrow{\alpha} t' \tag{2}$$

the new term $t'$ remains well-typed, *i.e.,*

$$\Delta'\cdot\emptyset\vdash^{w} S''\rhd t' : T \lhd S' \tag{3}$$

where the evolved $S''$ and $\Delta'$ are computed as $\textbf{after}(S,\alpha)=S''$ and $\textbf{after}(\Delta,\alpha,S)=\Delta'$. This ensures that the base type of the term is preserved (note the constant type $T$ in eqs. (1) and (3)). Furthermore, it ascertains that the term $t$ follows an interaction protocol starting from the initial session type $S$ up to the residual session type $S'$ (eq. (1)), since the updated session type $S''$ is defined for $\textbf{after}(S,\alpha)$.

**Theorem 2** (Session Fidelity)**.** *If $\Delta\cdot\emptyset\vdash^{w}_{\Sigma} S\rhd t : T\lhd S'$ and $t\xrightarrow[\Sigma]{\alpha} t'$, then there exists some $S''$ and $\Delta'$, such that $\Delta'\cdot\emptyset\vdash^{w}_{\Sigma} S''\rhd t' : T\lhd S'$ for **after**$(S,\alpha)=S''$ and **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 C.2. □

$$\begin{tikzcd}
t_1 \arrow[r, "\alpha_1"] \arrow[d, dotted, no head] & t_2 \arrow[r, "\alpha_2"] \arrow[d, dotted, no head] & \cdots \arrow[r, "\alpha_n"] & v \arrow[r, "\not"] \arrow[d, dotted, no head] & {} \\
S_1 & \textbf{after}(S_1, \alpha_1) = S_2 & & \textbf{after}(S_n, \alpha_n) = \mathsf{end} &
\end{tikzcd}$$

Figure 6: Repeated applications of *session fidelity*

As shown in Figure 6, by repeatedly applying Theorem 2, we can therefore conclude that all the (external) actions generated as a result of a computation (*i.e.,* sequence of transition steps) must all be actions that follow the protocol 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.,* $\textbf{after}(S_n, \alpha_n) = \mathsf{end}$. In such a case, we are guaranteed that the term will not produce further side-effects, as in the case of Figure 6 where the term is reduced all the way down to some value, $v$.

**Example 4.1.** We consider a concrete example to show the importance of session fidelity. The function called `pinger/1` is able to send `ping` and receive `pong` repeatedly.

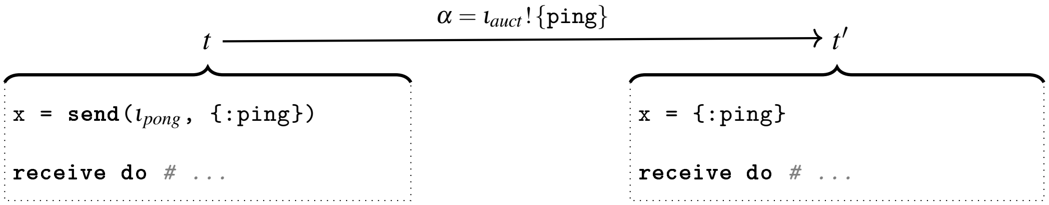
```
@session "X = !ping().?pong().X"
def pinger(pid) do
  x = send(pid, {:ping})

  receive do
    {:pong} -> IO.puts("Received pong.")
  end
  pinger(pid)
end
```

This function adheres to the following protocol:

$$\mathsf{X} = !\mathtt{ping}().?\mathtt{pong}().\mathsf{X}$$

A process evaluating the function `pinger` executes by first sending a message containing a `ping` label to the interacting processes' *pid* ($\iota_{pong}$), as shown below.



```
x = send(ιpong, {:ping})

receive do # ...
```

```
x = {:ping}

receive do # ...
```

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 $\alpha$, where $\alpha = \iota_{pong}!\{\mathtt{ping}\}$. By the After Function Definition, X evolves to a new session type X':

$$\mathsf{X'} = \textbf{after}(!\mathtt{ping}().?\mathtt{pong}().\mathsf{X}, \alpha) = ?\mathtt{pong}().\mathsf{X}$$

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 pong, before recursing. Although the process keeps executing indefinitely, by the session fidelity property, we know that each step of execution will be in line with the original protocol. ■

## 5 Related Work

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

**Type Systems for Elixir** Cassola *et al.* [4, 5] 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* [8]. It is a wrapper for its Erlang counterpart tool and takes a similar approach to [5], where gradual types are used. Another project, *TypeCheck* [35], 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 [5, 8, 35] 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 [11] 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. Harrison [12] extends [11] to add 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, [11, 12] perform automatic analysis of the implementation, however they do not verify communication with respect to a general protocol (*e.g.*, session types).

Another type system for Erlang was presented 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]).

**Session Type Systems.** Closest to our work is [23], 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 takes a more theoretic approach, so there is no implementation for [23]. 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 limited but pragmatic approach, where we introduce session types for functions within a module. Furthermore, 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 [24, 25]. They use the Scribble [15] 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 (akin to our function annotations). Similar to [24, 25], Fowler [9] 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. In contrast, although our work accepts a more limited language, ElixirST provides static guarantees where issues are flagged at pre-deployment stages, rather than flagging them at runtime.

Scalas and Yoshida [29] 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. Linearity checks are performed at runtime, which ensure that an implementation fully exhausts its protocol exactly once. Bartolo Burlò *et al.* [3] extended the aforementioned work [29], to monitor one side of an interaction statically and the other side dynamically using runtime monitors.

Harvey *et al.* [13] 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 [17, 19] and multiparty [6, 20] 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 [26] created a binary session type library for OCaml to provide static communication guarantees. This project was extended [22] 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.

## 6 Conclusion

In this work we established a correspondence between the ElixirST type system [32] 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 $\bar{S}$), they avoid certain communication errors (*e.g.*, a send statement without a corresponding receive construct). An extended version of this work can be found in the technical report [33].

**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, 30] or the body of work on intuitionistic session types organising processes hierarchically [2, 28]. 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 session type extensions that account for failure [13]. Finally, we also plan to augment our session typed protocols to account for resource usage and cost, along the lines of [7, 10].

**Acknowledgements.** We thank Simon Fowler and Matthew Alan Le Brun for their encouragement and suggestions for improvement on earlier versions of the work. 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).

## References

[1] Gul A. Agha (1990): *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, MIT Press.

[2] Stephanie Balzer & Frank Pfenning (2017): *Manifest sharing with session types*. *Proc. ACM Program. Lang.* 1(ICFP), pp. 37:1–37:29, doi:10.1145/3110281.

[3] Christian Bartolo Burlò, Adrian Francalanza & Alceste Scalas (2021): *On the Monitorability of Session Types, in Theory and Practice*. In Anders Møller & Manu Sridharan, editors: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, *LIPIcs* 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 20:1–20:30, doi:10.4230/LIPIcs.ECOOP.2021.20.

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

[5] Mauricio Cassola, Agustín Talagorria, Alberto Pardo & Marcos Viera (2022): *A gradual type system for Elixir*. *Journal of Computer Languages* 68, p. 101077, doi:10.1016/j.cola.2021.101077.

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

[7] Ankush Das, Jan Hoffmann & Frank Pfenning (2018): *Work Analysis with Resource-Aware Session Types*. In Anuj Dawar & Erich Grädel, editors: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, ACM, pp. 305–314, doi:10.1145/3209108.3209146.

[8] Erlang Solutions: *Gradient*. Available at `https://github.com/esl/gradient`.

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

[10] Adrian Francalanza, Edsko de Vries & Matthew Hennessy (2014): *Compositional Reasoning for Explicit Resource Management in Channel-Based Concurrency*. *Log. Methods Comput. Sci.* 10(2), doi:10.2168/LMCS-10(2:15)2014.

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

[12] Joseph Harrison (2019): *Runtime Type Safety for Erlang/OTP Behaviours*. In Adrian Francalanza & Viktória Fördós, editors: *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2019, Berlin, Germany, August 18, 2019*, ACM, pp. 36–47, doi:10.1145/3331542.3342571.

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

[14] Carl Hewitt, Peter Boehler Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *IJCAI*, William Kaufmann, pp. 235–245.

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

[16] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

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

[18] Robert M. Keller (1976): *Formal Verification of Parallel Programs*. *Commun. ACM* 19(7), pp. 371–384, doi:10.1145/360248.360251.

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

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

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

[22] Hernán C. Melgratti & Luca Padovani (2017): *Chaperone Contracts for Higher-Order Sessions*. *Proc. ACM Program. Lang.* 1(ICFP), pp. 35:1–35:29, doi:10.1145/3110279.

[23] Dimitris Mostrous & Vasco Thudichum Vasconcelos (2011): *Session Typing for a Featherweight Erlang*. In Wolfgang De Meuter & Gruia-Catalin Roman, editors: *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland,*

*June 6-9, 2011. Proceedings*, *Lecture Notes in Computer Science* 6721, Springer, pp. 95–109, doi:10.1007/978-3-642-21464-6_7.

[24] Rumyana Neykova & Nobuko Yoshida (2014): *Multiparty Session Actors*. In Alastair F. Donaldson & Vasco T. Vasconcelos, editors: *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014*, *EPTCS* 155, pp. 32–37, doi:10.4204/EPTCS.155.5.

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

[26] Luca Padovani (2017): *A Simple Library Implementation of Binary Sessions*. *J. Funct. Program.* 27, p. e4, doi:10.1017/S0956796816000289.

[27] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[28] Klaas Pruiksma & Frank Pfenning (2022): *Back to futures*. *Journal of Functional Programming* 32, p. e6, doi:10.1017/S0956796822000016.

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

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

[31] Hans Svensson, Lars-Åke Fredlund & Clara Benac Earle (2010): *A unified semantics for future Erlang*. In Scott Lystig Fritchie & Konstantinos Sagonas, editors: *Proceedings of the 9th ACM SIGPLAN workshop on Erlang, Baltimore, Maryland, USA, September 30, 2010*, ACM, pp. 23–32, doi:10.1145/1863509.1863514.

[32] Gerard Tabone & Adrian Francalanza (2021): *Session Types in Elixir*. In Elias Castegren, Joeri De Koster & Simon Fowler, editors: *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, pp. 12–23, doi:10.1145/3486601.3486708.

[33] Gerard Tabone & Adrian Francalanza (2022): *Static Checking of Concurrent Programs in Elixir Using Session Types*. Technical Report, University of Malta, Msida, Malta. Available at `https://gtabone.page.link/V9Hh`.

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

[35] Wiebe-Marten Wijnja: *TypeCheck: Fast and flexible runtime type-checking for your Elixir projects*. Available at `https://github.com/Qqwy/elixir-type_check`.

# Appendix

## A Additional Definitions

In this appendix, we formalise some auxiliary definitions that were used in Sections 2–4.

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

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

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

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

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

**Definition A.3** *(Variable Substitution)***.**

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

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

**Definition A.4** *(Type).*

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

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

$$\mathbf{vars}(\widetilde{p}) \stackrel{\text{def}}{=} \mathbf{vars}(p_1,\ \ldots,\ p_n) \stackrel{\text{def}}{=} \mathbf{vars}(p_1) \cup \cdots \cup \mathbf{vars}(p_n)$$
$$\mathbf{vars}(p) \stackrel{\text{def}}{=} \begin{cases} \emptyset & p = b \\ \{x\} & p = x \\ \mathbf{vars}(w_1) \cup \mathbf{vars}(w_2) & p = [w_1 \mid w_2] \\ \cup_{i \in 1..n} \mathbf{vars}(w_i) & p = \{w_1,\ \ldots,\ w_n\} \end{cases} \qquad \blacksquare$$

**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.

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

$\blacksquare$

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

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

$\blacksquare$

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

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

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

# B Type System Rules

In this appendix, we present the full typing rules of the type system, adapted from [32], which were omitted from the Preliminaries Section.

## B.1 Term Typing

In Section 2.4, we explained a few term typing rules, including [tBranch] and [tChoice]. In Figure 7, we present the full list of term typing rules.

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

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

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

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

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

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

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

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

Figure 7: Term typing rules

## B.2 Expression Typing

Expression are typechecked using the $\Gamma \vdash_{\text{exp}} e : T$ judgement, which states that "an expression $e$ has type $T$, subject to the *variable binding* environment $\Gamma$." The expression typing rules are listed in Figure 8.

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

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

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

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

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

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

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

Figure 8: Expression typing rules

## B.3 Pattern Typing

New variables may be created using patterns in the [tBranch] and [tCase] rules. These variables are matched to a type using the judgement, $\vdash^{w}_{\text{pat}} p : T \rhd \Gamma$. This judgement states that "a pattern $p$ is matched to type $T$, where it produces new variables and their types are collected $\Gamma$; under the assumption that the variable containing the dual *pid*, $w$, remains unchanged." The pattern typing rules are found in Figure 9.

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

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

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

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

Figure 9: Pattern typing rules

# C Proofs

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

## C.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 3 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 4 states that, if $x$ is not a free variable in $t$, then the same result should hold. Lemma 5 consists of two statements that compare the free variables in terms (or expressions) with those that include a substitution.

**Lemma 3.** $x \notin \boldsymbol{fv}(t) \cup \boldsymbol{bv}(t)$ *implies* $t\,[{}^{v}/{}_{x}] = t$

*Proof.* By induction on the structure of $t$. □

**Corollary 4.** $x \notin \boldsymbol{fv}(t)$ *implies* $t\,[{}^{v}/{}_{x}] = t$

*Proof.* A consequence of Lemma 3. □

**Lemma 5.**

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

*Proof.* By induction on the structure of $p$.

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

$$\mathbf{match}(b,b) = [\,] \tag{4a}$$

By the **vars** definition, **vars**$(b) = \emptyset$, which matches the result from eq. (4a) since no variables where substituted.

$[\boldsymbol{p = x}]$ By the **match** definition, for any $v$,

$$\mathbf{match}(x,v) = [{}^{v}/{}_{x}] \tag{4b}$$

By the **vars** definition, **vars**$(x) = \{x\}$, which matches the variable in the substitution of eq. (4b).

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

$$\mathbf{match}(p,v) = \mathbf{match}(w_1,v_1), \mathbf{match}(w_2,v_2) = [{}^{\widetilde{v_1}}/{}_{\widetilde{x_1}}]\,[{}^{\widetilde{v_2}}/{}_{\widetilde{x_2}}] \text{ where} \tag{4c}$$

$$\mathbf{match}(w_1,v_1) = [{}^{\widetilde{v_1}}/{}_{\widetilde{x_1}}] \tag{4d}$$

$$\mathbf{match}(w_2,v_2) = [{}^{\widetilde{v_2}}/{}_{\widetilde{x_2}}] \tag{4e}$$

By case analysis of $w_1$ and $w_2$ from eqs. (4d) and (4e), we conclude that

$$\mathbf{vars}(w_1) = \{\widetilde{x_1}\} \tag{4f}$$

$$\mathbf{vars}(w_2) = \{\widetilde{x_2}\} \tag{4g}$$

We need to show that $\mathbf{vars}([w_1 \mid w_2]) = \{\widetilde{x_1}, \widetilde{x_2}\}$. By the **vars** definition and eqs. (4f) and (4g), $\mathbf{vars}([w_1 \mid w_2]) = \mathbf{vars}(\widetilde{x_1}) \cup \mathbf{vars}(\widetilde{x_2}) = \{\widetilde{x_1}\} \cup \{\widetilde{x_2}\}$. This result matches the variables in the substitutions of eq. (4c).

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

**Lemma 7** (Closed Expression)**.** *$\boldsymbol{fv}(e) = \emptyset$ and $e \rightarrow e'$ implies $\boldsymbol{fv}(e') = \emptyset$*

*Proof.* By induction on the structure of $e$. □

Lemmata 3–7 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 $\alpha$. Lemma 7 is analogous; it states that expressions remain closed after reductions.

**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 $t$.

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

$[\boldsymbol{t} = (\boldsymbol{x} = \boldsymbol{t_1};\ \boldsymbol{t_2})]$ 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

$$t_1 \xrightarrow{\alpha} t'_1 \tag{6a}$$

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

$$\mathbf{fv}(t_1) = \emptyset \tag{6b}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{6c}$$

If we apply the inductive hypothesis to eqs. (6a) and (6b), we get

$$\mathbf{fv}(t'_1) = \emptyset \tag{6d}$$

So, by eqs. (6c) and (6d) 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

$$\mathbf{fv}(v) = \emptyset \tag{6e}$$

$$\mathbf{fv}(t_2) \setminus \{x\} = \emptyset \tag{6f}$$

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 4, $t_2 = t_2\,[{}^{v}\!/\!_{x}]$. Substituting this in eq. (6f), results in $\mathbf{fv}(t_2\,[{}^{v}\!/\!_{x}]) = \emptyset$, as required.

b. If $x \in \mathbf{fv}(t_2)$, then by Lemma 5, we get $\mathbf{fv}(t_2\,[{}^{v}\!/\!_{x}]) = \mathbf{fv}(t_2) \setminus \{x\}$. If we substitute this in eq. (6f), the case holds.

[$\boldsymbol{t = \texttt{send}\,(w, \{\,\texttt{:l}, e_1,\ \ldots,\ e_n\})}$] 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

$$\begin{aligned} t' &= \texttt{send}\left(\iota, \left\{\,\texttt{:l}, v_1,\ \ldots,\ v_{k-1},\ e'_k,\ \ldots,\ e_n\right\}\right) \\ e_k &\to e'_k \end{aligned} \tag{7a}$$

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

$$\mathbf{fv}(\iota) = \emptyset \tag{7b}$$

$$\mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..k-1 \tag{7c}$$

$$\mathbf{fv}(e_i) = \emptyset \text{ for } i \in k..n \tag{7d}$$

Applying the Closed Expression Lemma to eqs. (7a) and (7d), results in $\mathbf{fv}(e_k) = \emptyset$. Using this information along with eqs. (7b–d) and the $\mathbf{fv}$ 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 $\mathbf{fv}$ definition,

$$\mathbf{fv}(\iota) = \emptyset, \quad \mathbf{fv}(v_i) = \emptyset \text{ for } i \in 1..n \tag{7e}$$

To show that $\mathbf{fv}(\{\,\texttt{:l}_\mu, v_1,\ \ldots,\ v_n\}) = \emptyset$, we can apply eq. (7e) to the $\mathbf{fv}$ definition.

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

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i}) = \emptyset \qquad \text{for all } i \in I \tag{8a}$$

Given that current structure of t, we can deduce $t \xrightarrow{\alpha} t'$ using [RBRANCH], where $\alpha = ?\left\{\,\texttt{:l}_j,\ v_1,\ \ldots,\ v_n\right\}$ for some $j \in I$, and

$$\begin{aligned} \mathbf{match}(\widetilde{p}_j, \widetilde{v}) &= \sigma \text{ where } \sigma = \left[{}^{v'_1,\ \ldots,\ v'_k}\!/\!_{x_1,\ \ldots,\ x_k}\right] \\ t' &= t_j\sigma \end{aligned} \tag{8b}$$

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

$$\mathbf{vars}(\widetilde{p}_j) = \{x_1,\ \ldots,\ x_k\} \tag{8c}$$

Substituting eq. (8c) in eq. (8a) (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 5, 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 4. 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.

[$\boldsymbol{t = f\,(w, e_2,,\ \ldots,\ e_n)}$] 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 $\alpha = \tau$, $t = f\,(v_1,\ \ldots,\ v_{k-1},\ e_k,\ \ldots,\ e_n)$, $t' = f\left(v_1,\ \ldots,\ v_{k-1},\ e'_k,\ \ldots,\ e_n\right)$ and

$$e_k \to e'_k \tag{9a}$$

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

$$\mathbf{fv}(v_i) = \emptyset \quad \text{for all } i \in 1..k-1 \tag{9b}$$

$$\mathbf{fv}(e_i) = \emptyset \quad \text{for all } i \in k..n \tag{9c}$$

Applying the Closed Expression Lemma to eqs. (9a) and (9c) (for $i = k$), we get

$$\mathbf{fv}(e_k) = \emptyset \tag{9d}$$

So, using the **fv** definition with eqs. (9b–d), result $\mathbf{fv}(t') = \emptyset$ holds as expected.

2. $[\mathbf{RC\textsc{all}_2}]$ From the rule, we know that $\alpha = f/n$ and

$$t = f\,(\iota, v_2, \ldots, v_n) \tag{9e}$$

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

$$\Sigma\,(f/n) = \Omega \qquad \Omega.\mathtt{body} = t \qquad \Omega.\mathtt{params} = x_2, \ldots, x_n \qquad \Omega.\mathtt{dual} = y \tag{9f}$$

Since term reduction can only happen with respect to a well-formed *function information* environment $\Sigma$, 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

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

Thus, using this information and substituting the information from eq. (9f), we get

$$\mathbf{fv}(\bar{t}) \setminus \{y, x_2, \ldots, x_n\} = \emptyset \tag{9g}$$

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 5, 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 4. 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.

$[\boldsymbol{t} = \mathtt{\mathbf{case}}\ \boldsymbol{e}\ \mathtt{\mathbf{do}}\ (\boldsymbol{p_i} \to \boldsymbol{t_i})_{\boldsymbol{i \in I}}\mathtt{\mathbf{end}}]$ Given that current structure of $t$, we can derive $t \xrightarrow{\alpha} t'$ using two cases:

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

$$e \to e' \tag{10a}$$

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

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(p_i) = \emptyset \quad \text{for all } i \in I \tag{10b}$$

$$\mathbf{fv}(e) = \emptyset \tag{10c}$$

Applying Closed Expression Lemma to eqs. (10a) and (10c), results in $\mathbf{fv}(e') = \emptyset$. Thus, using this information, along with eq. (10b) and the **fv** definition, we get $\mathbf{fv}(t') = \emptyset$ as needed.

2. $[\mathbf{RC\textsc{ase}_2}]$ From the rule, we know that $t = \mathtt{case}\ v'\ \mathtt{do}\ (p_i \to t_i)_{i \in I}\mathtt{end}$, $e = v'$ and for some $j \in I$,

$$\mathbf{match}(p_j, v') = \sigma \text{ where } \sigma = [{}^{v_1,\ \ldots,\ v_n}/_{x_1,\ \ldots,\ x_n}] \tag{10d}$$

$$t' = t_j\sigma \tag{10e}$$

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

$$\mathbf{fv}(t_i) \setminus \mathbf{vars}(\widetilde{p_i}) = \emptyset \quad \text{for all } i \in I \tag{10f}$$

From eq. (10d), we can apply Lemma 6, to get

$$\mathbf{vars}(p_j) = \{x_1, \ldots, x_k\} \tag{10g}$$

Substituting eq. (10g) in eq. (10f) (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. □

## C.2 Proofs for Theorem 2

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

**Lemma 8** ($\Delta$-Weakening)**.** *If $\Delta \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$, then $(\Delta, \Delta') \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$*

*Proof.* Follows by induction on the derivation of $\Delta \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$. We analyse the significant cases:

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

$$(\Delta, f/n : S) \cdot (\Gamma, \Gamma') \vdash^y S \rhd \bar{t} : T \lhd S' \tag{11a}$$

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{11b}$$

Applying the inductive hypothesis to eq. (11a) results in $(\Delta, \Delta', f/n : S) \cdot (\Gamma, \Gamma') \vdash^y S \rhd t : T \lhd S'$, where we assume that $f/n \notin \mathbf{dom}(\Delta')$. So, using the latter result, eq. (11b) and [TRecUnknownCall] results in $(\Delta, \Delta') \cdot \Gamma \vdash^w S \rhd t : T \lhd S'$, as required.

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

$$\Delta\,(f/n) = S \tag{12a}$$

$$\Gamma \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{12b}$$

If we extend $\Delta$ by $\Delta'$, then $(\Delta, \Delta')(f/n) = S$ remains valid. So, using this information, along with eq. (12b) in [TRecKnownCall], we get $(\Delta, \Delta') \cdot \Gamma \vdash^w S \rhd t : T \lhd \mathsf{end}$, as required.

Cases [TChoice] and [TExpression] hold immediately since $\Delta$ 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 9) 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 ($\Gamma$), and thus can be removed from $\Gamma$. This lemma consists of two statements, where substitution is performed in *(i)* terms, and *(ii)* expressions.

**Lemma 9** (Substitution)**.**

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

*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 $\Delta \cdot (\Gamma, x : T') \vdash^{w} S \rhd t : T \lhd S'$ 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

$$x' \neq w \tag{13a}$$

$$\Gamma \vdash_{\text{exp}} v : T' \tag{13b}$$

$$\Delta \cdot (\Gamma, x : T') \vdash^{w} S \rhd t_1 : T'' \lhd S'' \tag{13c}$$

$$\Delta \cdot (\Gamma, x : T', x' : T'') \vdash^{w} S'' \rhd t_2 : T \lhd S' \tag{13d}$$

The *variable binding* environment of eq. (13d) can be reordered to

$$\Delta \cdot (\Gamma, x' : T'', x : T') \vdash^{w} S'' \rhd t_2 : T \lhd S' \tag{13e}$$

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

$$\Delta \cdot \Gamma \vdash^{w[{}^{v}/_{x}]} S \rhd x' = t_1\,[{}^{v}/_{x}];\, t_2\,[{}^{v}/_{x}] : T \lhd S' \tag{13f}$$

for $x \neq x'$ and $x' \neq v$. To obtain eq. (13f), we need some preliminary results. Applying the inductive hypothesis to eqs. (13b) and (13c), and similarly to eqs. (13b) and (13e), results in

$$\Delta \cdot \Gamma \vdash^{w[{}^{v}/_{x}]} S \rhd t_1\,[{}^{v}/_{x}] : T'' \lhd S'' \tag{13g}$$

$$\Delta \cdot (\Gamma, x' : T'') \vdash^{w[{}^{v}/_{x}]} S'' \rhd t_2\,[{}^{v}/_{x}] : T \lhd S' \tag{13h}$$

From eq. (13a) and the Variable Substitution Definition we know that $x \neq w\,[{}^{v}/_{x}]$. Applying this information, along with eqs. (13g) and (13h) to the premise of [tLet] results in eq. (13f), as required.

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

$$\Gamma \vdash_{\text{exp}} v : T' \tag{14a}$$

$$\begin{gathered} S = \&\left\{ ?\texttt{l}_i\left(T_i^1, \ldots, T_i^n\right).S_i \right\}_{i \in I} \\ t = \texttt{receive do } \left(\left\{ \texttt{:l}_i, p_i^1, \ldots, p_i^n \right\} \rightarrow t_i\right)_{i \in I}\texttt{end} \end{gathered} \tag{14b}$$

From the premise, we also know that, for all $i \in I$:

$$\vdash^{w}_{\text{pat}} p_i^j : T_i^j \;\rhd\; \Gamma_i^j \qquad \text{for all } j \in 1..n \tag{14c}$$

$$\Delta \cdot \left(\Gamma, x : T', \Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^{w} S_i \rhd t_i : T \lhd S' \tag{14d}$$

This case holds if the following statement is obtained:

$$\Delta \cdot \Gamma \vdash^{w[{}^{v}/_{x}]} S_i \rhd t_i\,[{}^{v}/_{x}] : T \lhd S' \tag{14e}$$

where $t\,[{}^{v}/_{x}] = \texttt{receive do } \left(\left\{ \texttt{:l}_i, p_i^1, \ldots, p_i^n \right\} \rightarrow\right)_{i \in I}\texttt{end}$. To obtain eq. (14e) we need to use the [tBranch] rule which requires multiple premises. Applying the inductive hypothesis to eqs. (14a) and (14d) results in

$$\Delta \cdot \left(\Gamma, \Gamma_i^1, \ldots, \Gamma_i^n\right) \vdash^{w[{}^{v}/_{x}]} S_i \rhd t_i\,[{}^{v}/_{x}] : T \lhd S' \qquad \text{for all } i \in I \tag{14f}$$

If $w \neq x$, then eq. (14c)

$$\vdash^{w[{}^{v}/_{x}]}_{\text{pat}} p_i^j : T_i^j \rhd \Gamma_i^j \quad \text{for all } j \in 1..n \tag{14g}$$

since by the Variable Substitution Definition, $w = w\,[{}^{v}/_{x}]$. Therefore, eqs. (14f) and (14g) can be applied to the premise of [TBRANCH] to obtain eq. (14e):

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

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. (14e) as required.

[TCHOICE] From the rule, we know that for some $i \in I$, $T = \{\texttt{atom}, T_i^1, \ldots, T_i^n\}$, $S = \oplus\{!\texttt{l}_i(\widetilde{T_i}).S_i\}_{i \in I}$ and

$$t = \texttt{send}(\iota, \{:\texttt{l}_i, e_1, \ldots, e_n\}) \tag{15a}$$

$$\Gamma, x : T' \vdash_{\text{exp}} e_j : T_i^j \quad \text{for all } j \in 1..n \tag{15b}$$

$$\Gamma \vdash_{\text{exp}} v : T' \tag{15c}$$

Applying eqs. (15b) and (15c) to Item *ii* of Lemma 9 results in $\Gamma \vdash_{\text{exp}} e_j\,[{}^{v}/_{x}] : T_i^j$ 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 $t\,[{}^{v}/_{x}] = \texttt{send}(w\,[{}^{v}/_{x}], \{:\texttt{l}_i, e_1\,[{}^{v}/_{x}], \ldots, e_n\,[{}^{v}/_{x}]\})$. □

Lemma 10 links expression types to the basic values (and vice versa), *e.g.* the value 5 has type number.

**Lemma 10** (Value Typing)**.**

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

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

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

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

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

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

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

Lemma 11 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 $\Gamma$, are assigned with the same types. Consequently, Corollary 12 provides the same guarantees but for a *sequence* of patterns and values.

**Lemma 11.** *For all patterns p and values v,*

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

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

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

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

$$\emptyset \vdash_{\text{exp}} v : T. \tag{16a}$$

From $\vdash^{w}_{\text{pat}} x : T \rhd \Gamma$ and $[\textsc{tpVariable}]$, we know that $\Gamma$ must contain $x : T$ only. Therefore, case holds by eq. (16a).

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

$$\mathbf{match}(w_1,v_1) = [{}^{v'_1,\,\ldots,\,v'_j}/_{x_1,\,\ldots,\,x_j}] \tag{17a}$$
$$\mathbf{match}(w_2,v_2) = [{}^{v'_k,\,\ldots,\,v'_n}/_{x_k,\,\ldots,\,x_n}] \text{ where } k = j+1 \tag{17b}$$

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

$$\emptyset \vdash_{\text{exp}} v_1 : T \text{ and } \emptyset \vdash_{\text{exp}} v_2 : [T] \tag{17c}$$

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

$$\vdash^{w}_{\text{pat}} w_1 : T \rhd \Gamma' \text{ and } \vdash^{w}_{\text{pat}} w_2 : [T] \rhd \Gamma'' \tag{17d}$$

Applying the inductive hypothesis twice to eqs. (17a–d) results in

$$\Gamma' = x_1 : T_1,\, \ldots,\, x_j : T_j \text{ and } \Gamma'' = x_k : T_k,\, \ldots,\, x_n : T_n \tag{17e}$$
$$\emptyset \vdash_{\text{exp}} v'_i : T_i \text{ for all } i \in 1..n \tag{17f}$$

Therefore, case holds by eqs. (17e) and (17f), since $\Gamma = \Gamma', \Gamma''$.

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

$$\mathbf{match}(w_i,v_i) = \sigma_i \text{ given that } \sigma = \sigma_1,\, \ldots,\, \sigma_m \tag{18a}$$

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

$$\emptyset \vdash_{\text{exp}} v_i : T_i \tag{18b}$$

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

$$\vdash^{w}_{\text{pat}} w_i : T_i \rhd \Gamma_i \tag{18c}$$

Applying the inductive hypothesis $m$ times to eqs. (18a–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 12.** *For all patterns $\widetilde{p} = p^1,\ \ldots,\ p^n$, values $\widetilde{v} = v_1,\ \ldots,\ v_n$ and $\forall j \in 1..n$, then the following implication holds.*

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

*Proof.* Take $j = 1$, where we know that $\mathbf{match}(p^1, v_1) = \sigma_1$, $\vdash^{y}_{\text{pat}} p^1 : T^1 \,\rhd\, \Gamma^1$ and $\emptyset \vdash_{\text{exp}} v_1 : T^1$. Then, applying this information to Lemma 11, we get

$$\Gamma^1 = x^1_1 : T^1_1,\ \ldots,\ x^1_m : T^1_m \tag{19a}$$

$$\emptyset \vdash_{\text{exp}} v^1_i : T^1_i \ \text{for}\ i \in 1..m \tag{19b}$$

Generalising for $j \in 1..n$, then $\widetilde{\Gamma} = \Gamma^1,\ \ldots,\ \Gamma^n$ holds by generalising eq. (19a). Also, $\emptyset \vdash_{\text{exp}} v'_i : T_i$ for $i \in 1..k$ holds by eq. (19b). Thus, Corollary 12 holds by applying Lemma 11 $n$ times. □

Lemma 13 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 13** (Preservation (Expressions))**.** *If $\emptyset \vdash_{exp} e : T$ and $e \longrightarrow e'$, 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

$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all}\ i \in 1..n \tag{20a}$$

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

$$e_k \longrightarrow e'_k \tag{20b}$$

Applying eqs. (20a) and (20b) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_k : T_k$. By the latter, eq. (20a) and [TTUPLE], we get $\emptyset \vdash_{\text{exp}} e' : T$, as required.

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

$$\emptyset \vdash_{\text{exp}} e_1 : \mathsf{number} \tag{21a}$$

$$\emptyset \vdash_{\text{exp}} e_2 : \mathsf{number} \tag{21b}$$

$e \longrightarrow e'$ 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

$$e_1 \longrightarrow e'_1 \tag{21c}$$

   Applying eqs. (21a) and (21c) to the inductive hypothesis results in $\emptyset \vdash_{\text{exp}} e'_1 : \mathsf{number}$. Using this information, along with eq. (21b) in [TARITHMETIC], results in $\emptyset \vdash_{\text{exp}} e' : \mathsf{number}$, as required.

2. [**REOPERATION$_2$**] Analogous to [REOPERATION$_1$].
3. [**REOPERATION$_3$**] From the rule, we know that $e = v_1 \diamond v_2$ and $e'$ has some value $v = v_1 \diamond v_2$. Since we know that $\emptyset \vdash_{\text{exp}} e : T$, or $\emptyset \vdash_{\text{exp}} v_1 \diamond v_2 : T$, then $\emptyset \vdash_{\text{exp}} e' : T$ follows immediately given that $e' = v = v_1 \diamond v_2$.

Regarding the remaining cases: Cases [TLITERAL], [TVARIABLE] and [TELIST] hold trivially, since $e \to e'$ does not apply. Cases [TCOMPARISON] and [TBOOLEAN] are analogous to [TARITHMETIC]. Cases [TLIST] and [TNOT] take a similar approach to [TTUPLE]. □

Lemmata 8–13 allow us to prove the Session Fidelity Theorem. This is the main result of Section 4.

**Theorem 2** (Session Fidelity). *If $\Delta \cdot \emptyset \vdash^w_\Sigma S \triangleright t : T \triangleleft S'$ and $t \xrightarrow[\Sigma]{\alpha} t'$, then there exists some $S''$ and $\Delta'$, such that $\Delta' \cdot \emptyset \vdash^w_\Sigma S'' \triangleright t' : T \triangleleft S'$ for **after**$(S, \alpha) = S''$ and **after**$(\Delta, \alpha, S) = \Delta'$*

*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 $x \neq w$, and

$$t = (x = t_1;\, t_2) \tag{22a}$$

$$\Delta \cdot \emptyset \vdash^w S \triangleright t_1 : T' \triangleleft S''' \tag{22b}$$

$$\Delta \cdot \left(x : T'\right) \vdash^w S''' \triangleright t_2 : T \triangleleft S' \tag{22c}$$

From the structure of $t$ (eq. (22a)), 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

$$t_1 \xrightarrow{\alpha} t_1' \tag{22d}$$

By eqs. (22b) and (22d) and the inductive hypothesis we obtain

$$\Delta' \cdot \emptyset \vdash^w S'' \triangleright t_1' : T' \triangleleft S''' \tag{22e}$$

where **after**$(S, \alpha) = S''$ and **after**$(\Delta, \alpha, S) = \Delta'$. Also, by the After Function Definition, we know that $\Delta'$ is an extension of $\Delta$, so we can apply the $\Delta$-Weakening Lemma on eq. (22c) to get

$$\Delta' \cdot \left(x : T'\right) \vdash^w S''' \triangleright t_2 : T \triangleleft S' \tag{22f}$$

Using eqs. (22e) and (22f) as the premise for rule [TLET], we obtain:

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

where $\Delta' \cdot \emptyset \vdash^w S'' \triangleright t' : T \triangleleft S'$ is the expected result.

2. [**RLET$_2$**] From the rule, we know that $t = (x = v;\, t_2)$, $t' = t_2\,[{}^{v}\!/\!_{x}]$ and $\alpha = \tau$. Since $t_1 = v$, by eq. (22b) and [TEXPRESSION], then $\emptyset \vdash_{\text{exp}} v : T'$ holds. If we apply this latter information and eq. (22c) to the Substitution Lemma, we obtain $\Delta \cdot \emptyset \vdash^{w[{}^{v}\!/\!_{x}]} S''' \triangleright t_2\,[{}^{v}\!/\!_{x}] : T \triangleleft S'$. This is the expected result, since by the Variable Substitution Definition, $w\,[{}^{v}\!/\!_{x}] = w$; and by the **after** definition, $S''' = S$ and $\Delta' = \Delta$.

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

$$S = \&\big\{?\mathtt{l}_i\big(T_i^1,\ \dots,\ T_i^n\big).S_i\big\}_{i\in I} \tag{23a}$$

$$t = \mathtt{receive\ \ do}\ \big(\big\{\,:\mathtt{l}_i, p_i^1,\ \dots,\ p_i^n\big\} \to t_i\big)_{i\in I}\mathtt{end} \tag{23b}$$

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

$$\forall i \in I \begin{cases} \vdash^w_{\text{pat}} p_i^j : T_i^j \ \triangleright\ \Gamma_i^j \ \text{ for all } j \in 1..n & \text{(23c)} \\ \Delta \cdot \big(\Gamma_i^1,\ \dots,\ \Gamma_i^n\big) \vdash^w S_i \triangleright t_i : T \lhd S' & \text{(23d)} \end{cases}$$

From the structure of $t$ (eq. (23b)), term reduction ($t \xrightarrow{\alpha} t'$) can only be derived using [rBranch], where execution progresses to a single branch (*i.e.*, $t_\mu$), 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

$$\alpha = ?\big\{\,:\mathtt{l}_\mu,\ v_1,\ \dots,\ v_n\big\} \tag{23e}$$

$$\mathbf{match}\big(\big(p_\mu^1,\ \dots,\ p_\mu^n\big), \big(v_1,\ \dots,\ v_n\big)\big) = \big[v'_1, \dots, v'_k / x_1, \dots, x_k\big] \tag{23f}$$

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

From eq. (23e), $\alpha$ refers to the message received from the dual process. We can compare the contents of this message to the original session type $S$ (eq. (23a)), to obtain information regarding the types of the individual values inside $\alpha$. We know that $\alpha$ 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 $?\mathtt{l}_\mu\big(\boldsymbol{T_\mu^1},\ \dots,\ \boldsymbol{T_\mu^n}\big).S_\mu$. Formally, this can be written as

$$\emptyset \vdash_{\text{exp}} v_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{23g}$$

Applying eqs. (23c), (23f) and (23g) into Corollary 12, results in $\widetilde{\Gamma_\mu} = \Gamma_\mu^1,\ \dots,\ \Gamma_\mu^n = x_1 : T_1,\ \dots,\ x_k : T_k$ and

$$\emptyset \vdash_{\text{exp}} v'_m : T_m \ \text{ for } m \in 1..k \tag{23h}$$

Applying eq. (23h) and $\Delta \cdot \widetilde{\Gamma_\mu} \vdash^w S_\mu \triangleright t_\mu : T \lhd S'$ (from eq. (23d) for $i = \mu$) repeatedly to the Substitution Lemma, we get

$$\Delta \cdot \emptyset \vdash^w S_\mu \triangleright t_\mu\big[v'_1, \dots, v'_k / x_1, \dots, x_k\big] : T \lhd S' \tag{23i}$$

Since $\mathbf{after}\big(\&\big\{?\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\big\}_{i\in I}, \alpha\big) = S_\mu$ and $\mathbf{after}(\Delta, \alpha, S) = \Delta$, then eq. (23i) is the expected result.

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

$$S = \oplus\big\{!\mathtt{l}_i\big(\widetilde{T_i}\big).S_i\big\}_{i\in I} \tag{24a}$$

$$t = \mathsf{send}\,\big(\iota, \big\{\,:\mathtt{l}_\mu, e_1,\ \dots,\ e_n\big\}\big) \tag{24b}$$

$$\emptyset \vdash_{\text{exp}} e_j : T_\mu^j \ \text{ for all } j \in 1..n \tag{24c}$$

From the structure of $t$ (eq. (24b)), 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 [**RChoice₁**], we know that $\alpha = \tau$ and

$$t' = \texttt{send}\left(\iota, \left\{\texttt{:l}, v_1,\, \dots,\, v_{k-1},\, e'_k,\, \dots,\, e_n\right\}\right)$$
$$e_k \to e'_k \tag{24d}$$

   Applying eq. (24c) (for $j = k$) and eq. (24d) to the Preservation (Expressions) Lemma, we get $\emptyset \vdash_{\text{exp}} e'_k : T_k$. Applying this and eq. (24c) to [tChoice] results in $\Delta \cdot \emptyset \vdash^{w} S \rhd t' : T \lhd S_\mu$. Since $\textbf{after}(S, \tau) = S$ and $\textbf{after}(\Delta, \alpha, S) = \Delta$, this holds.
2. [**RChoice₂**] From this rule we know that

$$t' = \left\{\texttt{:l}_\mu, v_1,\, \dots,\, v_n\right\}$$
$$\alpha = \iota\,!\left\{\texttt{:l}_\mu, v_1,\, \dots,\, v_n\right\} \tag{24e}$$

   where $\alpha$ (eq. (24e)) is the message being sent to the dual process with *pid* $\iota$.
   Recall eq. (24c), where we have $\emptyset \vdash_{\text{exp}} e_j : \mathbf{T}^{\mathbf{j}}_{\text{-}}$ for $j \in 1..n$. Notice, that the types $\boldsymbol{T}^{\boldsymbol{j}}_{\boldsymbol{\mu}}$ were obtained from the session type $S$ (eq. (24a)), where $S$ contains $!\texttt{l}_\mu\left(\,\boldsymbol{T}^{\mathbf{1}}_{\boldsymbol{\mu}},\, \dots,\, \boldsymbol{T}^{\boldsymbol{n}}_{\boldsymbol{\mu}}\,\right).S_\mu$. Now, by the premise of [RChoice₂], since $e_j = v_j$, then

$$\emptyset \vdash_{\text{exp}} v_j : T^j_\mu \;\text{ for all } j \in 1..n \tag{24f}$$

   By the Value Typing Lemma, we also know that $\emptyset \vdash_{\text{exp}} \texttt{:l}_\mu : \mathsf{atom}$. Using this latter information and eq. (24f) in [tTuple] and [tExpression], we get the required result:

$$[\textsc{tExpression}]\;\dfrac{[\textsc{tTuple}]\;\dfrac{\emptyset \vdash_{\text{exp}} \texttt{:l}_\mu : \mathsf{atom} \qquad \forall j \in 1..n \qquad \emptyset \vdash_{\text{exp}} v_j : T^j_\mu}{\emptyset \vdash_{\text{exp}} \left\{\texttt{:l}_\mu, v_1,\, \dots,\, v_n\right\} : \left\{\mathsf{atom}, T^1_\mu,\, \dots,\, T^n_\mu\right\}}}{\Delta \cdot \emptyset \vdash^{y} S_\mu \rhd \left\{\texttt{:l}_\mu, v_1,\, \dots,\, v_n\right\} : T \lhd S_\mu} \tag{24g}$$

   Result from eq. (24g) holds as required, since $\textbf{after}(S, \alpha) = S_\mu$ and $\textbf{after}(\Delta, \alpha, S) = \Delta$.

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

$$t = f\,(w, e_2,\, \dots,\, e_n) \tag{25a}$$
$$\emptyset \vdash_{\text{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{25b}$$

From the structure of $t$ (eq. (25a)), term transitions ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**RCall₁**] From this rule, we know that $t = f\,(v_1,\, \dots,\, v_{k-1},\, e_k,\, \dots,\, e_n)$, $\alpha = \tau$, $w = v_1$ and

$$t' = f\left(v_1,\, \dots,\, v_{k-1},\, e'_k,\, \dots,\, e_n\right)$$
$$e_k \to e'_k \tag{25c}$$

   Applying eq. (25b) (for $i = k$) and eq. (25c) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e'_k : T_k \tag{25d}$$

   By eqs. (25b) and (25d) and [tRecKnownCall], we get

$$\Delta \cdot \emptyset \vdash^{w} S \rhd f\left(v_1,\, \dots,\, v_{k-1},\, e'_k,\, \dots,\, e_n\right) : T \lhd S' \tag{25e}$$

   eq. (25e) holds since $\textbf{after}(S, \tau) = S$ and $v_1 = w$.

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

$$t = f\,(\iota, v_2,\, \dots,\, v_n) \tag{25f}$$

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

$$\Sigma\,(f/n) = \Omega \text{ where } \begin{cases} \Omega.\mathtt{return\_type} = T \\ \Omega.\mathtt{param\_types} = T_2,\, \dots,\, T_n \end{cases} \tag{25g}$$

$$\Delta\,(f/n) = S \tag{25h}$$

Since all *known* functions (*i.e.,* $f/n \in \mathbf{dom}(\Delta)$) by eq. (25h)) are already typechecked once before, then from the *function information* environment (*i.e.,* $\Sigma$) and eq. (25g), we can assume that

$$\Delta \cdot \Gamma' \vdash^{y} S \rhd \tilde{t} : T \lhd \mathsf{end} \tag{25i}$$

where $\Gamma'$ contains *only* the mapping from the parameter names to their types, *i.e.,* $\Gamma' = (y : \mathsf{pid}, x_2 : T_2,\, \dots,\, x_n : T_n)$ – our aim is to change $\Gamma'$ to $\emptyset$. This assumption in eq. (25i) is possible since a well-formed $\Sigma$ 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. (25f) and Value Typing Lemma we know that $\emptyset \vdash_{\mathrm{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (25i) to the Substitution Lemma results in

$$\Delta \cdot (x_2 : T_2,\, \dots,\, x_n : T_n) \vdash^{y[\iota/y]} S \rhd \tilde{t}\,[\iota/y] : T \lhd \mathsf{end} \tag{25j}$$

where by the Variable Substitution Definition, $y\,[\iota/y] = \iota = w$.
Applying the Substitution Lemma multiple times to eqs. (25b) and (25j), results in

$$\Delta \cdot \emptyset \vdash^{w} S \rhd \tilde{t}\,[\iota/y]\,[v_2,\, \dots,\, v_n / x_2,\, \dots,\, x_n] : T \lhd \mathsf{end} \tag{25k}$$

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. (25h), $f/n$ is already mapped to $S$ in the *session typing* environment, therefore $(\Delta, f/n : S) = \Delta$, as needed.

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

$$t = f\,(w, e_2,,\, \dots,\, e_n) \tag{26a}$$

$$\emptyset \vdash_{\mathrm{exp}} e_i : T_i \quad \text{for all } i \in 2..n \tag{26b}$$

From the premise we also know that

$$(\Delta, f/n : S) \cdot \big(y : \mathsf{pid}, \widetilde{x} : \widetilde{T}\big) \vdash^{y} S \rhd \tilde{t} : T \lhd S' \quad \text{where } \widetilde{x}, \widetilde{T}, \tilde{t}, T \text{ and } y \text{ are obtained from the } \textit{function information} \text{ environment}(\textit{i.e.,}\Sigma) \tag{26c}$$

From the structure of $t$ (eq. (26a)), term transitions ($t \xrightarrow{\alpha} t'$) can be derived using two rules, so we consider two sub-cases:

1. [**RCALL₁**] From this rule we know that $\alpha = \tau$, and

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

$$e_k \rightarrow e'_k \tag{26d}$$

Applying eq. (26b) (for $i = j$) and eq. (26d) to the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e'_j : T_j \tag{26e}$$

Using eq. (26b) and eq. (26e) in the rule [TRECUNKNOWNCALL], results in

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

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

2. [**RCALL₂**] From the rule, we know that $\alpha = f/n$ and

$$t = f\left(\iota, v_2,\ \ldots,\ v_n\right) \tag{26f}$$

$$w = \iota \tag{26g}$$

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

By eq. (26f) and the Value Typing Lemma we know that $\emptyset \vdash_{\text{exp}} \iota : \mathsf{pid}$. Applying this information and eq. (26c) to the Substitution Lemma results in

$$\left(\Delta, f/n : S\right) \cdot \left(\widetilde{x} : \widetilde{T}\right) \vdash^{y\left[{}^{\iota}/_{y}\right]} S \triangleright \bar{t}\left[{}^{\iota}/_{y}\right] : T \triangleleft S' \tag{26h}$$

where by the Variable Substitution Definition and eq. (26g), $y\left[{}^{\iota}/_{y}\right] = \iota = w$.
Applying the Substitution Lemma repeatedly to eqs. (26b) and (26h), results in

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

where $\textbf{after}(S, f/n) = S$ and $\textbf{after}(\Delta, f/n, S) = (\Delta, f/n : S)$, as required.

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

$$t = \texttt{case}\ e\ \texttt{do}\ \left(p_i \rightarrow t_i\right)_{i \in I}\texttt{end} \tag{27a}$$

$$\emptyset \vdash_{\text{exp}} e : U \tag{27b}$$

$$\vdash^{w}_{\text{pat}} p_i : U\ \triangleright\ \Gamma'_i \qquad \text{for all } i \in I \tag{27c}$$

$$\Delta \cdot \Gamma'_i \vdash^{w} S \triangleright t_i : T \triangleleft S' \qquad \text{for all } i \in I \tag{27d}$$

By eq. (27a), term reduction, $t \xrightarrow{\alpha} t'$, can be derived using two rules, so we consider two sub-cases:

1. [**RCASE₁**] From the rule we know that $t' = \texttt{case}\ e'\ \texttt{do}\ \left(p_i \rightarrow t_i\right)_{i \in I}\texttt{end}$, and from the premise we know that

$$e \rightarrow e' \tag{27e}$$

By eqs. (27b) and (27e) and the Preservation (Expressions) Lemma, we get

$$\emptyset \vdash_{\text{exp}} e' : U \tag{27f}$$

Using eqs. (27c), (27d), (27f), and [TCASE], we get

$$\Delta \cdot \emptyset \vdash^{w} S \triangleright \texttt{case}\ e'\ \texttt{do}\ \left(p_i \rightarrow t_i\right)_{i \in I}\texttt{end} : T \triangleleft S'$$

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

2. $[\textbf{R}\textsc{Case}_\mathbf{2}]$ 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 $j \in I$,

$$\textbf{match}(p_j, v) = \sigma \text{ where } \sigma = [{}^{v_1,\ \ldots,\ v_n}/_{x_1,\ \ldots,\ x_n}] \tag{27g}$$

$$t' = t_j \sigma \tag{27h}$$

By eqs. (27b), (27c) and (27g) and lemma 11, we know that $\Gamma'_j = x_1 : T_1,\ \ldots,\ x_n : T_n$ and

$$\emptyset \vdash_{\text{exp}} v_k : T_k \text{ for all } k \in 1..n \tag{27i}$$

Then, by repeatedly applying the Substitution Lemma to eq. (27i), (27d 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 $\textbf{after}(\Delta, \tau, S) = \Delta$. □