A Simple Library Implementation of Binary Sessions

Luca Padovani

A Simple Library Implementation of Binary Sessions

Luca Padovani – University of Torino, Italy

Abstract. We leverage on former foundational studies on binary sessions to realize a session type system using only ordinary notions of generic types and of type equality. The type system does not always prevent non-linear usages of session endpoints, but linearity violations that may compromise safety are detected at runtime. We demonstrate the approach implementing a simple, well-integrated `OCaml` library for session communications. As a bonus, `OCaml` infers possibly recursive, polymorphic session types and also supports a form of session subtyping.

1 Introduction

One major obstacle to the adoption and integration of session type systems into mainstream programming languages is their reliance on sophisticated and peculiar features, whose built-in support would require massive changes in both languages and their development tools. In the case of binary sessions [15,16], for example, the compiler would have to integrate features for (F.1) describing *structured protocols* as sequences of I/O operations and internal/external choices, (F.2) checking the *duality* of the protocols associated with the endpoints of a session to make sure that they are used in complementary ways, (F.3) tracking the *changes* in the type of endpoints as these are used for I/O operations, and (F.4) ensuring the *linear usage* of endpoints.

In this paper we show how to incorporate a form of session type checking into a broad class of conventional programming languages, by recasting these fancy features into ordinary ones. A step in this direction was taken by Gay and Vasconcelos [14], who realized that (F.3) could be achieved by a clever typing of the communication primitives. The technique is illustrated in program `P` below, written in an `ML`-like language:

```
let foo x0 =
  let n, x1 = receive x0 in       (* x0 : ?int.!bool.end  *)
  let x2 = send x1 (n = 0) in     (* x1 : !bool.end       *)
  close x2                        (* x2 : end             *)

let bar y0 =
  let y1 = send y0 42 in          (* y0 : !int.?bool.end  *)
  let b, y2 = receive y1 in       (* y1 : ?bool.end       *)
  close y2; print b               (* y2 : end             *)
```

According to [14], the primitives `send` and `receive` *consume* a session endpoint and *return* the same endpoint with a possibly different session type. Take for example the session type `!int.?bool.end`, which denotes an endpoint for sending an `int` and receiving a `bool`, in this order, before being closed. Then `send y0 42` in `bar` consumes endpoint `y0` of type `!int.?bool.end`, sends `42` on it, and returns the endpoint with type `?bool.end`, which is then bound to `y1`. Similarly, `receive y1` consumes `y1` of

type `?bool.end`, waits for a message from it, and returns a pair with the received message `b` of type `bool` and the endpoint with type `end`, which is bound to y_2. As the operational semantics in [14] clearly illustrates, the y_i's all actually refer to the same endpoint; its subsequent rebindings allow the type checker to track the change in its type without using any dedicated mechanism. Observe that `foo` has a complementary behavior compared to `bar`. This is witnessed by the session type of x_0, which is the *dual* of that of y_0: inputs have become outputs, and vice versa. If `foo` and `bar` are applied to the two endpoints of a session, duality guarantees communication safety.

The given semantics and typing of the communication primitives are not the only possible ones. An alternative semantics and a corresponding typing for the same primitives emerges from the studies of Kobayashi [20], Demangeon and Honda [10], and Dardha, Giachino and Sangiorgi [9]. These works show that an arbitrary sequence of communications in a session can be encoded as a sequence of communications in a *chain of linear channels*, each channel being used for one communication only. The chain is realized by pairing the payload in each message with a *continuation*, that is a fresh channel on which the next communication takes place. Let us re-interpret `P` as the program `Q` below, which is syntactically the same as `P`, but uses this alternative semantics and typing of the communication primitives:

```
let foo x0 =
  let n, x1 = receive x0 in       (* x0 : ?[int * ![bool * ∅[unit]]]  *)
  let x2 = send x1 (n = 0) in     (* x1 : ![bool * ∅[unit]]           *)
  close x2                        (* x2 : ∅[unit]                     *)

let bar y0 =
  let y1 = send y0 42 in          (* y0 : ![int * ![bool * ∅[unit]]]  *)
  let b, y2 = receive y1 in       (* y1 : ?[bool * ∅[unit]]           *)
  close y2; print b               (* y2 : ∅[unit]                     *)
```

Here x_i and y_j are linear channels and the communication primitives explicitly create and exchange continuations. Channel types have the form $\kappa[t]$, where $\kappa \in \{?, !, \emptyset\}$ is a capability denoting an input, an output, or the closing of a channel and t is the type of messages exchanged on the channel. For instance, the type `![int * ![bool * ∅[unit]]]` indicates that y_0 is a channel for sending a pair made of an `int` and another channel of type `![bool * ∅[unit]]`. Now the effect of `send` y_0 `42` is to create a fresh channel, to pair `42` with one reference to such channel of type `![bool * ∅[unit]]`, to send the pair, and to return another reference to the same fresh channel of type `?[bool * ∅[unit]]`. Accordingly, the type of the first reference matches that of x_1 in `foo` and the type of the second reference matches that of y_1 in `bar`.

Two reasons make the alternative typing of communication primitives in program `Q` relevant to our aims. First, program `Q` only uses ordinary types (channel types and products). Even if the exact correspondence between the session types in `P` and the types in `Q` is not entirely obvious, it is clear that both describe the same protocol, just written in different ways. Second, the structurally complex duality relation between the session types of x_i and y_i in `P` boils down to a much simpler duality relation between the channel types of x_i and y_i in `Q` matching input `?` with output `!` capabilities, *but only in the topmost channel type constructor*. This relation, as we will see, can be expressed solely in terms of type equality, given an appropriate representation of channel types.

In a nutshell, the communication primitives in program P have a natural semantics (not creating or exchanging continuations) but require fancy types and related notions; the communication primitives in program Q have an impractical semantics (creating and exchanging continuations) but their types look and behave much like ordinary ones.

The key observation is that it is makes sense to consider a third, intermediate configuration in which the communication primitives do not create or exchange continuations (as in program P) but are typed *as if* they did create and exchange continuations (as in program Q). This mix-up can be justified as an optimized implementation of the communication primitives in program Q: as the authors of [9] point out, there is no need to actually create fresh continuations, for the already existing channel can be reused. We take this optimization one step further: not only continuations need not be created, they need not be exchanged either, precisely because the channel being reused is already known by the interacting processes. In the end, we obtain a set of primitives whose semantics matches exactly that for session communications, but whose typing allows us to realize (F.1–3) in terms of standard notions of generic types and type equality. We do not propose or adopt compile-time mechanisms for (F.4). Instead, we rely on the runtime environment for detecting non-linear endpoint usages that may compromise safety. We will see that the type system is nonetheless capable of identifying a fair number of linearity violations, even if it is not intentionally designed to do so (Example 4).

Here is an account of our contributions:

1. We formalize a core functional language, called FuSe, that combines multithreading and session-based communications in the style of [14] with a runtime mechanism that detects endpoint linearity violations (Section 2).
2. We define an ordinary ML-style type language for FuSe and we adapt and extend the encoding of session types [9] into FuSe types. A carefully chosen representation of channel types allows us to express (encoded) session type duality solely in terms of type equality and generic types (Section 3).
3. We equip FuSe with a standard ML-style type system and we type FuSe primitives using encoded (as opposed to built-in) session types. Well-typed FuSe programs are shown to enjoy all the usual properties of sessions (safety, fidelity, progress) under the hypothesis that they use session endpoints linearly (Section 4).
4. To demonstrate the effectiveness of the approach, we detail the implementation of FuSe primitives as a simple OCaml module [22]. The implementation integrates well with OCaml, which is capable of inferring possibly recursive, polymorphic session types and of supporting a form of session subtyping as well (Section 5).

We share motivations and objectives with Neubauer and Thiemann [24] and with Pucella and Tov [29] although we adopt a different approach. In short: [24,29] hinge on advanced features of the host language (Haskell) to represent and handle conventional session types; we work with a representation of session types that makes them easy to handle in conventional type systems. Compared to [24,29], our approach realizes (F.1–3) with a substantially simpler machinery that better integrates with the host language and is less onerous on the programmer. A more in-depth discussion is deferred to Section 6.

The OCaml implementation of FuSe communication primitives, described in full in Section 5, can also be downloaded from the author's home page. *Proofs and supplementary technical material are in Appendixes A and B, beyond the page limit.*

Table 1. Syntax of FuSe expressions and processes.

Expression	$e ::=$	$\mathtt{c}$	(constant)
	$\mid$	u	(name)
	$\mid$	$\mathtt{fun}\ x \to e$	(abstraction)
	$\mid$	$e_1 e_2$	(application)
	$\mid$	(e_1, e_2)	(pair construction)
	$\mid$	$\mathtt{K}\ e$	(sum injection)
	$\mid$	$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$	(let binding)
	$\mid$	$\mathtt{let}\ x, y = e_1\ \mathtt{in}\ e_2$	(pair splitting)
	$\mid$	$\mathtt{match}\ e\ \mathtt{with}\ \{i\ x_i \to e_i\}_{i=\mathtt{L},\mathtt{R}}$	(case analysis)
Process	$P, Q ::=$	$\langle e \rangle$	(thread)
	$\mid$	$P \mid Q$	(composition)
	$\mid$	$(\nu a)P$	(session)
	$\mid$	$\mathtt{error}$	(runtime error)

2 Syntax and Semantics of FuSe

We use infinite sets of *variables* $x, y, \ldots$ and *sessions* $a, b, \ldots$; an *endpoint* or *channel* is a pair a^p made of a session a and a *polarity* $p, q \in \{+, -, *\}$. Polarities $+$ and $-$ denote *valid endpoints* that can be used for I/O operations whereas the polarity $*$ denotes *invalid endpoints* that are not supposed to be used. We define a partial involution $\overline{\cdot}$ on polarities such that $\overline{+} = -$ and $\overline{-} = +$ and leave $\overline{*}$ undefined. We say that a^p is the *peer* of $a^{\overline{p}}$ when $p \neq *$. We let u range over *names*, which are either variables or endpoints.

The syntax of *expressions* e and *processes* P, Q is given in Table 1. The symbol $\mathtt{c}$ ranges over the constants $\mathtt{()}$, $\mathtt{fix}$, $\mathtt{fork}$, $\mathtt{create}$, $\mathtt{close}$, $\mathtt{send}$, $\mathtt{receive}$, $\mathtt{left}$, $\mathtt{right}$, $\mathtt{branch}$, where $\mathtt{()}$ is the unitary value, $\mathtt{fix}$ the fixpoint combinator, and $\mathtt{fork}$ the primitive that creates new threads. The remaining constants represent communication primitives whose semantics will be detailed shortly. Expressions are conventional and include $\mathtt{let}$ bindings and introduction and elimination constructs for products and sums. The symbol $\mathtt{K}$ ranges over the tags $\mathtt{L}$ and $\mathtt{R}$ that inject values into disjoint sums. We use expressions to model threads – the sequential parts of programs – and processes to model parallel threads communicating via sessions. A process is either a thread $\langle e \rangle$, or the parallel composition $P \mid Q$ of two processes P and Q, or a restriction $(\nu a)P$ modeling a session a with scope P, or a runtime $\mathtt{error}$ resulting from an attempt to use an invalid endpoint. The notions of free and bound names are standard. We write $\mathsf{fn}(e)$ and $\mathsf{fn}(P)$ respectively for the sets of free names of e and P and we identify terms modulo alpha-renaming of bound names.

The operational semantics is defined in terms of a reduction relation for expressions, a structural congruence and a labeled reduction relation for processes. We make use of conventional notions of *values* v, w and of *evaluation contexts* $\mathscr{E}$, defined thus:

$$v, w ::= \mathtt{c} \mid a^p \mid \mathtt{fun}\ x \to e \mid (v, w) \mid \mathtt{K}\ v \mid \mathtt{fork}\ v \mid \mathtt{send}\ v$$

$$\begin{aligned}\mathscr{E} ::= {} & [\,] \mid \mathscr{E}\, e \mid v\mathscr{E} \mid (\mathscr{E}, e) \mid (v, \mathscr{E}) \mid \mathtt{K}\ \mathscr{E} \mid \mathtt{let}\ x = \mathscr{E}\ \mathtt{in}\ e \\ \mid {} & \mathtt{let}\ x, y = \mathscr{E}\ \mathtt{in}\ e \mid \mathtt{match}\ \mathscr{E}\ \mathtt{with}\ \{i\ x_i \to e_i\}_{i=\mathtt{L},\mathtt{R}}\end{aligned}$$

Table 2. Reduction of FuSe expressions and processes.

Reduction of expressions $\boxed{e \longrightarrow e'}$

$$
\begin{array}{lr}
\text{[R-BETA]} & (\texttt{fun}\, x \to e)\, v \longrightarrow e\{v/x\} \\
\text{[R-LET]} & \texttt{let}\, x = v\, \texttt{in}\, e \longrightarrow e\{v/x\} \\
\text{[R-FIX]} & \texttt{fix}\, v \longrightarrow v\ (\texttt{fix}\, v) \\
\text{[R-SPLIT]} & \texttt{let}\, x,y = (v,w)\, \texttt{in}\, e \longrightarrow e\{v,w/x,y\} \\
\text{[R-MATCH]} & \texttt{match}\ \mathtt{K}\ v\ \texttt{with}\ \{i\ x_i \to e_i\}_{i\in\{\mathtt{L},\mathtt{R}\}} \longrightarrow e_{\mathtt{K}}\{v/x_{\mathtt{K}}\}
\end{array}
$$

Reduction of processes $\boxed{P \xrightarrow{\ell} Q}$

$$
\begin{array}{lrll}
\text{[R-THREAD]} & \langle \mathscr{E}[e] \rangle \xrightarrow{\tau} \langle \mathscr{E}[e'] \rangle & & \text{if } e \longrightarrow e' \\
\text{[R-FORK]} & \langle \mathscr{E}[\texttt{fork}\ v\ w] \rangle \xrightarrow{\tau} \langle \mathscr{E}[()] \rangle \mid \langle vw \rangle & & \\
\text{[R-CREATE]} & \langle \mathscr{E}[\texttt{create}()] \rangle \xrightarrow{\tau} (\nu a)\langle \mathscr{E}[(a^+, a^-)] \rangle & & a \text{ fresh} \\
\text{[R-CLOSE]} & \langle \mathscr{E}[\texttt{close}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{close}\ a^{\overline{p}}] \rangle \xrightarrow{\mathtt{c}a} \langle \mathscr{E}_{\mathtt{c}a}[()] \rangle \mid \langle \mathscr{E}'_{\mathtt{c}a}[()] \rangle & & \\
\text{[R-COMM]} & \langle \mathscr{E}[\texttt{send}\ a^p\ v] \rangle \mid \langle \mathscr{E}'[\texttt{receive}\ a^{\overline{p}}] \rangle \xrightarrow{\mathtt{m}ap} \langle \mathscr{E}_{\mathtt{m}ap}[a^p] \rangle \mid \langle \mathscr{E}'_{\mathtt{m}ap}[(v_{\mathtt{m}ap}, a^{\overline{p}})] \rangle & & \\
\text{[R-LEFT]} & \langle \mathscr{E}[\texttt{left}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{branch}\ a^{\overline{p}}] \rangle \xrightarrow{\mathtt{m}ap} \langle \mathscr{E}_{\mathtt{m}ap}[a^p] \rangle \mid \langle \mathscr{E}'_{\mathtt{m}ap}[\mathtt{L}\ a^{\overline{p}}] \rangle & & \\
\text{[R-RIGHT]} & \langle \mathscr{E}[\texttt{right}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{branch}\ a^{\overline{p}}] \rangle \xrightarrow{\mathtt{m}ap} \langle \mathscr{E}_{\mathtt{m}ap}[a^p] \rangle \mid \langle \mathscr{E}'_{\mathtt{m}ap}[\mathtt{R}\ a^{\overline{p}}] \rangle & & \\
\text{[R-ERROR]} & \langle \mathscr{E}[\mathtt{c}\ a^*] \rangle \xrightarrow{\tau} \texttt{error} & & \\
\text{[R-PAR]} & P \mid R \xrightarrow{\ell} Q \mid R_\ell & & \text{if } P \xrightarrow{\ell} Q \\
\text{[R-NEW-1]} & (\nu a)P \xrightarrow{\tau} (\nu a)Q & & \text{if } P \xrightarrow{\mathtt{m}ap} Q \text{ or } P \xrightarrow{\mathtt{c}a} Q \\
\text{[R-NEW-2]} & (\nu a)P \xrightarrow{\ell} (\nu a)Q & & \text{if } P \xrightarrow{\ell} Q \text{ and } a^p \notin \mathsf{fn}(\ell) \\
\text{[R-STRUCT]} & P \xrightarrow{\ell} Q & & \text{if } P \equiv P' \xrightarrow{\ell} Q' \equiv Q
\end{array}
$$

Note that `fork` v and `send` v are values because `fork` and `send` represent curried binary functions. Evaluation contexts are standard for call-by-value; as usual, we write $\mathscr{E}[e]$ for the result of replacing the hole $[\,]$ in $\mathscr{E}$ with e. We also write $X\{v/u\}$ for the capture-avoiding substitution of v in place of the free occurrences of u in X, where X stands for an expression, a process, or an evaluation context.

The reduction relations are defined in Table 2. Reduction of expressions, in the upper part of the table, is standard. The lower part of the table defines a labeled reduction for processes where *labels* ℓ are either τ, denoting an internal action, or $\mathtt{m}ap$, denoting a message exchange from endpoint a^p to endpoint $a^{\overline{p}}$ in session a, or $\mathtt{c}a$, indicating that the session a has been closed. We define $\mathsf{fn}(\tau) \stackrel{\text{def}}{=} \emptyset$ and $\mathsf{fn}(\mathtt{m}ap) = \mathsf{fn}(\mathtt{c}a) \stackrel{\text{def}}{=} \{a^+, a^-\}$. Labels allow us to observe the behavior of processes on the channels they use. This information is necessary to show that well-typed processes respect protocols and also to *invalidate* the endpoints that have been used in a reduction.

Notation. For every expression, process, context X, we write X_ℓ for $X\{a^*/u\}_{u\in\mathsf{fn}(\ell)}$.

Intuitively, X_ℓ invalidates all the endpoints a^p in X by replacing them with a^*, after an ℓ-labeled reduction. When $\ell = \tau$, no (observable) endpoint is used so no endpoint is invalidated. When $\ell = \mathtt{m}ap$ or $\ell = \mathtt{c}a$, both a^+ and a^- are invalidated.

We now describe the reduction rules of processes. Rule [R-THREAD] simply lifts the reduction of expressions to processes. Rule [R-FORK] spawns a new thread, represented as

a function v that needs an argument w. The thread is started by applying v to w in parallel with the process that forks the thread. Rule [R-CREATE] creates a new session a. The expression `create()` reduces to a pair containing two valid endpoints of the session with opposite polarities. Rule [R-CLOSE] models the closing of session a. The operation invalidates every occurrence of a^+ and a^- in the program. In the formal development this operation is synchronous for simplicity, but in the implementation each endpoint can be closed independently of the other. Rule [R-COMM] models the communication between two threads connected by a session a. The message v in the sender thread is transferred to the receiving thread. Following the semantics of [14], the output operation reduces to the endpoint a^p used by the sender, while the input operation reduces to a pair containing the message and the endpoint $a^{\overline{p}}$ used by the receiver. All other occurrences of a^p and $a^{\overline{p}}$, including those in the message v, are invalidated. Next are [R-LEFT] and [R-RIGHT] modeling the selection of a particular branch in a structured conversation. They are akin to [R-COMM], except that the endpoint used by the receiver is injected into a disjoint sum with a tag `K` that represents the choice taken. Intuitively, the message being communicated in these cases is just the tag `K`. Note that there is no explicit creation or passing of continuations in [R-COMM], [R-LEFT], [R-RIGHT]: all that is transferred from one thread to another is just the payload. Rule [R-ERROR] generates a runtime error if there is an attempt to use an invalid endpoint, where "using an endpoint" means that the endpoint occurs as the first argument of a primitive `c`. Rule [R-PAR] closes reductions under parallel compositions. The label that decorates the reduction relation is used to propagate the effects of an invalidation to the entire program. Rules [R-NEW-1] and [R-NEW-2] close reductions under restrictions. The labels $\mathtt{m}ap$ and $\mathtt{c}a$ turn into a τ when they cross the restriction on a, as the session becomes unobservable. Finally, [R-STRUCT] closes reductions under the least structural congruence defined by the rules below:

$$\langle () \rangle \mid P \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$(\nu a)\langle () \rangle \equiv \langle () \rangle \qquad (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \qquad \frac{a^+, a^-, a^\star \notin \mathsf{fn}(Q)}{(\nu a)P \mid Q \equiv (\nu a)(P \mid Q)}$$

Structural congruence is essentially the same of the π-calculus, except that the idle process is written $\langle () \rangle$ and the last rule shrinks or extends the scope of a session only when no valid or invalid endpoint is captured.

Example 1 (mathematical server). We write examples in a language slightly richer than FuSe that includes numbers, booleans, `if-then-else`, and standard syntactic sugar for possibly recursive `let`-bindings. The examples compile and run using `OCaml` [22] and our implementation of the FuSe primitives, described in Section 5. Below is a simple server for mathematical operations, similar to that of [13]:

```
let rec server x =
  match branch x with                  (* wait for a request       *)
    `L x → close x                     (* close session            *)
  | `R x → let n, x = receive x in     (* receive first operand    *)
           let m, x = receive x in     (* receive second operand   *)
           let x = send x (n + m) in   (* send result              *)
           server x                    (* serve more requests      *)
```

The server is modeled as a function operating on an endpoint `x`. The function is recursive, so that the server is able to process an arbitrary number of requests within a single session. The server first waits for a request from the client, represented as a tag `'L` or `'R` (this is the `OCaml` syntax for polymorphic variant tags, see Section 5.3). If the client selects `'L`, the session is closed. If the client selects `'R`, the server expects to receive two integer numbers, it sends back their sum, and recurs.

A possible client, operating on an endpoint `y`, is shown below:

```
let client n y =
  let rec aux acc n y =                (* add n naturals           *)
    if n = 0 then begin
      close (left y); acc              (* close session and return *)
    end else
      let y = right y in               (* select plus operation    *)
      let y = send y acc in            (* send first operand       *)
      let y = send y n in              (* send second operand      *)
      let res, y = receive y in        (* receive result           *)
      aux res (n - 1) y                (* possibly add more        *)
  in aux 0 n y
```

In this case, the client is computing the sum of the first `n` naturals by repeated invocations of the server. If `n` is 0, the computation is over, the client closes the session and returns the result. Otherwise, the client invokes the "plus" operation offered by the server to compute a new partial result and recurs. Note how the endpoint `y` is used differently in the two branches of the `if-then-else`.

The code to fire up client and server is the following

```
let a, b = create () in              (* create the session       *)
let _ = Thread.create server a in    (* spawn the server         *)
print_int (client 100 b)             (* run the client           *)
```

where `Thread.create`, a function provided by the standard `OCaml` library for multithreading, corresponds to the `fork` primitive in `FuSe`. ■

3 Types

In this section we define the types for `FuSe`, we recall the encoding of [9] and extend it to an isomorphism between session types and a suitable subset of `FuSe` types. This gives us the basis for interpreting and understanding the type of `FuSe` communication primitives.

Types for FuSe. We let $\alpha, \beta, \ldots$ range over *type variables*. The syntax of (finite) *types* and of *type schemes* is given in Table 3. Type schemes σ are conventional; we will often abbreviate $\forall\alpha_1 \cdots \forall\alpha_n.t$ with $\forall\alpha_1 \cdots \alpha_n.t$. Types t, s are the regular trees generated by the type constructors in Table 3 and include the unitary type `unit`, arrows, products, and disjoint sums. In the examples we occasionally use other base types such as `int` and `bool`. The types $\circ$ and $\bullet$ respectively denote the *absence* and *presence* of a certain input/output capability in a channel type. They are not inhabited and are only used as

Table 3. Syntax of types, type schemes, and session types.

Type	$t,s ::= \alpha$	(variable)	**Type scheme**	$\sigma ::= t$	(mono type)
	$\mid \circ$	(no cap.)		$\mid \forall\alpha.\sigma$	(poly type)
	$\mid \bullet$	(one cap.)			
	$\mid$ `unit`	(unit)	**Session type**	$T,S ::=$ `end`	(termination)
	$\mid t \to s$	(arrow)		$\mid$ `?`$T.S$	(input)
	$\mid t * s$	(product)		$\mid$ `!`$T.S$	(output)
	$\mid t + s$	(sum)		$\mid T \,\&\, S$	(external choice)
	$\mid {}^{tt}[t]$	(channel)		$\mid T \oplus S$	(internal choice)

phantom type parameters. A channel type ${}^{t_i t_o}[s]$ has three type parameters: t_i and t_o respectively represent the *input* and *output* capabilities of the channel and are always either a type variable or $\circ$ or $\bullet$; s is the type of the messages that can be sent on/received from the channel. For example, ${}^{\circ\bullet}$`[int]` is a channel type without input capability ($\circ$) and with output capability ($\bullet$), hence it denotes a channel for sending messages of type `int`; the type ${}^{\bullet\circ}$`[int *` ${}^{\circ\bullet}$`[bool]]` denotes a channel for receiving a pair whose first component has type `int` and whose second component is another channel that can be used for sending a `bool`. A channel with type ${}^{\circ\circ}[t]$ bears no capabilities and cannot be used for I/O operations. Channel types ${}^{\bullet\bullet}[t]$ have no role in `FuSe`.

Type equality corresponds to regular tree equality. Recall that each regular tree consists of a finite number of distinct subtrees and admits finite representations as a system of equations or using the traditional μ notation ([8] is the standard reference for regular trees and their finite representations). For example, the equation $\alpha =$ `int` $\to \alpha$ is satisfied by the type t of the "ogre" function that eats infinitely many `int` arguments. The shape of the equation, with α guarded by a type constructor on the right hand side, effectively defines the unique type t such that $t =$ `int` $\to t$ (see [8, Theorem 4.3.1]). We can use infinite types for describing arbitrarily long communication protocols, like the one implemented by `server` and `client` in Example 1.

Duality relates the types of channels used in complementary ways:

Definition 1 (type duality). *Let* $\bot_{\mathtt{t}}$ *be the least relation such that* ${}^{t_1 t_2}[s] \;\bot_{\mathtt{t}}\; {}^{t_2 t_1}[s]$. *We write* $t^{\bot}$ *for the* s *such that* $t \;\bot_{\mathtt{t}}\; s$ *when* t *is a channel type;* $t^{\bot}$ *is undefined otherwise.*

The rationale for type duality is that if a process uses a channel according to some type t, then we expect another process to use the same channel according to the dual type $t^{\bot}$. So for example, ${}^{\circ\bullet}$`[int]` $\bot_{\mathtt{t}}$ ${}^{\bullet\circ}$`[int]` since the dual behavior of "send an `int`" is "receive an `int`". On the other hand, we have ${}^{\circ\circ}[t] \;\bot_{\mathtt{t}}\; {}^{\circ\circ}[t]$, since "do nothing" is dual of itself. Type duality is a partial involution: when $t^{\bot}$ is defined, we have $t^{\bot\bot} = t$.

Our representation of channel capabilities is a bit unusual. Most type systems with channel types, from the seminal paper on channel subtyping [28] to those for the linear π-calculus [21] and binary sessions [9], use types of the form $\kappa[t]$ or variants of this, where κ ranges over a finite set of capabilities such as $\{\mathtt{?}, \mathtt{!}, \emptyset\}$. We have used this notation also in Section 1, while discussing program `Q`. In these cases, capabilities and types belong to different sorts and computing the dual of a channel type essentially means defining a suitable dual operator for capabilities such that, for instance, $\mathtt{?}^{\bot} = \mathtt{!}$.

One shortcoming of this representation is that duality is easily defined only when capabilities are known. Dealing with unknown capabilities means introducing (possibly dualized) capability variables, and this machinery quickly taints the whole type system. Our approach departs from the aforementioned ones in two respects. First, we use *two* slots for representing the absence or presence of a certain I/O capability, instead of just one slot that contains either one or the other. This representation, which is also convenient in type reconstruction algorithms for the π-calculus [19,26,27], allows us to dualize a channel type by just swapping the content of the two slots. Second, we use types to represent capabilities so that type variables can stand for unknown capabilities. For example, the type $t \stackrel{\text{def}}{=} {}^{\alpha\beta}[\gamma]$ denotes a channel for which nothing is known. Nonetheless, the dual of t can still be obtained by swapping α and β, that is $t^{\perp} = {}^{\beta\alpha}[\gamma]$. Overall, the chosen representation makes it easy to relate a channel type and its dual even when capabilities are not known.

In the following we will make extensive use of channel types with unknown capabilities and unknown message types, so we reserve some convenient notation for them:

Notation (channel type variable)**.** We say that a type of the form ${}^{\alpha\beta}[\gamma]$ is a *channel type variable*. We let A, B, ... range over channel type variables and we write $\forall A\,.\,\sigma$ instead of $\forall\alpha\beta\gamma\,.\,\sigma$ when $A = {}^{\alpha\beta}[\gamma]$.

Note that the dual $A^{\perp}$ of a channel type variable A is always defined.

Session types. Even though our type system does not use built-in session types, the typing of `FuSe` communication primitives follows from the encoding of session types into ordinary types [9]. For this reason, in the remainder of this section we formalize the relationship between session types and `FuSe` types, recalling the encoding and instantiating it to our setting. Compared to [9], we use channel types with a slightly different representation of I/O capabilities and we consider possibly infinite (session) types.

The syntax of *session types* T, S is given in Table 3. The session type `end` denotes a channel on which no further communication is allowed. The session types $?T\,.\,S$ and $!T\,.\,S$ denote channels to be used respectively for one input and one output of a message of type T and according to S afterwards. The session types $T \,\&\, S$ and $T \oplus S$ respectively denote external and internal choices in a protocol. A process using a channel of type $T \oplus S$ decides whether to behave according to the "left" protocol T or the "right" protocol S. A process using a channel of type $T \,\&\, S$ accepts the decision of the process using the peer endpoint. As we have seen in Table 2, the choice is effectively encoded and transmitted as an appropriate message, hence $\oplus$ corresponds to an output and $\&$ to an input operation. As for types, we do not devise a concrete syntax for recursive session types and use regular trees in this case as well. For example, the (unique) session type T that satisfies the equality $T = !T_1\,.\,!T_1\,.\,?T_2\,.\,T$ denotes a channel for sending two messages of type T_1, receiving one message of type T_2, and then according to the same protocol, over and over again. The given syntax of session types disallows the description of polymorphic protocols and protocols for exchanging messages other than channels. These limitations are immaterial for we introduce session types for illustrative purposes only. All the results in this section extend to more general forms of session types.

Just like channel types, session types too support a notion of duality that relates complementary behaviors. It is defined thus:

Definition 2 (session type duality). Session type duality *is the largest relation* $\bot_{\mathsf{st}}$ *between session types that satisfies the rules*

$$\texttt{end} \bot_{\mathsf{st}} \texttt{end} \qquad \frac{S_1 \bot_{\mathsf{st}} S_2}{\texttt{?}T\,.\,S_1 \bot_{\mathsf{st}} \texttt{!}T\,.\,S_2} \qquad \frac{T_i \bot_{\mathsf{st}} S_i \ ^{(i=1,2)}}{T_1 \,\&\, T_2 \bot_{\mathsf{st}} S_1 \oplus S_2}$$

and the symmetric ones, omitted. Observe that $\bot_{\mathsf{st}}$ *is an endofunction on session types. We write* $T^\bot$ *for the session type* S *such that* $T \bot_{\mathsf{st}} S$ *and say that* S *is the* dual *of* T.

Duality relates inputs with outputs carrying the same message type and `end` with itself. For example, $\texttt{?}T\,.\,\texttt{!}S\,.\,\texttt{end} \bot_{\mathsf{st}} \texttt{!}T\,.\,\texttt{?}S\,.\,\texttt{end}$ and if T is the session type such that $T = \texttt{!}T_1\,.\,\texttt{!}T_1\,.\,\texttt{?}T_2\,.\,T$ then $T^\bot$ is the session type S such that $S = \texttt{?}T_1\,.\,\texttt{?}T_1\,.\,\texttt{!}T_2\,.\,S$. It is easy to establish that duality is an involution also for session types ($T^{\bot\bot} = T$).

We now formalize the claim made in Section 1 that "session types and their encoding describe the same protocol, written in different ways" as an isomorphism between the set $\mathbb{S}$ of session types and a suitable subset $\mathbb{P}$ of `FuSe` types which we call protocol types (such essentially syntactic isomorphism between types is supported by a semantic correspondence between processes, see [9]). The set $\mathbb{P}$ is defined thus:

Definition 3. *We write* $\mathbb{P}$ *for the largest subset of types such that* $t \in \mathbb{P}$ *implies either* $t = {}^{\circ\circ}[\texttt{unit}]$ *or* $t = {}^{s_1 s_2}[t_1 \odot t_2]$ *and* $\odot \in \{*, +\}$ *and* $\{s_1, s_2\} = \{\circ, \bullet\}$ *and* $t_1, t_2 \in \mathbb{P}$.

The morphism from $\mathbb{S}$ to $\mathbb{P}$ is given by the encoding of session types into ordinary types [9] and rests on the idea that multiple communications on one channel can be modeled as a sequence of one-shot communications on a chain of different channels. The chain is realized by sending, at each communication, a fresh continuation channel along with the communication payload. For example, the session type $\texttt{!}T\,.\,S$ describes a channel used for sending a message of type T first and according to S afterwards. It is encoded as the channel type ${}^{\circ\bullet}[t * s]$ where t is the encoding of T and s is the encoding of $S^\bot$. The reason why the type s of the continuation is the encoding of $S^\bot$ and not the encoding of S is because the tail S in $\texttt{!}T\,.\,S$ describes the behavior of the *sender* after it has sent a message of type T, while in the encoding the type of the continuation describes the behavior of the *receiver* of the continuation. Clearly, the sender will also use the same continuation, but according to the type $s^\bot$. In general we have:

Definition 4 (encoder). *The* encoder *function* $[\![\cdot]\!] : \mathbb{S} \to \mathbb{P}$ *is coinductively defined by:*

$$[\![\texttt{end}]\!] = {}^{\circ\circ}[\texttt{unit}] \qquad \begin{array}{ll} [\![\texttt{?}T\,.\,S]\!] = {}^{\bullet\circ}[[\![T]\!] * [\![S]\!]] & [\![T \,\&\, S]\!] = {}^{\bullet\circ}[[\![T]\!] + [\![S]\!]] \\ [\![\texttt{!}T\,.\,S]\!] = {}^{\circ\bullet}[[\![T]\!] * [\![S^\bot]\!]] & [\![T \oplus S]\!] = {}^{\circ\bullet}[[\![T^\bot]\!] + [\![S^\bot]\!]] \end{array}$$

As an example, if we consider again $T = \texttt{!}T_1\,.\,\texttt{!}T_1\,.\,\texttt{?}T_2\,.\,T$, then we derive:

$$\begin{array}{lll} [\![T]\!] = {}^{\circ\bullet}[t_1 * [\![\texttt{?}T_1\,.\,\texttt{!}T_2\,.\,T^\bot]\!]] & & \text{where } t_1 = [\![T_1]\!] \\ \quad = {}^{\circ\bullet}[t_1 * {}^{\bullet\circ}[t_1 * [\![\texttt{!}T_2\,.\,T^\bot]\!]]] & & \\ \quad = {}^{\circ\bullet}[t_1 * {}^{\bullet\circ}[t_1 * {}^{\circ\bullet}[t_2 * [\![T^{\bot\bot}]\!]]]] & & \text{where } t_2 = [\![T_2]\!] \\ \quad = {}^{\circ\bullet}[t_1 * {}^{\bullet\circ}[t_1 * {}^{\circ\bullet}[t_2 * [\![T]\!]]]] & & \end{array}$$

If we consider instead the session type S such that $S = \texttt{?}T_1\,.\,\texttt{?}T_1\,.\,\texttt{!}T_2\,.\,S$, which is in fact $T^\bot$, then $[\![S]\!] = {}^{\bullet\circ}[t_1 * {}^{\bullet\circ}[t_1 * {}^{\circ\bullet}[t_2 * [\![T]\!]]]]$. The choice of ${}^{\circ\circ}[\texttt{unit}]$ as the encoding of `end` is almost arbitrary. We could have used any type in place of `unit`.

The morphism from $\mathbb{P}$ to $\mathbb{S}$ reconstructs a session type T by interpreting the type of continuation channels as the tail(s) of T:

Definition 5 (decoder). *The* decoder *function* $\langle\!\langle \cdot \rangle\!\rangle : \mathbb{P} \to \mathbb{S}$ *is coinductively defined by:*

$$\langle\!\langle {}^{\circ\circ}[\texttt{unit}] \rangle\!\rangle = \texttt{end} \qquad \begin{array}{ll} \langle\!\langle {}^{\bullet\circ}[t * s] \rangle\!\rangle = ?\langle\!\langle t \rangle\!\rangle . \langle\!\langle s \rangle\!\rangle & \langle\!\langle {}^{\bullet\circ}[t + s] \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle \,\&\, \langle\!\langle s \rangle\!\rangle \\ \langle\!\langle {}^{\circ\bullet}[t * s] \rangle\!\rangle = !\langle\!\langle t \rangle\!\rangle . \langle\!\langle s^{\perp} \rangle\!\rangle & \langle\!\langle {}^{\circ\bullet}[t + s] \rangle\!\rangle = \langle\!\langle t^{\perp} \rangle\!\rangle \oplus \langle\!\langle s^{\perp} \rangle\!\rangle \end{array}$$

It is not immediate to see that $\langle\!\langle \cdot \rangle\!\rangle = [\![\cdot]\!]^{-1}$, because the two morphisms use different notions of *duality*, for session types and for channel types respectively. However, as observed in [9] and formally stated below, $[\![\cdot]\!]$ commutes with duality (and so does $\langle\!\langle \cdot \rangle\!\rangle$). This property is key to prove that $[\![\cdot]\!]$ and $\langle\!\langle \cdot \rangle\!\rangle$ are indeed one the inverse of the other.

Theorem 1 (commuting duality). $\perp_{\mathsf{t}} \circ [\![\cdot]\!] = [\![\cdot]\!] \circ \perp_{\mathsf{st}}$.

The existence of an isomorphism between $\mathbb{S}$ and $\mathbb{P}$ shows that using protocol types instead of built-in session types results in no loss of expressiveness (there is an encoding for every session type) and no loss in precision (every session type can be reconstructed from its encoding). Most importantly, Theorem 1 combined with our representation of channel types provides a straightforward method for checking whether $T \perp_{\mathsf{st}} S$ holds. Suppose for example that $[\![T]\!] = {}^{t_i t_o}[t]$ and $[\![S]\!] = {}^{s_i s_o}[s]$. Using Theorem 1 we deduce

$$T \perp_{\mathsf{st}} S \iff [\![T]\!] \perp_{\mathsf{t}} [\![S]\!] \iff t_i = s_o \wedge t_o = s_i \wedge t = s$$

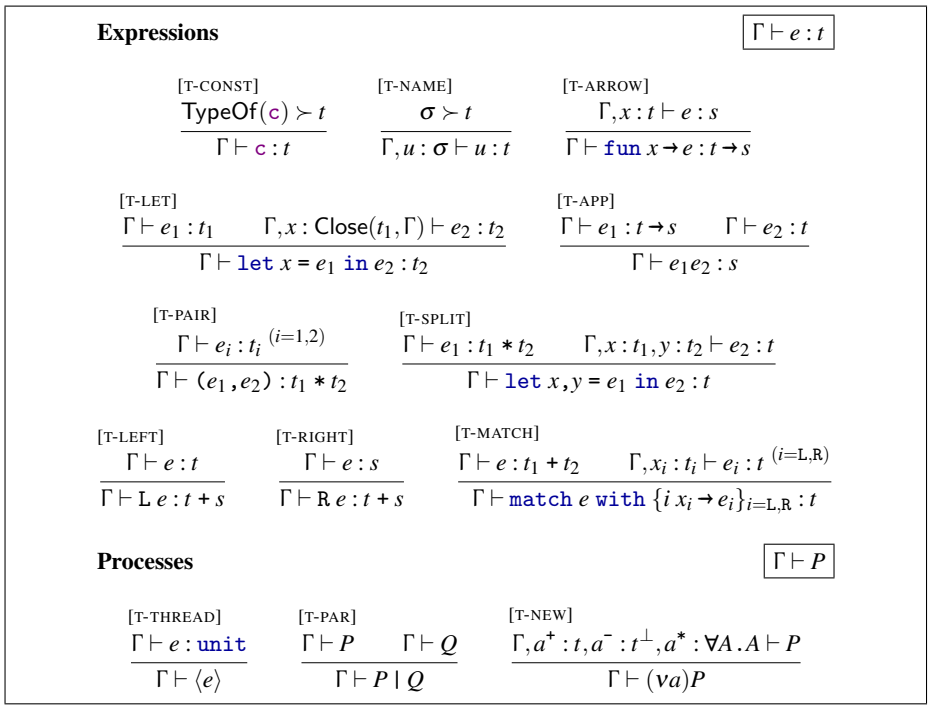
thereby turning the verification of a complex relation $T \perp_{\mathsf{st}} S$, which implies matching input with output capabilities across the whole structure of T and S, into three plain type equalities. In prospect of integrating a session type system into an existing type system, this is a major advantage of using encoded (as opposed to built-in) session types.

4 Type System

We present the type system for FuSe and state its properties. The type system is essentially standard for ML-like languages, in particular it has no baked-in features specifically targeted to session type checking. Compared to the type system in [14], the main differences concern the typing of communication primitives and the fact that the type system is not substructural.

Table 4 gives the typing rules. We let Γ range over *type environments* which are finite maps from names to type schemes written $u_1 : \sigma_1, \ldots, u_n : \sigma_n$ that keep track of the type of the free names of expressions and processes. We write $\emptyset$ for the empty type environment, $\mathsf{dom}(\Gamma)$ for the domain of Γ, and Γ, Γ' for the union of Γ and Γ' when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$. The rules for processes derive judgments of the form $\Gamma \vdash P$, stating that P is well typed in Γ. Rules [T-THREAD] and [T-PAR] are standard. Rule [T-NEW] introduces in the type environment three endpoints of a session: two of them are valid and typed with dual types (the fact that one of them is typed by $t^{\perp}$ implicitly means that t is a channel type); the third one is invalid and typed with $\forall A.A$. This way, distinct occurrences of an invalid endpoint can appear anywhere a channel is expected and need not be typed in the same way (see Remark 1). There is no typing rule for `error`.

Table 4. Typing rules for expressions and processes.



The rules for expressions derive judgments of the form $\Gamma \vdash e : t$ and are formulated using the same notation of Wright and Felleisen [32]. Since the rules are mostly standard, we just focus on a few details. Rules [T-CONST] and [T-NAME] respectively type constants and names by instantiating their type scheme. The type scheme of constants is retrieved by a global function $\mathsf{TypeOf}(\cdot)$, to be detailed shortly, while that of names is obtained from the type environment. Following [32], the relation $\sigma \succ t$ is defined by

$$t \succ t \qquad \frac{\sigma \succ t}{\forall \alpha\, .\, \sigma \succ t\{s/\alpha\}}$$

and instantiates a type scheme into a type. Rule [T-LET] generalizes the type of the `let`-bound variable by means of the function $\mathsf{Close}(\cdot)$, which is defined as in [32] by

$$\mathsf{Close}(t, \Gamma) \stackrel{\text{def}}{=} \forall \alpha_1 \cdots \alpha_n\, .\, t \qquad \text{where } \{\alpha_1, \ldots, \alpha_n\} = \mathsf{ftv}(t) \setminus \mathsf{ftv}(\Gamma)$$

where ftv collects the free type variables of types and type environments. Rule [T-LET] is well known for being unsound in *impure* languages, the best-known counterexample being that of polymorphic references (again, see [32]). However, the counterexample relies crucially on the fact that the *same* reference is used *twice*, in such a way that its type scheme can be instantiated with incompatible types in different parts of the program. [T-LET] is sound if we know that x is used linearly. Since the impure fragment

Table 5. Type schemes of FuSe constants.

`()` : `unit`	`send` : $\forall\alpha A.\,{}^{\circ\bullet}[\alpha * A] \to \alpha \to A^{\perp}$
`fix` : $\forall\alpha.\,(\alpha \to \alpha) \to \alpha$	`receive` : $\forall\alpha A.\,{}^{\bullet\circ}[\alpha * A] \to \alpha * A$
`fork` : $\forall\alpha.\,(\alpha \to \texttt{unit}) \to \alpha \to \texttt{unit}$	`left` : $\forall AB.\,{}^{\circ\bullet}[A + B] \to A^{\perp}$
`create` : $\forall A.\,\texttt{unit} \to A * A^{\perp}$	`right` : $\forall AB.\,{}^{\circ\bullet}[A + B] \to B^{\perp}$
`close` : ${}^{\circ\circ}[\texttt{unit}] \to \texttt{unit}$	`branch` : $\forall AB.\,{}^{\bullet\circ}[A + B] \to A + B$

of FuSe concerns only sessions and we are interested in stating the soundness of FuSe type system under the assumption that channels are indeed used linearly, we can live with just one typing rule for `let` and not impose the value restriction even if e_1 has side effects (Appendix B.1 details why the counterexample in [32] does not apply).

The function TypeOf is given in Table 5 as a set of associations `c` : TypeOf(`c`). The types of `()` and `fix` are standard. The type of `fork` has been chosen to match more closely the one of `Thread.create` in the `OCaml` multithreading module. According to the operational semantics, `fork` accepts a function representing the thread to be created and the argument of type α it needs to start executing. The type of `create` makes it clear that the primitive returns a pair of endpoints with dual types. Recall that a channel type variable like A is just syntactic sugar for a channel type of the form ${}^{\alpha\beta}[\gamma]$. Therefore, the desugared type scheme of `create` is $\forall\alpha\beta\gamma.\,\texttt{unit} \to {}^{\alpha\beta}[\gamma] * {}^{\beta\alpha}[\gamma]$. The ability to express channel types with unknown message types and capabilities gives `create` the most general type. The type of `close` is unremarkable. The type of `send` follows from the encoding of outputs (Definition 4): `send` takes a channel for sending messages of type $\alpha * A$, the payload of type α, and returns a channel of type $A^{\perp}$. According to its type, `send` should in principle communicate both the payload *and* the continuation. In reality, as the operational semantics illustrates, only the payload is sent. The type A of the continuation is used to correlate the future behaviors of sender and receiver after this interaction. The type of `receive` follows from the encoding of inputs: in this case the type of the continuation describes how the channel will be used by the receiver process, once the message has arrived. The types of `left` and `right` are analogous to that of `send`, and the type of `branch` is analogous to that of `receive`.

Observe that all the types of FuSe primitives can be expressed in any type system with generic types, once channel type variables have been desugared (Notation 3).

Example 2. Below we propose again the code of `server` in Example 1 in which we have indicated the type s_i of the (free) occurrence of `x` on line i:

```
1  let rec server x =
2    match branch x with                (* s2 = •°[s3 + s4]    *)
3      `L x → close x                   (* s3 = °°[unit]       *)
4    | `R x → let n, x = receive x in   (* s4 = •°[int * s5]   *)
5             let m, x = receive x in   (* s5 = •°[int * s6]   *)
6             let x = send x (n + m) in (* s6 = °•[int * s7]   *)
7             server x                  (* s7 = s2⊥            *)
```

The output on line 6 indicates that `server` sends a payload of type `int` and a (virtual) continuation of type s_7 to `client`. Therefore, s_7 describes the behavior of

`client` on the continuation channel, whereas `server` will use the same channel according to the type s_2, which is the dual of s_7. Overall, the argument `x` of `server` has type $s = {}^{\bullet\circ}[{}^{\circ\circ}[\texttt{unit}] + {}^{\bullet\circ}[\texttt{int} * {}^{\bullet\circ}[\texttt{int} * {}^{\circ\bullet}[\texttt{int} * s^{\perp}]]]]$ therefore we have $\texttt{server} : s \to \texttt{unit}$. It is then easy to derive $\texttt{client} : \texttt{int} \to s^{\perp} \to \texttt{int}$, confirming that `client` and `server` can interact flawlessly. ■

We now investigate the relationship between well-typed programs and the three standard properties of sessions: every message sent in a session has the expected type (*communication safety*); the sequence of interactions in a session follows the prescribed protocol (*protocol fidelity*); if the interaction in a session stops, there are no pending I/O operations (*progress*). Obviously, well typing alone is not enough to guarantee these properties, for two reasons: first, the FuSe type system does not enforce the linear usage of endpoints, therefore there exist well-typed programs that try to use endpoints non-linearly causing runtime errors; second, the FuSe type system does not prevent deadlocks, which jeopardize progress and may occur even if endpoint linearity is respected. To take these facts into account, we must weaken the statements of our results with additional hypotheses: that endpoints are used linearly, and that no deadlocks occur. Note that these properties are undecidable in general.

The semantics of our communication primitives allows for a simple definition of *linear endpoint usage*. Recall that each communication primitive applied to an endpoint a^p *invalidates* every other occurrence of a^p before (possibly) returning a^p itself. Therefore, any attempt to use an invalid endpoint means that another occurrence of the same endpoint has already been used in the past. It is not enough to check endpoint validity at one particular point in time, for instance in the initial program state, for an endpoint might be duplicated as the program executes. We resort to a coinductive definition that requires linear endpoint usage to be preserved across all possible executions of a process.

Definition 6 (affine and linear endpoint usage). *Let* EA *and* EL *be the largest predicates on processes that are closed by reductions and such that:*

- *If* $\mathsf{EA}(P)$ *or* $\mathsf{EL}(P)$ *and* $P \equiv (\nu a_1) \cdots (\nu a_n)(\langle \mathscr{E}[\mathtt{c}\ a^p] \rangle \mid Q)$*, then* $p \in \{+, -\}$.
- *If* $\mathsf{EL}(P)$ *and* $P \equiv (\nu a)Q$ *and* $a^p \in \mathsf{fn}(Q)$ *where* $p \in \{+, -\}$*, then* $a^{\overline{p}} \in \mathsf{fn}(Q)$.

In words, EA is the set of *endpoint affine processes*, which never try to use the same endpoint twice, while EL is the set of *endpoint linear processes* which, in addition, never discard a valid endpoint if its peer is being used. Note that $\mathsf{EL} \subseteq \mathsf{EA}$ and that EL is *coarser* than the property enforced by linear type systems. In particular, duplications of an endpoint are allowed provided that only valid endpoints are actually used. For example, the expression

```
send (let x, y = (a, a) in x)
```

temporarily duplicates the endpoint `a` but may occur in a process that satisfies EL. The same expression is ill typed according to the type system in [14].

In order to state subject reduction we have to consider that, like in many other behavioral type systems, the type associated with endpoints may change as the result of interactions occurring on such endpoints. To express this change, we define a suitable reduction relation for type environments mimicking that of processes (Table 2).

Definition 7. *Let* $\xrightarrow{\ell}$ *be the least relation between type environments such that:*

$$\begin{array}{rl}
\Gamma \xrightarrow{\tau} \Gamma & \\
\Gamma, a^p : {}^{\circ\bullet}[t * s], a^{\overline{p}} : {}^{\bullet\circ}[t * s] \xrightarrow{\mathtt{m}ap} \Gamma, a^p : s^{\perp}, a^{\overline{p}} : s & \\
\Gamma, a^p : {}^{\circ\bullet}[t_1 + t_2], a^{\overline{p}} : {}^{\bullet\circ}[t_1 + t_2] \xrightarrow{\mathtt{m}ap} \Gamma, a^p : t_i^{\perp}, a^{\overline{p}} : t_i & i \in \{1,2\} \\
\Gamma, a^p : {}^{\circ\circ}[\mathtt{unit}], a^{\overline{p}} : {}^{\circ\circ}[\mathtt{unit}] \xrightarrow{\mathtt{c}a} \Gamma &
\end{array}$$

We write $\Gamma \xrightarrow{\ell}$ *if there exists* Γ' *such that* $\Gamma \xrightarrow{\ell} \Gamma'$ *and* $\Gamma \not\xrightarrow{\ell}$ *if not* $\Gamma \xrightarrow{\ell}$.

Observe that $\Gamma \xrightarrow{\mathtt{m}ap}$ implies $\Gamma \not\xrightarrow{\mathtt{m}a\overline{p}}$ and $\Gamma \not\xrightarrow{\mathtt{c}a}$. That is, if communication from a^p to $a^{\overline{p}}$ is allowed at some point of an interaction in session a, communication in the opposite direction is forbidden, as is closing a, at the same point. Similarly, $\Gamma \xrightarrow{\mathtt{c}a}$ implies $\Gamma \not\xrightarrow{\mathtt{m}a+}$ and $\Gamma \not\xrightarrow{\mathtt{m}a-}$ (in a closing session a no communication is allowed) and $\Gamma \xrightarrow{\mathtt{c}a} \xrightarrow{\ell}$ implies $a^+, a^- \notin \mathsf{fn}(\ell)$ (once session a has been closed, no more actions are allowed).

The last ingredient we need to state subject reduction is that of balanced type environment: Γ balanced if, whenever there is an association for some valid endpoint a^p in Γ, then there are associations also for its peer $a^{\overline{p}}$ and for a^* as well, with the requirement that peer endpoints must have dual types and invalid ones have type $\forall A.A$. Formally:

Definition 8 (balanced type environment). *We say that* Γ *is* balanced *if:*

1. $a^p \in \mathsf{dom}(\Gamma)$ *with* $p \in \{+,-\}$ *implies* $a^{\overline{p}}, a^* \in \mathsf{dom}(\Gamma)$ *and* $\Gamma(a^p) \perp_{\mathsf{t}} \Gamma(a^{\overline{p}})$*;*
2. $a^* \in \mathsf{dom}(\Gamma)$ *implies* $\Gamma(a^*) = \forall A.A$.

Remark 1. To appreciate the relevance of condition (2) in Definition 8 consider

$$\begin{array}{rl}
P \stackrel{\text{def}}{=} & \langle \mathtt{close}\ (\mathtt{send}\ a^+\ 42)\rangle \mid \langle \mathtt{let}\ _, x = \mathtt{receive}\ a^-\ \mathtt{in}\ \mathtt{close}\ x\rangle \\
\mid & \langle \mathtt{close}\ (\mathtt{send}\ (\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ c^+\ \mathtt{else}\ a^+)\ 31)\rangle
\end{array}$$

and the environment $\Gamma \stackrel{\text{def}}{=} a^+, c^+ : t, a^-, c^- : t^{\perp}, a^*, c^* : \forall A.A$ where $t \stackrel{\text{def}}{=} {}^{\circ\bullet}[\mathtt{int} * {}^{\circ\circ}[\mathtt{unit}]]$. Observe that P is well typed in Γ and that $\mathsf{EA}(P)$ holds despite P contains two occurrences of a^+, because a^+ is never actually *used* twice. We have

$$\begin{array}{rl}
P \xrightarrow{\mathtt{m}a+} & \langle \mathtt{close}\ a^+\rangle \mid \langle \mathtt{let}\ _, x = (42, a^-)\ \mathtt{in}\ \mathtt{close}\ x\rangle \\
\mid & \langle \mathtt{close}\ (\mathtt{send}\ (\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ c^+\ \mathtt{else}\ a^*)\ 31)\rangle
\end{array}$$

where one occurrence of a^+ has been invalidated and

$$\Gamma \xrightarrow{\mathtt{m}a+} a^+, a^- : {}^{\circ\circ}[\mathtt{unit}], c^+ : t, c^- : t^{\perp}, a^*, c^* : \forall A.A \stackrel{\text{def}}{=} \Gamma'$$

Note that a^+ and c^+ have the same type in Γ and incompatible types in Γ'. If the type of a^* could not be instantiated with an arbitrary channel type (t in this case), then the residual process would be ill typed in Γ'. ■

Theorem 2 (subject reduction). *If* $\Gamma \vdash P$ *where* Γ *is balanced and* $\mathsf{EA}(P)$ *and* $P \xrightarrow{\ell} Q$*, then there exists* Γ' *such that* $\Gamma \xrightarrow{\ell} \Gamma'$ *and* $\Gamma' \vdash Q$.

Protocol fidelity follows immediately from Theorem 2 and the observations below Definition 7: if an ℓ-labeled reduction cannot be performed by a type environment Γ, then it cannot be performed by an endpoint affine process that is well-typed in Γ. Communication safety is a straightforward consequence of typing and is formalized below. Note that endpoint affinity suffices for proving both safety and fidelity.

Proposition 1 (safety). *Let* $\Gamma \vdash P$ *and* $\mathsf{EA}(P)$. *Then:*

1. *if* $P \equiv \langle \mathscr{E}[\mathtt{send}\ u\ v] \rangle \mid Q$, *then* $\Gamma(u) = {}^{\circ\bullet}[t * s]$ *and* $\Gamma \vdash v : t$;
2. *if* $P \equiv \langle \mathscr{E}[\mathtt{c}\ u] \rangle \mid Q$ *and* $\mathtt{c} \in \{\mathtt{left}, \mathtt{right}\}$, *then* $\Gamma(u) = {}^{\circ\bullet}[t + s]$.

Concerning progress, we first give a syntactic characterization of deadlock:

Definition 9 (deadlock). *We say that* P *is* deadlocked *if*

$$P \equiv (\nu a_1) \cdots (\nu a_n) \textstyle\prod_{i \in I} \langle \mathscr{E}_i[\mathtt{c}_i\ c_i^{p_i}] \rangle$$

where $I \neq \emptyset$ *and for every* $i \in I$ *there exists* $j \in I$ *such that* $c_i^{\overline{p}_i} \in \mathsf{fn}(\mathscr{E}_j)$ *and* $\mathtt{c}_i \in \{\mathtt{close}, \mathtt{send}, \mathtt{receive}, \mathtt{left}, \mathtt{right}, \mathtt{branch}\}$.

Intuitively, in a deadlocked process all threads are blocked on input/output operations and the peer of the (valid) endpoint in each of such operations occurs guarded by another blocked operation. A well-typed, endpoint linear process P enjoys a partial form of progress: if P cannot reduce anymore and is not deadlocked, then P has no pending I/O operations on open sessions.

Theorem 3 (partial progress). *If* $\emptyset \vdash P$ *and* $\mathsf{EL}(P)$, *then either there exists* Q *such that* $P \xrightarrow{\tau} Q$ *or* $P \equiv \langle () \rangle$ *or* P *is deadlocked.*

We conclude the section discussing a representative range of errors that go undetected by the type system.

Example 3 (deadlocks). Below are some typical examples of endpoint linear programs that eventually deadlock.

```
let program_A =
  let worker x y =
    let z, x = receive x in
    let y = send y z in
    close x; close y in
  let a, b = create () in
  let c, d = create () in
  fork (worker a) d;
  fork (worker c) b
```

```
let program_B =
  let a, b = create () in
  let n, a = receive a in
  let b = send n in
  close a; close b

let program_C =
  let a, b = create () in
  close (send a b)
```

In `program_A` we have two threads connected by two distinct sessions, each thread waits to receive a message from the other one before sending its own. In `program_B` one thread attempts to use the same session for receiving and sending a message sequentially. Finally, in `program_C` a thread is sending on channel `a` its peer `b`. This example is typeable by giving to `b` the infinite type $t = {}^{\bullet\circ}[t * {}^{\circ\circ}[\mathtt{unit}]]$ and to `a` its dual $t^{\perp}$.

In all these cases, static detection of the eventual deadlock requires stronger typing disciplines that either prevent the creation of cyclic network topologies [6,31,23] or rely on non-trivial extensions of session types [3,4,25]. ■

Example 4 (linearity violations). The condition $\mathsf{EA}(P) \wedge \neg\mathsf{EL}(P)$ indicates that P respects endpoint affinity but discards valid endpoints that may be necessary in order to have progress. For example, the following well-typed program

```
let a, b = create () in close (send a 42)
```

discards `b` and reduces to a stuck configuration which is *not* a deadlock. A compiler might give notice of unused value declarations like `b` in this example, but it would likely stay quiet if `b` is replaced by an anonymous pattern `_` (`OCaml` behaves like this).

The condition $\neg\mathsf{EA}(P)$ indicates that P attempts to use an invalid endpoint. This happens if the endpoint is used more than once, in a way that disrespects the explicit threading of continuations required by the communication primitives. Two instances of this event, which we call *overlap*, are illustrated below:

```
let foo x =
  let _ = send x 42 in
  let x = send x 43 in (* ↯ *)
  close x
```

```
let bar y =
  let _ = send y 42 in
  let _, y = receive y in (* ↯ *)
  close y
```

The function `foo` can be typed giving `x` type $\circ\bullet$`[int *` $\circ\circ$`[unit]]`, even though the second `send` overlaps with the first. A similar problem occurs in `bar`, where `receive` overlaps with `send`. However, the overlap in `bar` is detected by the type system, because attempting to send and receive a message using the same `y` requires `y` to have incompatible capability annotations ($\circ\bullet$ for `send`, $\bullet\circ$ for `receive`). All the overlaps of `send` and `left`/`right` or of `receive` and `branch` are also detected because the `*` constructor in the types of `send` and `receive` is incompatible with the `+` constructor in the types of `left`/`right`. Overall, the only overlaps that can go undetected are those concerning multiple uses of the same communication primitive with the same message types. Undetected overlaps are subtle, though, since their effects are generally unpredictable. In Section 5.2 we will discuss how overlaps can be detected at runtime. ■

5 Implementation

We describe the `OCaml` module that implements the `FuSe` primitives for session communication. We start with a basic version of the module (Section 5.1) which we then extend with runtime detection of invalid endpoint usage (Section 5.2) and generalized choices (Section 5.3). The extensions are easy for shared-memory processes. We also discuss whether and how they scale to a distributed setting.

5.1 The Basics

The `OCaml` module that realizes the `FuSe` communication primitives is shown in full in Figure 1. The interface exports the abstract types $\circ$ and $\bullet$ (lines 1–2) and an abstract channel type (line 3). In `OCaml`, the channel type ${}^{t_i}\!{}^{t_o}[s]$ is written (t_i, t_o, s) `t` and the sum type $t + s$ becomes the polymorphic variant type `[‘L of` t `| ‘R of` s`]`. We use polymorphic variants [12] because they easily generalize sums to arbitrary tags and

Interface

```
1  type ∘           (* absent I/O capability  *)
2  type •           (* present I/O capability *)
3  type (α,β,φ) t   (* channel type           *)
4  val create     : unit → (α,β,φ) t * (β,α,φ) t
5  val close      : (∘,∘,unit) t → unit
6  val send       : (∘,•,φ * (α,β,ψ) t) t → φ → (β,α,ψ) t
7  val receive    : (•,∘,φ * (α,β,ψ) t) t → φ * (α,β,ψ) t
8  val left       : (∘,•,[‘L of (α,β,φ) t | ‘R of (γ,δ,ψ) t]) t → (β,α,φ) t
9  val right      : (∘,•,[‘L of (α,β,φ) t | ‘R of (γ,δ,ψ) t]) t → (δ,γ,ψ) t
10 val branch     : (•,∘,[‘L of (α,β,φ) t | ‘R of (γ,δ,ψ) t] as ε) t → ε
```

Implementation

```
11 type ∘           (* no representation *)
12 type •           (* no representation *)
13 type (α,β,φ) t = φ Event.channel
14 let create ()  = let u = Event.new_channel () in (u, u)
15 let close _    = ()
16 let send u x   = Event.sync (Event.send u (Obj.magic x)); Obj.magic u
17 let receive u  = Obj.magic (Event.sync (Event.receive u), u)
18 let left u     = Event.sync (Event.send u (Obj.magic ‘L)); Obj.magic u
19 let right u    = Event.sync (Event.send u (Obj.magic ‘R)); Obj.magic u
20 let branch u   = Obj.magic (Event.sync (Event.receive u), u)
```

Fig. 1. Interface and implementation of the `OCaml` module for session communications.

support a form of subtyping that is consistent with subtyping for session types [13]. We will see these features at work in Section 5.3.

The types of the primitives (lines 4–10) are essentially syntactic variations of those shown in Table 5, so we only make a couple of remarks. First, as in `FuSe` types, we can switch from one channel type to its dual by the mere flipping of its first two type parameters (see *e.g.* the type of `create` on line 4). Second, in the type of `branch` (line 10), the type expression t `as` ε denotes the same type as t and creates an alias ε that stands for t itself. Such construction has several uses: here, it is handy to refer to the same variant type in the codomain of `branch` without rewriting the whole type. Since t `as` ε binds ε also *within* t, the same construction is also used in `OCaml` for creating recursive types. We will see an instance of this feature at work in Example 5.

We have based the implementation of the primitives on the `Event` module in `OCaml`'s standard library, which provides an API for communication and synchronization in the style of Concurrent `ML` [30]. The `Event` module has been chosen out of mere convenience; our primitives can be built on top of any minimal API for message passing. In the `Event` module, the type t `Event.channel` denotes a channel for exchanging messages of type t and the functions `Event.send` and `Event.receive`, instead of performing communications directly, construct *communication events*. In order for com-

munication to actually take place, both the sender and the receiver must *synchronize* by applying `Event.sync` to such events.

The representation of a channel type `(α,β,φ) t` is a `φ Event.channel` (line 13), namely channels in `FuSe` are `Event.channels` in `OCaml`. Note that α and β play no role in the representation of channel types; they are meant to be instantiated with $\circ$ and $\bullet$ which have no data constructors (lines 11–12). Polarities are not represented either, they are an artifact of the formal model so that peer endpoints can be typed differently.

The implementation of `create` (line 14) and `close` (line 15) is dull: the first creates an `Event` channel and returns a pair with two references (with dual types) to it; the second does nothing (`OCaml`'s garbage collector automatically reclaims unused channels).

Concerning the implementation of `send` (line 16), we have to keep in mind that the `Event.channel` underlying our endpoints expects messages that, in principle, contain both the payload `x` as well as the continuation endpoint `u`, but we only communicate the payload `x`. For this reason, we cast `x` using `Obj.magic` so that `x` appears to the type checker as having the type of the pair `(x, u)`. This cast cannot compromise the correct functioning of the `Event` module: since the `Event.channel` type is *parametric* in the type of messages, `Event` functions cannot make any assumption on their concrete representation. The value returned by `send` is the same reference `u` used for the communication, except that its type is cast to the dual type of the continuation. The trickery in `send` forces a corresponding implementation of `receive` (line 17): `OCaml` believes that the event created by `Event.receive` yields a pair consisting of a payload and a continuation, whereas only the former is actually received. We explicitly pair the endpoint `u` (which is known to the receiver) to the payload, and we perform another cast so that the pair is typed correctly.

The implementation of `left`, `right`, and `branch` (lines 18–20) follows the same lines. In these cases, only a tag `‘L` or `‘R` is communicated, instead of the continuation channel `u` injected through one of such tags as the type of `left` and `right` suggests. The injection is performed on the receiver's side and resorts to one last magic: since the internal `OCaml` representation of `‘K u` – that is channel `u` injected through the `K` tag – is the same as that of the pair `(‘K, u)`, we create such a pair and cast its type to that of the injected channel. This trick spares us one pattern matching on the tag and scales to arbitrary tag sets (see Section 5.3).

Example 5 (session type inference and duality). Below are the types of `server` and `client` from Example 1 that `OCaml` infers automatically when these functions are linked against the module that implements `FuSe` primitives:

```
val server :
  (•,○,[ ‘R of (•,○,int * (•,○,int * (○,•,int * (○,•,α) t) t) t) t
       | ‘L of (○,○,unit) t ] as α) t → unit
val client :
  int →
  (○,•,[ ‘R of (•,○,int * (•,○,int * (○,•,int * (○,•,α) t) t) t) t
       | ‘L of (○,○,unit) t ] as α) t → int
```

These two types correspond exactly to those we have guessed in Example 2. Since the channel type in the type of `server` is dual of the channel type in the type of `client`,

```
1  exception InvalidEndpoint
2  type ∘                (* no representation *)
3  type •                (* no representation *)
4  type (α,β,φ) t      = φ Event.channel * bool ref
5  let check v         = if not (compare_and_swap v true false) then
6                          raise InvalidEndpoint
7  let fresh u         = (u, ref true)
8  let create ()       = let u = Event.new_channel () in (fresh u, fresh u)
9  let close (_, v)    = check v
10 let send (u, v) x   = check v; ...; Obj.magic (fresh u)
11 let receive (u, v)  = check v; Obj.magic (..., fresh u)
12 let left (u, v)     = check v; ...; Obj.magic (fresh u)
13 let right (u, v)    = check v; ...; Obj.magic (fresh u)
14 let branch (u, v)   = check v; Obj.magic (..., fresh u)
```

Fig. 2. Implementation of session communications with invalid endpoint detection.

the program of Example 1 is well typed and `client` and `server` interact safely. Consider now the following variation of `client`

```
let client' n u =
  let rec aux acc n u =   (* acc : float            *)
    ...                   (* same code as in client *)
  in aux 0.0 n u          (* 0.0 : float            *)
```

where the initial value for the partial result `acc` is `0.0` instead of `0`, therefore turning `acc`'s type from `int` to `float`. Taken in isolation, `client'` is well typed and `OCaml` infers the following type for it:

```
val client' :
  int →
  (∘,•,[ `R of (•,∘,float * (•,∘,int * (∘,•,float * (∘,•,α) t) t) t) t
       | `L of (∘,∘,unit) t ] as α) t → float
```

However, the channel types of `client'` and `server` are not dual of each other (the corresponding message types are not unifiable). `OCaml` detects this problem and fails to compile a program that connects `client'` and `server` with a session. ■

5.2 Runtime Detection of Invalid Endpoint Usages

Figure 2 extends our module with the endpoint invalidation semantics of `FuSe` so that an exception (declared on line 1) is raised whenever an invalid endpoint is used. Note that invalidation in the formal model is a rather powerful mechanism that acts *atomically* on all the occurrences of an endpoint in a possibly distributed program. The code in Figure 2 implements the invalidation semantics assuming that processes have access to a shared memory. The idea is to represent endpoints as pairs consisting of an `Event.channel` and a mutable flag indicating whether the endpoint is valid or not (line 4). The flag effectively approximates the endpoint polarity, with the difference

that it only distinguishes between valid and invalid endpoints. Whenever a thread attempts to use an endpoint, the flag in the pair is checked first: if the flag is `true`, then the endpoint is valid and can be used; if the flag is `false`, then the endpoint has already been used and an exception is raised. The auxiliary function `check` (lines 5–6) implements this behavior. Checking that the flag is `true` and setting it to `false` must be performed atomically, for concurrent threads may attempt to use the same endpoint simultaneously. Therefore, we realize `check` using a conventional `compare_and_swap` operation, whose implementation is undetailed.

Ideally, when a communication primitive returns a continuation endpoint, the flag associated with the endpoint should be restored to `true`, but doing so on the existing pair might induce other users of the endpoint into thinking that the reference they own is valid, while in fact it is not. The idea is that communication primitives return a *fresh* pair that contains the same `Event.channel` in the old pair with a *fresh* flag reset to `true`. This refreshing of pairs is implemented by the auxiliary function `fresh` (line 7). In essence, the cost we pay for detecting the usage of invalid endpoints is the allocation of a new pair and a `bool` reference at each invocation of a communication primitive.

With this setup, the communication primitives can be implemented by prefixing them with a call to `check` and wrapping the returned `Event.channel`(s) with `fresh` (lines 8–14). In Figure 2 we have elided with `...` the unchanged code fragments from Figure 1. Observe that `check` and `refresh` remain confined within the module, which exports the same interface it had before, plus the `InvalidEndpoint` exception.

The naive generalization of this mechanism to a distributed setting requires maintining the consistency of the flag associated with each endpoint across different locations and is clearly unfeasible (recall that endpoints can be communicated in messages). Nonetheless, the mechanism can be adapted to a distributed setting if we assume that communicated endpoints are always meant to be used by the receiver. Otherwise, an endpoint could be sent in a message and simultaneously retained and used by the sender, making it unusable by the receiver. The idea is that an endpoint being sent in a message is invalidated in the sender, electing the receiver as the only owner of a valid reference to the endpoint. As a result, across the whole distributed system, there is always at most one location containing valid references to the roaming endpoint, and linearity violations within such location can be efficiently detected using the code shown in Figure 2. This mechanism can be implemented either by runtime inspection of exchanged messages, or by means of a dedicated primitive for sending endpoints (like `throw` in [16]). Both possibilities are very reasonable, given that the communication of endpoints in a distributed environment is likely to require some special handling anyway.

5.3 Generalized Choices

Although binary choices suffice to model protocols with an arbitrary branching structure, being able to use multiple tags, with possibly meaningful names, is desirable. The main challenge with generalizing choices to arbitrary tags is that the tags appear explicitly in the types of `left`, `right`, and `branch`, whereas we would like the interface of our library to be as general as possible. One solution is to replace `left` and `right` with a generic selection primitive `select` and revise `branch` so that `select` and `branch` have these types:

```
val select : (○,●,[>] as φ) t → ((α,β,ψ) t → φ) → (β,α,ψ) t
val branch : (●,○,[>] as φ) t → φ
```

The semantics of `branch` is simply to receive a message of type φ. The semantics of `select` is similar to that of `send`, except that `send` takes a message ready to be sent, whereas `select` takes a function of type (α,β,ψ) `t` $\to \varphi$ which produces the message, of type φ, when applied to a continuation endpoint of type (α,β,ψ) `t`. Typically, such function will be the η-expansion of a tag

```
fun x → ‘Tag x
```

that injects a continuation channel into a polymorphic variant type.

The type expression `[>] as` φ in the types of `select` and `branch` indicates that φ can only be instantiated with a polymorphic variant type. This constraint is crucial for the safety of the library: leaving φ unconstrained would make the type (α,β,φ) `t` unifiable with the type $(\alpha,\beta,\psi *(\gamma,\delta,\varepsilon)$ `t)` `t` and an ordinary message sent with `send` could be received with `branch` as if it were a label, or a label selected with `select` could be received with `receive` as if it were an ordinary message.

The implementation of `select` and `branch` is similar to that of `send` and `receive`, with the difference that `select` transfers the function over the channel, camouflaging the function as if it were the message produced by the function:

```
let select u f = Event.sync (Event.send u (Obj.magic f)); Obj.magic u
let branch u   = Obj.magic (Event.sync (Event.receive u)) u
```

This handling of arbitrary tags does not scale well to a distributed setting, because sending a function over a channel assumes that sender and receiver share the same address space and trust each other. It is not clear, however, if a really general solution exists in this case. After all, the tags occurring in a protocol are domain-specific, and it might be reasonable for applications to provide specialized versions of `select` and/or `branch` that involve the transfer of only the tag, as we have done for `left` and `right`.

Example 6. Below is a revised and extended version of Example 1 where the mathematical server supports three operations identified by the tags `‘Quit`, `‘Plus`, and `‘Eq` and the client uses `select` to choose the appropriate ones.

```
let rec server x =
  match branch x with
    ‘Quit x → close s
  | ‘Plus x →
      let n, x = receive x in
      let m, x = receive x in
      let x = send x (n + m) in
      server s
  | ‘Eq x →
      let n, x = receive x in
      let m, x = receive x in
      let x = send x (n = m) in
      server s
```

```
let client n y =
  let rec aux acc n y =
    if n = 0 then begin
      let y =
        select y (fun x → ‘Quit x)
      in close u; acc
    end else
      let y =
        select y (fun x → ‘Plus x)
      in let y = send y acc in
      let y = send y n in
      let res, y = receive y in
      aux res (n - 1) u
  in aux 0 n u
```

It is instructive to look at the types inferred by OCaml for these two functions:

```
val server :
  (•,○,[< ‘Eq of (•,○,β * (•,○,β * (○,•,bool * (○,•,α) t) t) t) t
       |  ‘Plus of (•,○,int * (•,○,int * (○,•,int * (○,•,α) t) t) t) t
       |  ‘Quit of (○,○,unit) t ] as α) t → unit
val client :
  int → ((○,•,[> ‘Plus of (•,○,int * (•,○,int * (○,•,int * α) t) t) t
              |  ‘Quit of (○,○,unit) t ]) t as α) → int
```

Notice that server is parametric in the type β of the operands of the ‘Eq operation, as a consequence of the fact that equality is polymorphic in OCaml. Also, the type of x is not exactly the dual of the type of y, because the choice in one type has three tags ‘Eq, ‘Plus, and ‘Quit while the other one has only two. The question then is whether OCaml is able to infer that client and server interact successfully, despite this mismatch in the types of the endpoints they use. This is indeed the case, and the reason lies in the < and > symbols that decorate variant types. The < symbol indicates a *closed* variant type, one for which the set of tags constitutes an *upper bound*: the match in the server body handles three tags ‘Eq, ‘Plus, ‘Quit, but not others. The > symbol indicates an *open* variant type, one for which the set of tags constitutes a *lower bound*: the ‘Plus and ‘Quit tags may be produced by client, but this variant type is unifiable with others providing a superset of tags, like the one in the type of x. In conclusion, the rules governing variant types allow OCaml to infer that client and server interact successfully because client needs only a subset of the operations provided by server. If client attempted to use a ‘Mult operation, OCaml would signal an error at the point where client and server are connected through a session.

The fact that the revised server interacts correctly with client, despite the type of x is not exactly dual to that of y, is formally explained in terms of *subtyping* for session types [13]: the dual of the type of x is a subtype of the type of y, meaning that client uses fewer features than those offered by server. We exploit once more the encoding of session types into ordinary types to lift OCaml's built-in subtyping of variant types at the level of channel (and therefore protocol) types. ■

6 Concluding Remarks

Inspired by the encoding of session types into (linear) channel types [9], we have shown how to realize some key features of a session type system in terms of ordinary features of any type system with generic types. The choice of OCaml for our proof-of-concept implementation allowed us to showcase the full potential of the approach in exchange for the least effort. Nonetheless, the approach is applicable to a broad range of programming languages, albeit with varying degrees of integration and/or convenience.

Substructural type systems are becoming increasingly popular in theoretical models of programming languages, but they are (still) rare in practice. This fact motivated our quest for an alternative handling of linearity violations that could be easily implemented as part of our library for session communications. The typing discipline resulting from our approach is nonetheless able to detect a number of linearity violations (Example 4)

and, if the host language supports affine/linear types, the typing of the communication primitives can be easily refined to statically enforce affine/linear endpoint usage.

The choice of synchronous communication in both the formal model and the implementation was motivated by convenience. Asynchronous communication can be modeled like in [14] using explicit queues, adjusting the formal semantics so that the peers of a session are invalidated independently, and basing the implementation on a suitable asynchronous API. The use of a single primitive `create` to open new sessions has been inspired by Singularity OS [18,3]. Most session calculi and languages provide a pair of `accept`/`request` primitives to establish sessions via shared channels [16,14]. Shared channels, `accept`, and `request` do not pose particular challenges and the implementation (available online) features them already.

A more substantial extension concerns *multiparty sessions* [17], those involving an arbitrary, possibly fixed number of participants. It has been shown that some classes of multiparty sessions can be realized in terms of binary sessions connecting pairs of participants (see [25, extended version] and [5]). As it stands, our approach could deal with each binary session in isolation, but would be unable to recognize the sessions as part of a single multiparty session. Whether the approach can be extended to model "true" multiparty sessions remains an open question.

Related work. Our work aims at the same objectives as [24,29], but follows a substantially different approach. We focus the comparison on linearity and type representation.

The typing disciplines proposed in [24,29] rely on monads to simultaneously track the changes in the types of endpoints (F.3) and enforce their linear usage (F.4). We realize (F.3) using the same technique as [14] and rely on the runtime system to detect those linearity violations that may compromise safety. The monadic approach gives stronger static guarantees concerning linearity, but has a cost in terms of either expressiveness or usability: in [24], monadic computations can involve a single channel only; in [29], channels (or, better, their capabilities) are encapsulated and stacked in the monad, and the programmer must write explicit monadic actions that literally dig into the stack to reach the channel/capability to be used. The provided linearity guarantees weaken to affinity in presence of exceptions, which are a known challenge for substructural type systems. We give up on static detection of (all) linearity violations in favor of a lighter and more open-ended API. Not committing to a specific mechanism, our approach can immediately benefit from native support for affine/linear typing from the host language, if available, or can be complemented by a monadic API in the style of [29], if desired.

Both [24] and [29] propose a faithful modeling of session types as sequences of I/O actions and internal/external choices, whereas we work with session types encoded into ordinary types [9]. Our approach improves the results of [24,29] in various respects. Duality is not addressed in [24] and is expressed in [29] using rather sophisticated mechanisms, such as multiparameter type classes and functional dependencies or explicitly provided duality proofs. We have shown that none of these mechanisms is necessary: duality for encoded session types can be expressed in terms of type equality in any type system with generic types. In general, the encoding favors a smoother integration of (encoded) session types within built-in features of the host language. This is clear by looking at the handling of choices and recursion. In Section 5 we have used `OCaml` polymorphic variants to model choices, but we could have used plain algebraic data types as

well. In `Scala` it might be reasonable to use case classes, and in languages like `Java` or `C++` one could rely on specific class hierarchies. In summary, encoded internal/external choices can be modeled using idiomatic features of the host language, favoring the integration with native notions of subtyping when available (Example 6). We can make similar observations concerning recursion. In [29], recursive session types are represented using de Brujin indexes and type-level Peano numerals and require the programmer to write explicit monadic actions for entering/invoking recursions. We have shown that none of these mechanisms is necessary if the host language features recursive types: the encoding lifts native recursive types to recursive session types transparently (Example 5) and allows the programmer to write recursive/iterative code according to the language style (Example 1).

We conclude observing that the runtime detection of linearity violations is somehow related to the runtime monitoring of session communications [7,2,11,1]. Runtime monitoring is achieved either by a service [7,2,11] or by an active communication middleware [1] that compares the observable behavior of processes against the declared contracts/session types and possibly issues notifications when violations are detected. Like monitoring, our runtime mechanism is meant to ensure communication safety and protocol fidelity. Unlike monitoring, our mechanism is *internal* to processes and only detects linearity violations, which would not necessarily imply corresponding protocol violations in their observable behavior.

References

1. Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In *Proceedings of FORTE'15*, LNCS 9039, pages 161–177. Springer, 2015.
2. Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *Proceedings of FMOODS/-FORTE'13*, LNCS 7892, pages 50–65. Springer, 2013.
3. Viviana Bono and Luca Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012.
4. Viviana Bono, Luca Padovani, and Andrea Tosatto. Polymorphic Types for Leak Detection in a Session-Oriented Functional Language. In *Proceedings of FORTE'13*, LNCS 7892, pages 83–98. Springer, 2013.
5. Luís Caires and Jorge A. Pérez. A typeful characterization of multiparty structured conversations based on binary sessions. Technical report, 2014. Available at `http://arxiv.org/abs/1407.4242`.
6. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of CONCUR'10*, LNCS 6269, pages 222–236. Springer, 2010.
7. Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *Proceedings of TGC'11*, LNCS 7173, pages 25–45. Springer, 2011.
8. Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comp. Sci.*, 25:95–169, 1983.
9. Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of PPDP'12*, pages 139–150. ACM, 2012.
10. Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *Proceedings of CONCUR'11*, LNCS 6901, pages 280–296. Springer, 2011.

11. Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods in System Design*, 46(3):197–225, 2015.
12. Jacques Garrigue. Programming with polymorphic variants. In *Informal proceedings of ACM SIGPLAN Workshop on ML*, 1998.
13. Simon Gay and Malcolm Hole. Subtyping for Session Types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
14. Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.
15. Kohei Honda. Types for dyadic interaction. In *Proceedings of CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.
16. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.
17. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL'08*, pages 273–284. ACM, 2008.
18. Galen Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.
19. Atsushi Igarashi and Naoki Kobayashi. Type Reconstruction for Linear π-Calculus with I/O Subtyping. *Inf. and Comp.*, 161(1):1–44, 2000.
20. Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, LNCS 2757, pages 439–453. Springer, 2002. Extended version at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.
21. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
22. Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 4.02 - Documentation and user's manual*, 2014. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/`.
23. Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *Proceedings of ESOP'15*, LNCS 9032, pages 560–584. Springer, 2015.
24. Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proceedings of PADL'04*, LNCS 3057, pages 56–70. Springer, 2004.
25. Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proceedings of CSL-LICS'14*, pages 72:1–72:10. ACM, 2014. Extended version available at `http://hal.archives-ouvertes.fr/hal-00932356v2/document`.
26. Luca Padovani. Type reconstruction for the linear π-calculus with composite and equirecursive types. In *Proceedings of FoSSaCS'14*, LNCS 8412, pages 88–102. Springer, 2014.
27. Luca Padovani, Tzu-Chun Chen, and Andrea Tosatto. Type Reconstruction Algorithms for Deadlock-Free and Lock-Free Linear π-Calculi. In *Proceedings of COORDINATION'15*, volume 9037 of *LNCS*, pages 83–98. Springer, 2015.
28. Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
29. Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of HASKELL'08*, pages 25–36, New York, NY, USA, 2008. ACM.
30. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
31. Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.
32. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. and Comput.*, 115(1):38–94, 1994.

A Supplement to Section 3

In order to clarify what we mean by "coinductively defined function" we flesh out a more rigorous definition of the encoder function.

Definition 10 (encoder). *Let $\mathscr{C}$ be the largest relation between session types and types such that $T \mathscr{C} t$ implies either:*

- $T = \mathtt{end}$ *and* $t = {}^{\circ\circ}[\mathtt{unit}]$, *or*
- $T = ?T_1 \,.\, T_2$ *and* $t = {}^{\bullet\circ}[t_1 * t_2]$ *and* $T_i \mathscr{C} t_i$ *for every* $i = 1,2$, *or*
- $T = !T_1 \,.\, T_2$ *and* $t = {}^{\circ\bullet}[t_1 * t_2]$ *and* $T_1 \mathscr{C} t_1$ *and* $T_2^{\perp} \mathscr{C} t_2$, *or*
- $T = T_1 \,\&\, T_2$ *and* $t = {}^{\bullet\circ}[t_1 + t_2]$ *and* $T_i \mathscr{C} t_i$ *for every* $i = 1,2$, *or*
- $T = T_1 \oplus T_2$ *and* $t = {}^{\circ\bullet}[t_1 + t_2]$ *and* $T_i^{\perp} \mathscr{C} t_i$ *for every* $i = 1,2$.

Observe that $\mathscr{C}$ is a function from session types to types, for $T \mathscr{C} t_1$ and $T \mathscr{C} t_2$ implies $t_1 = t_2$. Let $[\![\cdot]\!] \stackrel{\text{def}}{=} \mathscr{C}$ and observe that $[\![\cdot]\!]$ satisfies the relations in Definition 4.

Theorem 1. $\perp_{\mathsf{t}} \circ [\![\cdot]\!] = [\![\cdot]\!] \circ \perp_{\mathsf{st}}$.

Proof. We show that $[\![T]\!]^{\perp} = [\![T^{\perp}]\!]$ by case analysis on the shape of T. We consider two representative cases, the others being similar or trivial.

$\boxed{T = ?T_1 \,.\, T_2}$ We derive:

$$\begin{array}{ll}
[\![T]\!]^{\perp} = [\![?T_1 \,.\, T_2]\!]^{\perp} & \text{by definition of } T \\
= {}^{\bullet\circ}[[\![T_1]\!] * [\![T_2]\!]]^{\perp} & \text{by definition of } [\![\cdot]\!] \\
= {}^{\circ\bullet}[[\![T_1]\!] * [\![T_2]\!]] & \text{by definition of duality on channel types} \\
= {}^{\circ\bullet}[[\![T_1]\!] * [\![T_2^{\perp\perp}]\!]] & \text{because duality is an involution} \\
= [\![!T_1 \,.\, T_2^{\perp}]\!] & \text{by definition of } [\![\cdot]\!] \\
= [\![T^{\perp}]\!] & \text{by definition of duality on session types}
\end{array}$$

$\boxed{T = T_1 \oplus T_2}$ We derive:

$$\begin{array}{ll}
[\![T]\!]^{\perp} = [\![T_1 \oplus T_2]\!]^{\perp} & \text{by definition of } T \\
= {}^{\circ\bullet}[[\![T_1^{\perp}]\!] + [\![T_2^{\perp}]\!]]^{\perp} & \text{by definition of } [\![\cdot]\!] \\
= {}^{\bullet\circ}[[\![T_1^{\perp}]\!] + [\![T_2^{\perp}]\!]] & \text{by definition of duality on channel types} \\
= [\![T_1^{\perp} \,\&\, T_2^{\perp}]\!] & \text{by definition of } [\![\cdot]\!] \\
= [\![T^{\perp}]\!] & \text{by definition of duality on session types}
\end{array}$$

□

Theorem 4. $\langle\!\langle \cdot \rangle\!\rangle = [\![\cdot]\!]^{-1}$.

Proof. We show that $\langle\!\langle \cdot \rangle\!\rangle \circ [\![\cdot]\!] = \mathrm{id}_{\mathbb{S}}$ where $\mathrm{id}_{\mathbb{S}}$ is the identity on $\mathbb{S}$. It suffices to show that $\mathscr{R} \stackrel{\text{def}}{=} \{(\langle\!\langle [\![T]\!] \rangle\!\rangle, T) \mid T \in \mathbb{S}\}$ coincides with $\mathrm{id}_{\mathbb{S}}$. We prove the two inclusions $\mathscr{R} \subseteq \mathrm{id}_{\mathbb{S}}$ and $\mathrm{id}_{\mathbb{S}} \subseteq \mathscr{R}$ in this order.

Concerning the relation $\mathscr{R} \subseteq \mathrm{id}_{\mathbb{S}}$, take $S \mathscr{R} T$. Then $S = \langle\!\langle [\![T]\!] \rangle\!\rangle$. We proceed reasoning by cases on the shape of T. We consider only the case $T = !T_1 \,.\, T_2$, the others being

analogous. We have to show that there exist S_1 and S_2 such that $S = \,!S_1\,.\,S_2$ and $S_i \mathrel{\mathscr{R}} T_i$ for every $i = 1,2$. We derive:

$$\begin{aligned}
S = \langle\!\langle [\![T]\!] \rangle\!\rangle &= \langle\!\langle [\![\,!T_1\,.\,T_2]\!] \rangle\!\rangle && \text{by definition of } T\\
&= \langle\!\langle {}^{\circ\bullet}[[\![T_1]\!] * [\![T_2^\perp]\!]] \rangle\!\rangle && \text{by definition of } [\![\cdot]\!]\\
&= \,!\langle\!\langle [\![T_1]\!] \rangle\!\rangle\,.\,\langle\!\langle [\![T_2^\perp]\!]^\perp \rangle\!\rangle && \text{by definition of } \langle\!\langle\cdot\rangle\!\rangle\\
&= \,!\langle\!\langle [\![T_1]\!] \rangle\!\rangle\,.\,\langle\!\langle [\![T_2^{\perp\perp}]\!] \rangle\!\rangle && \text{by Theorem 1}\\
&= \,!\langle\!\langle [\![T_1]\!] \rangle\!\rangle\,.\,\langle\!\langle [\![T_2]\!] \rangle\!\rangle && \text{because duality is an involution}
\end{aligned}$$

and we conclude by taking $S_i \stackrel{\text{def}}{=} \langle\!\langle [\![T_i]\!] \rangle\!\rangle$ and observing that $S_i \mathrel{\mathscr{R}} T_i$ by definition of $\mathscr{R}$ for every $i = 1,2$.

Concerning the relation $\mathrm{id}_{\mathbb{S}} \subseteq \mathscr{R}$, take $(S,T) \in \mathrm{id}_{\mathbb{S}}$, meaning $S = T$. We have to show that $S \mathrel{\mathscr{R}} T$. By definition of $\mathscr{R}$ we have $\langle\!\langle [\![T]\!] \rangle\!\rangle \mathrel{\mathscr{R}} T$ and from the relation $\mathscr{R} \subseteq \mathrm{id}_{\mathbb{S}}$ we deduce $\langle\!\langle [\![T]\!] \rangle\!\rangle = T = S$. We conclude by definition of $\mathscr{R}$. □

B Supplement to Section 4

B.1 Polymorphic References

Here we show why the standard example that motivates the value restriction in the typing rule for `let` [32] is not a counterexample for the soundness of our type system, despite our rule for `let` generalizes type variables even when the bound expression is not a value. To begin with, we define an `OCaml` module `Ref` that models mutable references using sessions:

```
module Ref =
  struct
    let ref v0 =
      let rec aux v x =
        match branch x with
          `Delete x → close x
        | `Set x → let v, x = receive x in aux v x
        | `Get x → let x = send x v in aux v x
      in
      let a, b = create () in
      let _ = Thread.create (aux v0) a in b
    let delete r = close (select r (fun x → `Delete x))
    let set r = send (select r (fun x → `Set x))
    let get r = receive (select r (fun x → `Get x))
  end
```

The module exports four functions `ref`, `delete`, `set`, and `get`. The first two functions respectively create and destroy a mutable reference, while `set` and `get` respectively set and retrieve the content of a reference. According to `ref`, a mutable reference is a thread whose body is represented as a recursive function `aux` parameterized on the current value `v0` stored in the reference and an endpoint `x` on which the reference listens for three kinds of operations, identified by the tags `Delete`, `Set`, and `Get`. The functions `delete`, `set`, and `get` simply select the corresponding operation and possibly perform the required I/O. `OCaml` infers the following signature for `Ref`:

```
module Ref :
  sig
    val ref :
      α → (○,●,[< 'Delete of (○,○,unit) t
                | 'Get of (○,●,α * (○,●,β) t) t
                | 'Set of (●,○,α * (●,○, b) t) t ] as β) t
    val delete : (○,●,[> 'Delete of (○,○,unit) t ]) t → unit
    val set : (○,●,[> 'Set of (●,○,α * (β,γ,δ) t) t ]) t → α → (γ,β,δ) t
    val get : (○,●,[> 'Get of (○,●,α * (β,γ,δ) t) t ]) t → α * (β,γ,δ) t
  end
```

Now we try to use this implementation of mutable references to reproduce the counterexample in [32]. The first attempt is based on the program below, which respects the threading in the use of `r` as required by our EL predicate:

```
1  let r = Ref.ref (fun x → x) in        (* r : ∀α.(α→α) ref  *)
2  let r = Ref.set r (fun x → x + 1) in  (* r : (int→int) ref  *)
3  let f, r = Ref.get r in (* f : int→int, r : (int→int) ref  *)
4  f true                                (* type error         *)
```

On each line we show the type of each binding. Note that the type of `r` is generalized on line 1 according to our rule [T-LET], but it is subsequently instantiated on line 2. Therefore, by the time an attempt is made to apply the function stored in `r` to `true`, the type of `r` correctly records the fact that the type of the function is `int → int` (line 3) and its application to `true` is flagged as ill typed (line 4).

The second attempt is based on a program that disrespects the threading of the use of `r`, so that its polymorphic type can be instantiated multiple times:

```
1  let r = Ref.ref (fun x → x) in        (* r : ∀α.(α→α) ref  *)
2  let _ = Ref.set r (fun x → x + 1) in
3  let f, _ = Ref.get r in               (* runtime error ↯    *)
4  f true
```

This program is well typed thanks to our rule [T-LET], which generalizes the type variables in the type of `r` (line 1). However, by the time the reference `r` is accessed for the `get` operation (line 3), it has been invalidated by the previous `set` operation (line 2), therefore the program results in a runtime error. In general, all the programs in which rule [T-LET] yields dangerous generalizations either are ill typed or they violate the EA predicate. Therefore, the premises of Theorem 2 suffice to establish subject reduction.

B.2 Subject Reduction for Expressions

We just recall the key type preservation result from [32], whose proof only requires minor adaptations concerning the set of values in FuSe.

Lemma 1 (subject reduction for expressions). *If* $\Gamma \vdash e : t$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : t$.

Proof. This is a straightforward adaptation of [32, Lemma 4.3], where the values include a few more cases. □

B.3 Subject reduction for processes

The next proposition establishes a few properties of the reduction relation on type environments.

Proposition 2. *If* $\Gamma \xrightarrow{\ell} \Gamma'$*, then the following properties hold:*

1. $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)$*;*
2. *if* Γ *is balanced, then so is* Γ'*;*
3. $\Gamma'(u) = \Gamma(u)$ *for every* $u \in \mathsf{dom}(\Gamma) \setminus \mathsf{fn}(\ell)$*.*

Proof. Straightforward from the definition of type environment reduction. □

A reduction may invalidate endpoints occurring in an expression or in a process, even if there is no redex in such terms. The following result shows that typing is preserved when invalidations occur. This is a consequence of the fact that the type $\forall A.\, A$ of invalid endpoints allows them to be typed with any instance of a channel type.

Lemma 2 (invalidation). *Let* $\Gamma \xrightarrow{\ell} \Gamma'$ *where* Γ *is balanced. Then* $\Gamma \vdash e : t$ *implies* $\Gamma' \vdash e_\ell : t$*, and* $\Gamma \vdash P$ *implies* $\Gamma' \vdash P_\ell$*.*

Proof. A simple induction on the typing derivation. □

The following two lemmas allow us to reason on the typing of terms occurring in the hole of an evaluation context. In the statements of these results, by "sub-derivation of $\mathscr{D}$" we mean a sub-tree of $\mathscr{D}$. Note that the replacement lemma differs from the one in [32] since the expression e' is replaced not in the original evaluation context $\mathscr{E}$, but in the context $\mathscr{E}_\ell$ where some endpoints may have been invalidated.

Lemma 3 (typability of subterms). *If* $\mathscr{D}$ *be a derivation for* $\Gamma \vdash \mathscr{E}[e] : t$*, then there exists a sub-derivation* $\mathscr{D}'$ *of* $\mathscr{D}$ *that concludes* $\Gamma \vdash e : s$*.*

Proof. By induction on $\mathscr{E}$, observing that the $[\,]$ of an evaluation context is never found within the scope of a binder. □

Lemma 4 (replacement). *If*

1. $\Gamma \xrightarrow{\ell} \Gamma'$ *where* Γ *is balanced,*
2. $\mathscr{D}$ *is a derivation concluding* $\Gamma \vdash \mathscr{E}[e] : t$*,*
3. $\mathscr{D}'$ *is a sub-derivation of* $\mathscr{D}$ *concluding* $\Gamma \vdash e : s$*,*
4. *the position of* $\mathscr{D}'$ *in* $\mathscr{D}$ *corresponds to that of* $[\,]$ *in* $\mathscr{E}$*, and*
5. $\Gamma' \vdash e' : s$*,*

then $\Gamma' \vdash \mathscr{E}_\ell[e'] : t$*.*

Proof. By induction on $\mathscr{E}$. □

The type system is not substructural, so it enjoys a standard form of weakening.

Lemma 5 (weakening). *The following properties hold:*

1. *If* $\Gamma \vdash e : t$, *then* $\Gamma, \Gamma' \vdash e : t$;
2. *If* $\Gamma \vdash P$, *then* $\Gamma, \Gamma' \vdash P$.

Proof. Standard properties of any non-substructural type system. □

Structural congruence alters the basic arrangement of processes without affecting typing.

Lemma 6 (congruence preserves typing). *If* $\Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Gamma \vdash Q$.

Below is the statement of subject reduction of processes (Section 4) with its full proof. The result is essentially standard, except for the fact that endpoints may be invalidated in the reduct. The hypothesis $\mathsf{EA}(P)$ suffices to exclude the possibility that `error` occurs in the reduct.

Theorem 2 (subject reduction for processes)**.** If $\Gamma \vdash P$ and Γ is balanced and $\mathsf{EA}(P)$ and $P \xrightarrow{\ell} Q$, then there exist Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$ and $\Gamma' \vdash Q$.

Proof. By induction on the derivation of $P \xrightarrow{\ell} Q$ and by cases on the last rule applied.

[R-FORK] Then $P = \langle \mathscr{E}[\texttt{fork}\ v\ w] \rangle$ and $Q = \langle \mathscr{E}[()] \rangle \mid \langle vw \rangle$ and $\ell = \tau$. From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{fork}\ v\ w] : \texttt{unit}$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{fork})$ we deduce that $\Gamma \vdash \texttt{fork}\ v\ w : \texttt{unit}$ and $\Gamma \vdash v : t \to \texttt{unit}$ and $\Gamma \vdash w : t$. From Lemma 4 we deduce $\Gamma \vdash \mathscr{E}[()] : \texttt{unit}$. From [T-APP] we deduce $\Gamma \vdash vw : \texttt{unit}$. We conclude with two applications of [T-THREAD] and one application of [T-PAR] by taking $\Gamma' = \Gamma$.

[R-CREATE] Then $P = \langle \mathscr{E}[\texttt{create}()] \rangle$ and $Q = (\nu a)\langle \mathscr{E}[(a^+, a^-)] \rangle$ where a is fresh and $\ell = \tau$. From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{create}()] : \texttt{unit}$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{create})$ we deduce $\Gamma \vdash \texttt{create}() : t * t^\perp$. Since a is fresh we have $a^+, a^-, a^* \notin \mathsf{dom}(\Gamma)$. From Lemma 4 we deduce $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A.A \vdash \mathscr{E}[(a^+, a^-)] : \texttt{unit}$. We conclude with one application of [T-THREAD] and one application of [T-NEW] by taking $\Gamma' = \Gamma$.

[R-CLOSE] Then $P = \langle \mathscr{E}[\texttt{close}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{close}\ a^{\overline{p}}] \rangle$ and $Q = \langle \mathscr{E}_\ell[()] \rangle \mid \langle \mathscr{E}'_\ell[()] \rangle$ and $\ell = \mathsf{c}a$. From [T-PAR] and [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{close}\ a^p] : \texttt{unit}$ and $\Gamma \vdash \mathscr{E}'[\texttt{close}\ a^{\overline{p}}] : \texttt{unit}$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{close})$ we deduce $\Gamma(a^p) = \Gamma(a^{\overline{p}}) = {}^{\circ\circ}[\texttt{unit}]$ and $\Gamma \vdash \texttt{close}\ a^p : \texttt{unit}$ and $\Gamma \vdash \texttt{close}\ a^{\overline{p}} : \texttt{unit}$. Therefore, $\Gamma = \Gamma', a^p : {}^{\circ\circ}[\texttt{unit}], a^{\overline{p}} : {}^{\circ\circ}[\texttt{unit}]$ for some Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$. From Lemma 4 we deduce $\Gamma' \vdash \mathscr{E}_\ell[()] : \texttt{unit}$ and $\Gamma' \vdash \mathscr{E}'_\ell[()] : \texttt{unit}$. We conclude with two applications of [T-THREAD] and one application of [T-PAR].

[R-COMM] Then $P = \langle \mathscr{E}[\texttt{send}\ a^p\ v] \rangle \mid \langle \mathscr{E}'[\texttt{receive}\ a^{\overline{p}}] \rangle$ and $Q = \langle \mathscr{E}_\ell[a^p] \rangle \mid \langle \mathscr{E}'_\ell[(v, a^{\overline{p}})] \rangle$ and $\ell = \mathtt{m}ap$. From the hypothesis $\Gamma \vdash P$ and rules [T-PAR] and [T-THREAD] we deduce that $\Gamma \vdash \mathscr{E}[\texttt{send}\ a^p\ v] : \texttt{unit}$ and $\Gamma \vdash \mathscr{E}'[\texttt{receive}\ a^{\overline{p}}] : \texttt{unit}$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{send})$ and the hypothesis that Γ is balanced we deduce that there exists Γ'' such that $\Gamma = \Gamma'', a^p : {}^{\circ\bullet}[t * s], a^{\overline{p}} : {}^{\bullet\circ}[t * s]$ and $\Gamma \vdash \texttt{send}\ a^p\ v : s^\perp$ and $\Gamma \vdash v : t$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{receive})$ we deduce that and $\Gamma \vdash \texttt{receive}\ a^{\overline{p}} : t * s$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma'', a^p : s^\perp, a^{\overline{p}} : s$ and observe that $\Gamma \xrightarrow{\ell} \Gamma'$. From Lemma 4 we derive $\Gamma' \vdash \mathscr{E}_\ell[a^p] : \texttt{unit}$. From one application of [T-PAIR] and Lemma 4 we derive $\Gamma' \vdash \mathscr{E}'_\ell[(v, a^{\overline{p}})] : \texttt{unit}$. We conclude with two applications of [T-THREAD] and one application of [T-PAR].

[R-LEFT] Then $P = \langle \mathscr{E}[\texttt{left}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{branch}\ a^{\overline{p}}] \rangle$ and $Q = \langle \mathscr{E}_\ell[a^p] \rangle \mid \langle \mathscr{E}'_\ell[\mathtt{L}\ a^{\overline{p}}] \rangle$ and $\ell = \mathtt{m}ap$. From the hypothesis $\Gamma \vdash P$ and rules [T-PAR] and [T-THREAD] we deduce that $\Gamma \vdash \mathscr{E}[\texttt{left}\ a^p] : \texttt{unit}$ and $\Gamma \vdash \mathscr{E}'[\texttt{branch}\ a^{\overline{p}}] : \texttt{unit}$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{left})$ and the hypothesis that Γ is balanced we deduce that there exists Γ'' such that $\Gamma = \Gamma'', a^p : {}^{\circ\bullet}[t + s], a^{\overline{p}} : {}^{\bullet\circ}[t + s]$ and $\Gamma \vdash \texttt{left}\ a^p : t^\perp$. From Lemma 3 and $\mathsf{TypeOf}(\texttt{branch})$ we deduce that $\Gamma \vdash \texttt{branch}\ a^{\overline{p}} : t + s$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma'', a^p : t^\perp, a^{\overline{p}} : t$ and observe that $\Gamma \xrightarrow{\ell} \Gamma'$. From Lemma 4 we derive $\Gamma' \vdash \mathscr{E}_\ell[a^p] : \texttt{unit}$. From one application of [T-LEFT] and Lemma 4 we derive $\Gamma' \vdash \mathscr{E}'_\ell[\mathtt{L}\ a^{\overline{p}}] : \texttt{unit}$. We conclude with two applications of [T-THREAD] and one application of [T-PAR].

[R-RIGHT] Symmetric of the previous case.

[R-ERROR] Then $P = \langle \mathscr{E}[\mathtt{c}\ a^*] \rangle$ and $Q = \texttt{error}$ and $\ell = \tau$. This case is impossible for it contradicts the hypothesis $\mathsf{EA}(P)$.

[R-PAR] Then $P = P' \mid R$ and $P' \xrightarrow{\ell} Q'$ and $Q = Q' \mid R_\ell$. From [T-PAR] we deduce $\Gamma \vdash P'$ and $\Gamma \vdash R$. By induction hypothesis we deduce $\Gamma' \vdash Q'$ for some Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$. From Lemma 2 we deduce that $\Gamma' \vdash R_\ell$. We conclude with an application of [T-PAR].

[R-NEW-1] Then $P = (\nu a)P'$ and $P' \xrightarrow{\ell'} Q'$ and $Q = (\nu a)Q'$ and $\ell = \tau$ and ℓ' is either $\mathtt{m}ap$ or $\mathtt{c}a$. From [T-NEW] we deduce $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \vdash P'$. Observe that $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A$ is balanced if so is Γ. We distinguish two subcases, depending on the shape of ℓ'.

- If $\ell' = \mathtt{c}a$, then by induction hypothesis we deduce $\Gamma, a^* : \forall A\,.\,A \vdash Q'$. By Lemma 5, we deduce $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \vdash Q'$ and we conclude by taking $\Gamma' = \Gamma$ and one application of [T-NEW].
- If $\ell' = \mathtt{m}ap$, then by induction hypothesis we deduce that there exists s such that $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \xrightarrow{\mathtt{m}ap} \Gamma, a^+ : s, a^- : s^\perp, a^* : \forall A\,.\,A$ and $\Gamma, a^+ : s, a^- : s^\perp, a^* : \forall A\,.\,A \vdash Q'$. We conclude by taking $\Gamma' = \Gamma$ and one application of [T-NEW].

[R-NEW-2] Then $P = (\nu a)P'$ and $P' \xrightarrow{\ell} Q'$ and $Q = (\nu a)Q'$ and $a^p \notin \mathsf{fn}(\ell)$. From [T-NEW] we deduce $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \vdash P'$. Observe that $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A$ is balanced if so is Γ. By induction hypothesis we deduce $\Gamma', a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \vdash Q'$ for some Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$. We conclude with an application of [T-NEW].

[R-STRUCT] A simple induction using Lemma 6. □

B.4 Soundness

This section contains the proof of partial progress (Theorem 3). The first auxiliary result provides a syntactic characterization of those expressions that are unable to reduce further. These are not necessarily values, for expressions may contain instances of the communication primitives that reduce only at the level of processes.

Lemma 7. *Let Γ ground and $\Gamma \vdash e : t$ and $e \not\longrightarrow$. Then either e is a value or there exist $\mathscr{E}$, $\mathtt{c}$, and v such that $e = \mathscr{E}[\mathtt{c}\ v]$ where $\mathtt{c} \notin \{(), \texttt{fix}\}$.*

Proof. If e is a value there is nothing left to prove, so we assume that e is not a value and proceed by induction on the structure of e and by cases on its shape, excluding the case when e is a variable. We only discuss the case when $e = e_1 e_2$, the others being simpler or similar.

- If e_1 is not a value, then from the hypothesis $e \not\longrightarrow$ we deduce $e_1 \not\longrightarrow$. From the hypothesis $\Gamma \vdash e : t$ we deduce $\Gamma \vdash e_1 : s \to t$. By induction hypothesis we deduce that there exist $\mathscr{E}'$, $\mathtt{c}$, and v such that $e_1 = \mathscr{E}'[\mathtt{c}\ v]$ where $\mathtt{c} \notin \{(), \mathtt{fix}\}$. We conclude by taking $\mathscr{E} = \mathscr{E}' e_2$.
- If e_1 is a value but e_2 is not, then from the hypothesis $e \not\longrightarrow$ we deduce $e_2 \not\longrightarrow$. From the hypothesis $\Gamma \vdash e : t$ we deduce $\Gamma \vdash e_2 : s$ for some s. By induction hypothesis we deduce that there exist $\mathscr{E}'$, $\mathtt{c}$, and v such that $e_2 = \mathscr{E}'[\mathtt{c}\ v]$ where $\mathtt{c} \notin \{(), \mathtt{fix}\}$. We conclude by taking $\mathscr{E} = e_1 \mathscr{E}'$.
- If both e_1 and e_2 are values, then from the hypothesis $e \not\longrightarrow$ we can exclude the possibility that e_1 is an abstraction or $\mathtt{fix}$, for in these cases e always reduces. From the hypothesis $\Gamma \vdash e : t$ we deduce that $\Gamma \vdash e_1 : s \to t$. By inspecting the syntax of values and looking at Table 5, there are two possibile values other than abstractions and $\mathtt{fix}$ that can have arrow type. Either e_1 is a constant $\mathtt{c} \notin \{(), \mathtt{fix}\}$, in which case we conclude by taking $\mathscr{E} = [\,]$ and $v = e_2$, or e_1 has the form $\mathtt{c}\ v$ where $\mathtt{c} \in \{\mathtt{fork}, \mathtt{send}\}$, in which case we conclude by taking $\mathscr{E} = [\,] e_2$. □

Theorem 3 (partial progress)**.** If $\emptyset \vdash P$ and $\mathsf{EL}(P)$, then either there exists Q such that $P \xrightarrow{\tau} Q$ or $P \equiv \langle () \rangle$ or P is deadlocked.

Proof. Observe that $P \xrightarrow{\ell}$ implies $\ell = \tau$, because P is typed in an empty environment and so is a closed process. Suppose that $P \xrightarrow{\tau} \not\longrightarrow$ and $P \not\equiv \langle () \rangle$, for otherwise there is nothing left to prove. Using structural congruence, we can always derive

$$P \equiv (\nu a_1) \cdots (\nu a_n) \textstyle\prod_{i \in I} \langle e_i \rangle$$

where $I \neq \emptyset$.

From the hypothesis $P \xrightarrow{\tau} \not\longrightarrow$ we deduce $e_i \not\longrightarrow$ for every $i \in I$ and from the hypothesis $\emptyset \vdash P$ we know that each e_i is well typed and has type $\mathtt{unit}$. Hence, from Lemma 7 we deduce that for every $i \in I$ either e_i is a value or there exist $\mathscr{E}_i$, $\mathtt{c}_i$, and v_i such that $e_i = \mathscr{E}_i[\mathtt{c}_i\ v_i]$ where $\mathtt{c}_i \notin \{(), \mathtt{fix}\}$. Since the only value of type $\mathtt{unit}$ is $()$, we can assume that none of the e_i is $()$, for such threads could be removed by structural congruence. From the hypothesis $P \xrightarrow{\tau} \not\longrightarrow$ we also deduce that none of the $\mathtt{c}_i$ is $\mathtt{create}$ or $\mathtt{fork}$, for otherwise P would be able to reduce according to the rules in Table 2. In summary, we can derive

$$P \equiv (\nu a_1) \cdots (\nu a_n) \textstyle\prod_{i \in I} \langle \mathscr{E}_i[\mathtt{c}_i\ v_i] \rangle$$

where $\mathtt{c}_i \notin \{(), \mathtt{fix}, \mathtt{fork}, \mathtt{create}\}$ for every $i \in I$.

From the hypothesis $\emptyset \vdash P$ we deduce that for all $i \in I$ there exist c_i and p_i such that $v_i = c_i^{p_i}$. Therefore we derive $P \equiv (\nu a_1) \cdots (\nu a_n) Q$ where $Q = \prod_{i \in I} \langle \mathscr{E}_i[\mathtt{c}_i\ c_i^{p_i}] \rangle$. We observe that if $\mathtt{c}_i = \mathtt{send}$, then $\mathscr{E}_i \neq [\,]$ because the partial application $\mathtt{send}\ v_i$ cannot have type $\mathtt{unit}$. Therefore, all the applications of $\mathtt{send}$ are saturated and ready to

synchronize with the corresponding `receive`, if this occurred in an evaluation context. From the hypothesis $\mathsf{EL}(P)$, we also know that $p_i \in \{+,-\}$ for every $i \in I$.

Now, from the hypotheses $\emptyset \vdash P$ and [T-NEW], we know that there exists Γ balanced such that $\Gamma \vdash Q$. Consider $i \in I$. From the hypothesis $\mathsf{EL}(P)$ we deduce that $c_i^{\overline{p}_i}$ must occur somewhere in P. We reason by cases on $\mathtt{c}_i$, and discuss only one case, when $\mathtt{c}_i = \mathtt{send}$, the others being analogous. Then, there exist t and s such that $\Gamma \vdash c_i^{p_i} : {}^{\circ\bullet}[t * s]$ and $\Gamma \vdash c_i^{\overline{p}_i} : {}^{\bullet\circ}[t * s]$. Suppose that $c_i^{\overline{p}_i} = c_j^{p_j}$ for some $j \in I$. It cannot be the case that $\mathtt{c}_j = \mathtt{receive}$, for otherwise P would be able to reduce, therefore $\mathtt{c}_j \in \{\mathtt{send}, \mathtt{left}, \mathtt{right}, \mathtt{branch}\}$. But this is impossible too, either because the capability annotation in the type of the endpoint is incompatible with $\bullet\circ$ (when $\mathtt{c}_j \in \{\mathtt{send}, \mathtt{left}, \mathtt{right}\}$, or because the message type has a topmost $+$ type constructor, while it should have a topmost $*$ type constructor (when $\mathtt{c}_j = \mathtt{branch}$). In conclusion, we deduce that $c_i^{\overline{p}_i}$ cannot be any of the $c_j^{p_j}$ for $j \in I$. Therefore, it must be the case that $c_i^{\overline{p}_i} \in \mathsf{fn}(\mathscr{E}_j)$ for some $j \in I$. □