A Simple Library Implementation of Binary Sessions

Under consideration for publication in J. Functional Programming

A Simple Library Implementation of Binary Sessions

LUCA PADOVANI∗
Dipartimento di Informatica, Università di Torino, ITALY
(*e-mail:* luca.padovani@di.unito.it)

Abstract

Inspired by the continuation-passing encoding of binary sessions, we describe a simple approach to embed a hybrid form of session type checking into any programming language that supports parametric polymorphism. The approach combines static protocol analysis with dynamic linearity checks. To demonstrate the effectiveness of the technique, we implement a well-integrated `OCaml` module for session communications. For free, `OCaml` provides us with equi-recursive session types, parametric behavioural polymorphism, complete session type inference, and session subtyping.

1 Introduction

Session type systems (Honda, 1993; Honda et al., 1998) are an established means for enforcing structured communication protocols through static analysis. Gay and Vasconcelos (2010) have presented a session type system that integrates smoothly with an `ML`-like functional language. Elaborating on previous works, Wadler (2014) has investigated the theoretical relevance of this type system showing its connections with axioms and rules of linear logic. From a practical viewpoint, however, the type system put forward by Gay and Vasconcelos (2010) appears challenging to adopt in a mainstream programming language, for it requires peculiar features for (F.1) describing *structured protocols* as sequences of I/O operations and branching points, (F.2) checking that the peer endpoints of a session are used according to *dual* protocols, and (F.3) ensuring the *linear usage* of session endpoints.

Successful attempts to *encode* these features in Haskell have been reported by Neubauer and Thiemann (2004), Sackman and Eisenbach (2008), Pucella and Tov (2008), and Imai et al. (2010). These works accomplish (F.1) and (F.2) using Haskell's type system and (F.3) by encapsulating endpoints in a suitable monad: not being able to *name* endpoints prevents the programmer from using them non-linearly. As discussed by Imai et al. (2010), however, these encodings have a price in terms of expressiveness, usability, or portability: some of them omit or constrain key features of sessions (*e.g.* session interleaving or delegation), some require programmers to write considerable amounts of boilerplate code (*e.g.* to access nameless channels or to unwind recursions), and most make use of features that are unique to Haskell's type system (*e.g.* functional dependencies or type-level computations) thus hampering their portability to other languages.

∗ Supported by ICT COST Action IC1201 BETTY and MIUR Project CINA.

In this paper we describe another approach to equip a programming language with session type checking, diverging from the aforementioned ones in two ways. First, we work with a form of session types inspired by the continuation-passing encoding of binary sessions (Dardha et al., 2012) that makes it possible to express session type duality solely in terms of type equality. Second, we do not try to enforce the linear usage of session endpoints statically, but we rely on a lightweight mechanism that detects potentially unsafe linearity violations at runtime. Similar mechanisms have been described by Tov and Pucella (2010) and Hu and Yoshida (2016). Overall, we find that our approach compares favourably with the previous ones in terms of features, simplicity of use and implementation, and portability: it provides direct access to sessions through first-class session endpoints, not imposing contraints on session interleaving or delegation; it supports recursive session types and session subtyping by piggybacking on the host type system; it is portable to any programming language whose type system features parametric polymorphism.

To introduce the key idea behind our approach, we start by recalling the type of the communication primitives `send` and `receive` as given by Gay and Vasconcelos (2010):

$$\texttt{send} : t \to \,!t\,.\,T \to T \qquad \texttt{receive} : ?t\,.\,S \to t * S \tag{1}$$

The application `send` $v\ a$ sends a message v of type t on a session endpoint a of type $!t\,.\,T$ and reduces to the same endpoint a with its type changed to T. The session type T describes the protocol according to which a must be used by the sender after this communication. The application `receive` b waits for a message from the endpoint b and reduces to a pair (w,b) containing the message w and the endpoint b itself. The endpoint has type $?t\,.\,S$ before the reduction and type S after the reduction. When a and b are the peer endpoints of the same session, the session types $!t\,.\,T$ and $?t\,.\,S$ (hence T and S) must be *dual* of each other: every input in T must be matched by a corresponding output in S, and vice versa. This ensures that sender and receiver interact correctly also in the rest of the conversation.

The given semantics and typing of `send` and `receive` are not unique. An alternative semantics for the same primitives emerges from the studies of Kobayashi (2002), Demangeon and Honda (2011), and Dardha et al. (2012). These works show that a sequence of communications in a session can be encoded as a sequence of one-shot communications in a *chain of distinct channels*, each channel being used for one communication only. According to this intuition, the effect of `send` $v\ a$ would be to create a fresh channel c (the *continuation* of a), to send the pair (v,c) on a, and to return another reference to c. Correspondingly, the effect of `receive` a would be to return the pair received from a. This semantics induces the following alternative typing of the same communication primitives

$$\texttt{send} : t \to\, !\,[t * \kappa[s]] \to \overline{\kappa}[s] \qquad \texttt{receive} : ?\,[t * \kappa[s]] \to t * \kappa[s] \tag{2}$$

where κ stands for an I/O capability, such as $?$ or $!$, and types like $\kappa[s]$ describe channels to be used according to capability κ for exchanging one message of type s. Note that the continuation c created by `send` is given two different types: the reference paired with the payload v and sent on a has type $\kappa[s]$, whereas the reference returned by `send` has type $\overline{\kappa}[s]$, where $\overline{\kappa}$ denotes the dual capability of κ. Overall, $\overline{\kappa}[s]$ and $\kappa[s]$ in (2) play the same roles as T and S in (1), but the types of the two versions of `send` differ for one detail: T in $!t\,.\,T$ describes the behaviour of the *sender* on the endpoint in the rest of the interaction, whereas $\kappa[s]$ in $!\,[t * \kappa[s]]$ describes the behaviour of the *receiver*

on the continuation channel in the rest of the interaction. In particular, we notice that the structurally complex duality relation between the session types $!t.T$ and $?t.S$ of peer endpoints in (1) boils down to a much simpler duality relation between the types $![t * \kappa[s]]$ and $?[t * \kappa[s]]$ in (2). These types differ only for their topmost capability. In fact, we will see that this simplified duality can be expressed solely in terms of type equality, given a suitable representation of channel types. In prospect of encoding static protocol analysis into an existing type system, this is a major advantage of the types in (2) compared to those in (1), granted that they embed the same amount of information concerning the described communication protocol.

Of course, we are not willing to pay the overhead resulting from the creation and exchange of continuations just for the sake of a more palatable typing of the communication primitives. We observe that it makes sense to consider a third version of the primitives, whereby `send` and `receive` have the semantics given by Gay and Vasconcelos (2010), but they are typed like in (2) *as if* they did create and exchange continuations. This mix-up has a practical justification: given that each channel in the continuation-passing encoding of a session is supposed to be used only once, there is no need for `send` v a to create a fresh continuation c for a. Instead, as observed by Dardha et al. (2012), the channel a can be recycled and sent as its own continuation. We take this optimisation one step further: no continuation needs to be actually sent, for the recycled channel a is already known to both communicating parties. In the end, the occurrences of $\kappa[s]$ in (2) need not correspond to any actual piece of transmitted data. They are the types of *ghost continuations* whose sole – but fundamental – purpose is to relate the future behaviours of sender and receiver.

We give the details of our approach in the rest of the paper, following this structure:

- We formalise a core functional language, called `FuSe`, that combines multithreading, session-based communications primitives *à la* Gay and Vasconcelos (2010), and a runtime mechanism that detects endpoint linearity violations (Section 2).
- We define an ordinary `ML`-style type language for `FuSe` and we adapt and extend the encoding of session types (Dardha et al., 2012) into `FuSe` types. We introduce a representation of channel types that allows us to express session type duality solely in terms of type equality (Section 3).
- We equip `FuSe` with an `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). We cannot infer these properties directly from the soundness of the encoding given by Dardha et al. (2012) since `FuSe` primitives do not exchange continuations.
- We describe an implementation of `FuSe` primitives in `OCaml` (Leroy et al., 2014) that allows us to write and type check programs written in the style of Gay and Vasconcelos (2010) without the need of a built-in session type system. The implementation integrates well with `OCaml`, from which we leverage equi-recursive session types with arbitrary branches, parametric (behavioural) polymorphism, complete session type inference, and session subtyping as defined by Gay and Hole (2005) (Section 5).

Section 6 discusses related work further and Section 7 concludes. Proofs and auxiliary results related to Sections 3 and 4 are in Appendixes A and B. The implementation is

available at `http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html`. An early draft of this paper (Padovani, 2015) was uploaded on HAL in October 2015.

2 Syntax and semantics of FuSe

We use infinite sets of *variables* x, y, ... and *sessions* a, b, ...; 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* and *processes* is given below:

$$\begin{array}{rrcl} \textbf{Expressions} & e & ::= & \mathtt{c} \mid x \mid a^p \mid \lambda x.e \mid e_1\ e_2 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\ \textbf{Processes} & P,Q & ::= & \langle e \rangle \mid P \mid Q \mid \mathtt{new}\ a\ \mathtt{in}\ P \mid \mathtt{error} \end{array}$$

Expressions model *threads*, that is the sequential parts of programs, and are mostly conventional. The symbol $\mathtt{c}$ ranges over the following *constants*: the fixpoint combinator `fix`, the unitary value `()`, the usual data constructors `pair`, `inl`, and `inr`, the corresponding deconstructors `fst`, `snd`, and `cases`, the primitive `fork` that spawns new threads, and the session primitives `create`, `close`, `send`, `receive`, `left`, `right`, and `branch` whose semantics will be detailed shortly. Endpoints are not supposed to occur in source programs, they appear as the result of reductions. Processes 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 $\mathtt{new}\ a\ \mathtt{in}\ P$ modelling a session a with scope P, or a runtime `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 labelled reduction relation for processes. We make use of conventional notions of *values* and of *evaluation contexts*, defined thus:

$$\begin{array}{rrcl} \textbf{Values} & v,w & ::= & \mathtt{c} \mid \mathtt{c}^1\ v \mid \mathtt{c}^2\ v\ w \mid a^p \mid \lambda x.e \\ \textbf{Evaluation Contexts} & \mathscr{E} & ::= & [\,] \mid \mathscr{E}\ e \mid v\ \mathscr{E} \mid \mathtt{let}\ x = \mathscr{E}\ \mathtt{in}\ e \end{array}$$

where $\mathtt{c}^1$ ranges over $\{\mathtt{pair},\mathtt{inl},\mathtt{inr},\mathtt{cases},\mathtt{fork},\mathtt{send}\}$ and $\mathtt{c}^2$ over $\{\mathtt{pair},\mathtt{cases}\}$. In values as in expressions application associates to the left. Therefore, $\mathtt{c}^2\ v\ w$ reads as $((\mathtt{c}^2\ v)\ w)$. Note that $\mathtt{send}\ v\ w$ and $\mathtt{cases}\ v_1\ v_2\ v_2$ are not values for these expressions are meant to reduce. 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 ranges over expressions, processes, and evaluation contexts.

The upper part of Table 1 defines reduction of expressions, which is standard. The lower part of the table defines a labelled 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 $\mathsf{ep}(\tau) \stackrel{\text{def}}{=} \emptyset$ and $\mathsf{ep}(\mathtt{m}ap) = \mathsf{ep}(\mathtt{c}a) \stackrel{\text{def}}{=} \{a^+,a^-\}$. Labels allow us to observe the behaviour

Table 1. Reduction of expressions and processes.

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

[R-BETA]	$(\lambda x.e)\ v \longrightarrow e\{v/x\}$
[R-LET]	$\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e\{v/x\}$
[R-FIX]	$\texttt{fix}\ v \longrightarrow v\ (\lambda x.\texttt{fix}\ v\ x)$
[R-FST]	$\texttt{fst}\ (\texttt{pair}\ v\ w) \longrightarrow v$
[R-SND]	$\texttt{snd}\ (\texttt{pair}\ v\ w) \longrightarrow w$
[R-INL]	$\texttt{cases}\ (\texttt{inl}\ w)\ v_1\ v_2 \longrightarrow v_1\ w$
[R-INR]	$\texttt{cases}\ (\texttt{inr}\ w)\ v_1\ v_2 \longrightarrow v_2\ w$

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

[R-ERROR]	$\langle \mathscr{E}[K\ a^*] \rangle \xrightarrow{\tau} \texttt{error}$	
[R-FORK]	$\langle \mathscr{E}[\texttt{fork}\ v\ w] \rangle \xrightarrow{\tau} \langle \mathscr{E}[()] \rangle \mid \langle v\ w \rangle$	
[R-CREATE]	$\langle \mathscr{E}[\texttt{create}\,()] \rangle \xrightarrow{\tau} \texttt{new}\ a\ \texttt{in}\ \langle \mathscr{E}[\texttt{pair}\ a^+\ a^-] \rangle$	a fresh
[R-CLOSE]	$\langle \mathscr{E}[\texttt{close}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{close}\ a^{\overline{p}}] \rangle \xrightarrow{\texttt{c}a} \langle \mathscr{E}_{\texttt{c}a}[()] \rangle \mid \langle \mathscr{E}'_{\texttt{c}a}[()] \rangle$	
[R-COMM]	$\langle \mathscr{E}[\texttt{send}\ a^p\ v] \rangle \mid \langle \mathscr{E}'[\texttt{receive}\ a^{\overline{p}}] \rangle \xrightarrow{map} \langle \mathscr{E}_{map}[a^p] \rangle \mid \langle \mathscr{E}'_{map}[\texttt{pair}\ v_{map}\ a^{\overline{p}}] \rangle$	
[R-LEFT]	$\langle \mathscr{E}[\texttt{left}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{branch}\ a^{\overline{p}}] \rangle \xrightarrow{map} \langle \mathscr{E}_{map}[a^p] \rangle \mid \langle \mathscr{E}'_{map}[\texttt{inl}\ a^{\overline{p}}] \rangle$	
[R-RIGHT]	$\langle \mathscr{E}[\texttt{right}\ a^p] \rangle \mid \langle \mathscr{E}'[\texttt{branch}\ a^{\overline{p}}] \rangle \xrightarrow{map} \langle \mathscr{E}_{map}[a^p] \rangle \mid \langle \mathscr{E}'_{map}[\texttt{inr}\ a^{\overline{p}}] \rangle$	
[R-THREAD]	$\langle \mathscr{E}[e] \rangle \xrightarrow{\tau} \langle \mathscr{E}[e'] \rangle$	$e \longrightarrow e'$
[R-PAR]	$P \mid R \xrightarrow{\ell} Q \mid R_\ell$	$P \xrightarrow{\ell} Q$
[R-NEW]	$\texttt{new}\ a\ \texttt{in}\ P \xrightarrow{\ell \setminus a} \texttt{new}\ a\ \texttt{in}\ Q$	$P \xrightarrow{\ell} Q$
[R-STRUCT]	$P \xrightarrow{\ell} Q$	$P \equiv P' \xrightarrow{\ell} Q' \equiv Q$

of processes on the channels they use. This information is key to show that well-typed processes respect protocols and also to *invalidate* the endpoints used in a reduction.

Definition 1 (invalidation)**.** We let $X_\ell \stackrel{\text{def}}{=} X\{a^*/a^p\}_{a^p \in \texttt{ep}(\ell)}$.

Intuitively, X_ℓ invalidates some of the endpoints a^p in X by replacing them with a^*, after an ℓ-labelled reduction. When $\ell = \tau$, no endpoint is invalidated. When $\ell = map$ or $\ell = \texttt{c}a$, both a^+ and a^- are invalidated. For example, $(\texttt{send}\ c^p\ a^+)_{maq} = \texttt{send}\ c^p\ a^*$.

We now describe the reduction rules of 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 spawner process. 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. For simplicity, in the formal development this operation is synchronous; 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 Gay and Vasconcelos (2010), the output operation reduces to the endpoint a^p used by the sender, while the input operation reduces to a pair containing the received 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. Rule [R-LEFT] models the

selection of a branch in a structured conversation. It is akin to [R-COMM], except that the endpoint used by the receiver is injected into a disjoint sum with `inl` that represents the choice taken. Rule [R-RIGHT] is symmetric. Intuitively, in [R-LEFT] and [R-RIGHT] the message being communicated is just a constructor `inl` or `inr`. 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] yields a runtime error if there is an attempt to "use" an invalid endpoint. The grammar

$$K \;::=\; \texttt{close} \mid \texttt{send}\, v \mid \texttt{receive} \mid \texttt{left} \mid \texttt{right} \mid \texttt{branch}$$

characterises the terms K that "use" an endpoint when applied to it. Rule [R-THREAD] lifts the reduction of expressions to processes. 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. Rule [R-NEW] closes reductions under restrictions. We let $\ell \setminus a \stackrel{\text{def}}{=} \tau$ if $\ell = \mathtt{m}ap$ or $\ell = \mathtt{c}a$ and $\ell \setminus a \stackrel{\text{def}}{=} \ell$ otherwise. Intuitively, 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 structural congruence, which is basically the same of the π-calculus except that the idle process is written $\langle () \rangle$ and the scope of a session can be shrunk or extended only when no valid or invalid endpoint is captured.

Example 1 (mathematical server)**.** We write examples in a sugared and richer version of `FuSe` that includes numbers, booleans, `if-then-else`, pattern matching, and possibly recursive `let`-bindings. All these features can be easily added or encoded in `FuSe` and do not affect any of the results that follow. The examples compile and run using `OCaml` (Leroy et al., 2014) and our implementation of the `FuSe` primitives (Section 5). Below is a simple server for mathematical operations analogous to that given by Gay and Hole (2005):

```
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 1st operand *)
           let m, x = receive x in   (* receive 2nd operand *)
           let x = send (n + m) x in (* send result        *)
           server x                  (* serve more requests *)
```

The server is modelled as a function operating on an endpoint `x`. The function is recursive, so that the server is able to handle several requests within a single session. Each request from the client is represented by a polymorphic variant tag `` `L `` or `` `R `` (we use polymorphic variants as generalisations of `inl` and `inr`, 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 =
  let rec aux acc n y =              (* add n naturals      *)
    if n = 0 then
      begin close (left y); acc end  (* close session       *)
    else
      let y = right y in             (* select + operation  *)
      let y = send acc y in          (* send 1st operand    *)
```

```
    let y = send n y in            (* send 2nd operand    *)
    let res, y = receive y in      (* receive result      *)
    aux res (n - 1) y              (* possibly add more   *)
in aux 0
```

In this case, the client is computing the sum of the first `n` naturals by repeated interactions with 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`. With these definitions in place, the function

```
let main n =
  let a, b = create () in          (* create session      *)
  let _ = fork server a in         (* spawn server        *)
  client n b                       (* run client          *)
```

sets up a new session to compute the sum of the first `n` naturals. ■

Example 2 (configurable forwarder)**.** The function below implements a *configurable forwarder* that can be used to connect two processes, a producer streaming messages on (the peer endpoint of) `src` and a consumer receiving messages from (the peer endpoint of) `dst`.

```
let forwarder mode src dst =
  if mode then                     (* check modality      *)
    let rec aux src dst =
      let msg, src = receive src in  (* receive from source *)
      let dst = send msg dst in      (* send to destination *)
      aux src dst                    (* forever             *)
    in aux src (left dst)            (* select forwarding   *)
  else
    close (send src (right dst))     (* select delegation   *)
```

The forwarder is configurable in the sense that it has two operating modalities determined by the `mode` argument. When `mode` is `true`, the messages streaming from `src` are forwarded on `dst`. When `mode` is `false`, the function performs a *delegation* whereby `dst` is sent on `src`. This way, the consumer receives messages directly from the producer. Observe that the session endpoint `dst` is meant to be used in the same way regardless of the chosen modality. In other words, the producer does not know if messages are delivered directly to the consumer or if they go through the forwarder. ■

3 Types

In this section we define the types for FuSe, we recall the encoding of Dardha et al. (2012) 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 primitives.

3.1 Types for FuSe

The syntax of *type schemes* and of (finite) *types* is given by the grammar

$$\begin{array}{rrcl} \textbf{Type Schemes} & \sigma & ::= & t \mid \forall\alpha\,.\,\sigma \\ \textbf{Types} & t,s & ::= & \bullet \mid \texttt{unit} \mid t \to s \mid t * s \mid t + s \mid \texttt{<}t\texttt{,}s\texttt{>} \end{array}$$

where α, β, ... range over *type variables*. Type schemes are conventional; we will often abbreviate $\forall\alpha_1.\cdots\forall\alpha_n.t$ with $\forall\alpha_1\cdots\alpha_n.t$. Types include the unitary type `unit`, arrows, products, and disjoint sums. In the examples we occasionally use other base types such as `int` and `bool`. In addition, the type $\langle t, s\rangle$ denotes a channel that can be used for receiving messages of type t and sending messages of type s, while the type $\bullet$ is not inhabited and denotes "no message". In particular, $\langle\bullet, t\rangle$ denotes a channel that can only be used for sending messages of type t and $\langle t * \langle\bullet, s\rangle, \bullet\rangle$ denotes a channel for receiving a pair made of a value of type t and another channel that can be used for sending messages of type s. A channel with type $\langle\bullet,\bullet\rangle$ cannot be used for I/O operations.

Instead of introducing concrete syntax for representing recursive types, we consider as types the possibly infinite, regular trees generated by the type constructors shown above and keep using the metavariables t and s to range over such trees. Recall that each regular tree has finitely many distinct subtrees and admits finite representations as a system of equations or using the familiar μ notation (Courcelle, 1983). We define an infinite type t as the (unique) solution of an equation $t = s$ where s contains (guarded) occurrences of the metavariable t. For example, the equation $t = \texttt{int}\to t$ is satisfied by the type $\texttt{int}\to\texttt{int}\to\cdots$ of the "ogre" function that eats infinitely many `int` arguments. The shape of the equation, with the metavariable t unguarded on the left hand side and guarded by a type constructor on the right hand side, guarantees existence and uniqueness of the regular tree that satisfies it (Courcelle, 1983, Theorem 4.3.1). We use infinite types for describing arbitrarily long communication protocols, like the one implemented by `server` and `client` in Example 1. Note that type equality corresponds to regular tree equality.

Duality relates channels types with complementary capabilities:

Definition 2 (type duality)**.** Let $\perp_{\mathsf{ct}}$ be the least relation such that $\langle t, s\rangle \perp_{\mathsf{ct}} \langle s, t\rangle$. When t is a channel type, we write $t^\perp$ for the type s such that $t \perp_{\mathsf{ct}} s$; otherwise, $t^\perp$ is undefined.

The intuition for type duality is that if a process uses an endpoint according to some type t, then we expect another process to use the peer endpoint according to the dual type $t^\perp$. So for example, $\langle\bullet, \texttt{int}\rangle \perp_{\mathsf{ct}} \langle\texttt{int}, \bullet\rangle$ since the dual behaviour of "send an `int`" is "receive an `int`". On the other hand, we have $\langle\bullet,\bullet\rangle \perp_{\mathsf{ct}} \langle\bullet,\bullet\rangle$, since "do nothing" is dual of itself. Type duality is a partial involution: it is only defined on channel types and, when $t^\perp$ is defined, we have $t^{\perp\perp} = t$.

Our representation of channel types is slightly unusual. Most type systems with channel types, from the seminal paper on channel subtyping (Pierce and Sangiorgi, 1996) to those for the linear π-calculus (Kobayashi et al., 1999) and binary sessions (Dardha et al., 2012), use types of the form $\kappa[t]$ or variants of this where $\kappa \in \{?, !\}$ represents an input/output capability. We have used this notation also in Section 1. In these cases, computing the dual of a channel type essentially means defining a suitable dual operator for capabilities such that, for instance, $\overline{?} = !$. One shortcoming of this representation is that duality is easily defined only when capabilities are known. Dealing with unknown capabilities means introducing (possibly dualised) capability variables and this machinery complicates the type system. With our representation, we can dualise a channel type by just swapping the content of its two type parameters, even when we do not know the kind of I/O operations allowed on the channel. For example, the type $t \stackrel{\text{def}}{=} \langle\alpha, \beta\rangle$ denotes a channel for which nothing is known. Nonetheless, the dual of t can still be obtained by swapping α and β,

that is $t^{\perp} = <\beta, \alpha>$. At the end of Section 3.2 we will see that the chosen representation of channel types is key to reduce session type duality to type equality.

To improve readability, sometimes we use the notations `?[`t`]` and `![`t`]` as syntactic sugar for $<t, \bullet>$ and $<\bullet, t>$, respectively. In the following, we make extensive use of channel types with unknown message types, so we reserve some convenient notation also for them:

Definition 3 (channel type variable)**.** We let A, B, ... range over *channel type variables*, namely types of the form $<\alpha, \beta>$, and we write $\forall A \,.\, \sigma$ instead of $\forall \alpha\beta \,.\, \sigma$ when $A = <\alpha, \beta>$.

The dual $A^{\perp}$ of any channel type variable A is always defined.

3.2 From session types to `FuSe` *types and back*

Even though our type system does not use built-in session types, the typing of `FuSe` communication primitives follows from the continuation-passing encoding of session types into channel types (Dardha et al., 2012). In the remainder of this section we formalise the relationship between session types and `FuSe` types, recalling the encoding and adapting it to our setting. Compared to Dardha et al. (2012), we use channel types with a different representation of I/O capabilities and we consider possibly infinite (session) types.

The syntax of (finite) *session types* is given below:

Session Types $T, S \quad ::= \quad \texttt{end} \mid \texttt{?}t\texttt{.}T \mid \texttt{!}t\texttt{.}T \mid T \,\&\, S \mid T \oplus S$

The session type `end` describes an endpoint on which no further communication is allowed and that can only be closed. The session types `?`t`.`T and `!`t`.`T describe endpoints to be used respectively for one input and one output of a message of type t and according to T afterwards. The session types $T \,\&\, S$ and $T \oplus S$ respectively represent external and internal choices in a protocol. A process using an endpoint of type $T \oplus S$ decides whether to behave according to the "left" protocol T or the "right" protocol S. A process using an endpoint of type $T \,\&\, S$ accepts the decision of the process using the peer endpoint. As we have seen in Table 1, the choice is effectively encoded and transmitted as an appropriate message, hence $\oplus$ corresponds to an output and `&` to an input operation. As in 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 $T =$ `!int.!int.?bool.`T describes an endpoint for sending two messages of type `int`, receiving one message of type `bool`, and then used according to the same protocol, over and over again. The given syntax of session types disallows the description of protocols involving delegations, since message types cannot be themselves session types. This limitation is irrelevant, for we introduce session types for illustrative purposes only and there is no problem describing higher-order channels using `FuSe` types (*cf.* Example 4). All the definitions, discussions, and 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 behaviours. It is defined thus:

Definition 4 (session type duality)**.** *Session type duality* is the largest relation $\perp_{\mathsf{st}}$ between session types such that $T \perp_{\mathsf{st}} S$ implies either:

- $T = S =$ `end`, or

- $T = ?t\,.\,T'$ and $S = !t\,.\,S'$ and $T' \perp_{\mathsf{st}} S'$, or
- $T = !t\,.\,T'$ and $S = ?t\,.\,S'$ and $T' \perp_{\mathsf{st}} S'$, or
- $T = T_1 \,\&\, T_2$ and $S = S_1 \oplus S_2$ and $T_i \perp_{\mathsf{st}} S_i$ for every $i = 1,2$, or
- $T = T_1 \oplus T_2$ and $S = S_1 \,\&\, S_2$ and $T_i \perp_{\mathsf{st}} S_i$ for every $i = 1,2$.

Observe that $\perp_{\mathsf{st}}$ is an endofunction on session types. We write $T^\perp$ for the session type S such that $T \perp_{\mathsf{st}} S$ and say that T and S are *dual* of each other.

Duality relates inputs with outputs carrying the same message type and `end` with itself. For example, $?t\,.\,!s\,.\,\mathtt{end} \perp_{\mathsf{st}} !t\,.\,?s\,.\,\mathtt{end}$ and if T is the session type such that $T = !\mathtt{int}\,.\,!\mathtt{int}\,.\,?\mathtt{bool}\,.\,T$, then $T^\perp$ is the session type S such that $S = ?\mathtt{int}\,.\,?\mathtt{int}\,.\,!\mathtt{bool}\,.\,S$. Clearly session type duality is an involution, namely $T^{\perp\perp} = T$ for every T.

We now show that session types and their encoding embed the same information, just written in different ways. To do so, we define an isomorphism between the set $\mathbb{S}$ of session types and a subset $\mathbb{P}$ of `FuSe` types which we call protocol types. The isomorphism that we define is purely syntactic, but is supported by a semantic correspondence between processes (Dardha et al., 2012).

Definition 5 (protocol types)**.** The set of *protocol types* is the largest subset $\mathbb{P}$ of types such that $t \in \mathbb{P}$ implies either:

- $t = \langle\bullet,\bullet\rangle$, or
- $t = \langle t_1 * t_2, \bullet\rangle$ or $t = \langle\bullet, t_1 * t_2\rangle$ and $t_2 \in \mathbb{P}$, or
- $t = \langle t_1 + t_2, \bullet\rangle$ or $t = \langle\bullet, t_1 + t_2\rangle$ and $t_1, t_2 \in \mathbb{P}$.

The morphism from $\mathbb{S}$ to $\mathbb{P}$ is given by the encoding described by Dardha et al. (2012) and rests on the idea that multiple communications on a session endpoint can be modelled as a sequence of one-shot communications in a chain of linear channels. The chain is realised by sending, at each communication, a fresh continuation channel along with the communication payload. For example, the session type $!t\,.\,T$ describes an endpoint used for sending a message of type t first and according to T afterwards. It is encoded as the channel type $\langle\bullet, t * s\rangle$ where s is the encoding of $T^\perp$. The reason why the type s of the continuation is the encoding of $T^\perp$ and not the encoding of T is because the tail T in $!t\,.\,T$ describes the behaviour of the *sender* of the message of type t after it has sent the message, whereas in the encoding the type of the continuation describes the behaviour of the *receiver* of the message on the continuation. Clearly, the sender will also use the same continuation, but according to the type $s^\perp$. In general we have:

Definition 6 (encoder)**.** The *encoder* is the function $[\![\cdot]\!] : \mathbb{S} \to \mathbb{P}$ that satisfies the equations:

$$[\![\mathtt{end}]\!] = \langle\bullet,\bullet\rangle \qquad \begin{array}{ll} [\![?t\,.\,T]\!] = ?\,[t * [\![T]\!]] & [\![T \,\&\, S]\!] = ?\,[[\![T]\!] + [\![S]\!]] \\ [\![!t\,.\,T]\!] = !\,[t * [\![T^\perp]\!]] & [\![T \oplus S]\!] = !\,[[\![T^\perp]\!] + [\![S^\perp]\!]] \end{array}$$

For example, if we consider again $T = !\mathtt{int}\,.\,!\mathtt{int}\,.\,?\mathtt{bool}\,.\,T$, then we derive:

$$\begin{aligned} [\![T]\!] &= ![\mathtt{int} * [\![?\mathtt{int}\,.\,!\mathtt{bool}\,.\,T^\perp]\!]] \\ &= ![\mathtt{int} * ?[\mathtt{int} * [\![!\mathtt{bool}\,.\,T^\perp]\!]]] \\ &= ![\mathtt{int} * ?[\mathtt{int} * ![\mathtt{bool} * [\![T^{\perp\perp}]\!]]]] \\ &= ![\mathtt{int} * ?[\mathtt{int} * ![\mathtt{bool} * [\![T]\!]]]] \end{aligned}$$

If we consider instead $T^{\perp}$, namely the session type S such that $S = \texttt{?int}.\texttt{?int}.\texttt{!bool}.S$, then $[\![S]\!] = \texttt{?[int * ?[int * ![bool * } [\![T]\!] \texttt{]]]}$.

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 7 (decoder)**.** The *decoder* is the function $\langle\!\langle \cdot \rangle\!\rangle : \mathbb{P} \to \mathbb{S}$ that satisfies the equations:

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

The decoder has practical utility to decipher possibly obscure types that have been automatically inferred. `FuSe` comes along with a companion tool that implements $\langle\!\langle \cdot \rangle\!\rangle$ and that is capable of pretty printing `OCaml` types and whole `OCaml` module signatures using the traditional notation of session types.

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 by Dardha et al. (2012) and formally stated in Lemma 1 below, $[\![\cdot]\!]$ commutes with duality and so does $\langle\!\langle \cdot \rangle\!\rangle$. This is key to show that $[\![\cdot]\!]$ and $\langle\!\langle \cdot \rangle\!\rangle$ are indeed one the inverse of the other.

Lemma 1 (commuting duality)**.** *We have* $\perp_{\mathsf{ct}} \circ [\![\cdot]\!] = [\![\cdot]\!] \circ \perp_{\mathsf{st}}$ *and* $\perp_{\mathsf{st}} \circ \langle\!\langle \cdot \rangle\!\rangle = \langle\!\langle \cdot \rangle\!\rangle \circ \perp_{\mathsf{ct}}$.

The 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).

Theorem 1 (isomorphism)**.** $\langle\!\langle \cdot \rangle\!\rangle = [\![\cdot]\!]^{-1}$.

Finally, the most useful consequence of Lemma 1 in combination with our representation of channel types is to provide an alternative, simple method for checking whether $T \perp_{\mathsf{st}} S$ holds. Suppose for example that $[\![T]\!] = \texttt{<}t_i\texttt{,}t_o\texttt{>}$ and $[\![S]\!] = \texttt{<}s_i\texttt{,}s_o\texttt{>}$. By Lemma 1 we deduce

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

thereby turning the verification of a complex relation $T \perp_{\mathsf{st}} S$, which involves matching inputs with outputs across the whole structure of T and S, in two plain type equalities. This is a strong point in favour of encoded (as opposed to built-in) session types.

4 Type system

In this section we present the type system for `FuSe` and state its properties. The typing rules are essentially standard for `ML`-like languages, in particular they have no baked-in features specifically targeted to session type checking. Compared to the type system of Gay and Vasconcelos (2010), the main differences concern the typing of communication primitives, the fact that the type system is not substructural, and the handling of invalid endpoints.

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 $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$. Table 2 shows the typing rules. 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

Table 2. Typing rules for expressions and processes.

Expressions $\boxed{\Gamma \vdash e : t}$

$$\frac{\mathsf{TypeOf}(\mathtt{c}) \succ t}{\Gamma \vdash \mathtt{c} : t}\ [\text{T-CONST}] \qquad \frac{\sigma \succ t}{\Gamma, u : \sigma \vdash u : t}\ [\text{T-NAME}]$$

$$\frac{\Gamma, x : t \vdash e : s}{\Gamma \vdash \lambda x.e : t \to s}\ [\text{T-FUN}] \qquad \frac{\Gamma \vdash e_1 : t \to s \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : s}\ [\text{T-APP}]$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : \mathsf{Close}(t_1, \Gamma) \vdash e_2 : t_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : t_2}\ [\text{T-LET}]$$

Processes $\boxed{\Gamma \vdash P}$

$$\frac{\Gamma \vdash e : \mathtt{unit}}{\Gamma \vdash \langle e \rangle}\ [\text{T-THREAD}] \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\ [\text{T-PAR}]$$

$$\frac{\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A.A \vdash P}{\Gamma \vdash \mathtt{new}\ a\ \mathtt{in}\ P}\ [\text{T-NEW}]$$

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 an endpoint is expected and they need not be typed in the same way (we will see why this is necessary in Example 5). As usual, there is no typing rule for `error`.

The rules for expressions derive judgments of the form $\Gamma \vdash e : t$ and are formulated using the same notation of Wright and Felleisen (1994). 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 $\mathsf{TypeOf}(\cdot)$, defined in Table 3, while that of names is obtained from the type environment. Following Wright and Felleisen (1994), 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] generalises the type of `let`-bound variables by means of the function $\mathsf{Close}(\cdot)$, defined as

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

and $\mathsf{ftv}(\cdot)$ collects the free type variables in 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 (Wright and Felleisen, 1994). 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 different (incompatible) types in different parts of the program. Conversely, [T-LET] is sound if we know that x is used linearly. Since the impure fragment of `FuSe` concerns only sessions and we are interested in stating the soundness of `FuSe` type system under the assumption that endpoints are indeed used linearly, we can generalise the type of an arbitrary `let`-bound expression e_1 even if e_1 has side effects.

Table 3. Type schemes of FuSe constants.

`()`	:	`unit`	`fork`	:	$\forall\alpha\,.\,(\alpha \to \texttt{unit}) \to \alpha \to \texttt{unit}$
`fix`	:	$\forall\alpha\beta\,.\,((\alpha\to\beta)\to\alpha\to\beta)\to\alpha\to\beta$	`create`	:	$\forall A\,.\,\texttt{unit}\to A * A^{\perp}$
`pair`	:	$\forall\alpha\beta\,.\,\alpha\to\beta\to\alpha * \beta$	`close`	:	`<•,•>`$\to$`unit`
`fst`	:	$\forall\alpha\beta\,.\,\alpha * \beta\to\alpha$	`send`	:	$\forall\alpha A\,.\,\alpha \to\, !\,[\alpha * A] \to A^{\perp}$
`snd`	:	$\forall\alpha\beta\,.\,\alpha * \beta\to\beta$	`receive`	:	$\forall\alpha A\,.\,?\,[\alpha * A] \to \alpha * A$
`inl`	:	$\forall\alpha\beta\,.\,\alpha\to\alpha + \beta$	`left`	:	$\forall AB\,.\,!\,[A + B] \to A^{\perp}$
`inr`	:	$\forall\alpha\beta\,.\,\beta\to\alpha + \beta$	`right`	:	$\forall AB\,.\,!\,[A + B] \to B^{\perp}$
`cases`	:	$\forall\alpha\beta\gamma\,.\,(\alpha+\beta)\to(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma$	`branch`	:	$\forall AB\,.\,?\,[A + B] \to A + B$

The function TypeOf is given in Table 3 as a set of associations `c` : TypeOf(`c`). The types on the lhs of Table 3 are standard and those of `fork` and `close` are unremarkable. 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 $\texttt{<}\alpha\,\texttt{,}\,\beta\texttt{>}$. Therefore, the desugared type scheme of `create` is $\forall\alpha\beta\,.\,\texttt{unit}\to\texttt{<}\alpha\,\texttt{,}\,\beta\texttt{>} * \texttt{<}\beta\,\texttt{,}\,\alpha\texttt{>}$. The ability to express channel types with unknown message types and capabilities gives `create` the most general type. The type of `send` follows from the encoding of outputs (Definition 6): `send` takes the payload of type α, a channel for sending messages of type $\alpha * A$, 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 ghost continuation is used to correlate the future behaviours of sender and receiver after this interaction. The type of `receive` follows from the encoding of inputs: in this case the type of the ghost 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 parametric polymorphism, once channel type variables have been desugared (Definition 3).

Example 3. 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 = <•,•>         *)
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 (n + m) x 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 (ghost) continuation of type $s_7^{\perp}$ to `client`. Therefore, $s_7^{\perp}$ describes the behaviour of `client` on the continuation channel, whereas `server` will use the same channel according to the type $s_2 = s_7$. Overall we derive `server` : $s \to$ `unit` where $s = \texttt{?[<•,•>} + \texttt{?[int} * \texttt{?[int} * \texttt{![int} * s^{\perp}\texttt{]]]]}$. Analogously, it is possible to derive `client` : `int` $\to s^{\perp} \to$ `int`. ■

Example 4. The code of the configurable forwarder in Example 2 does not depend on the nature of the messages being forwarded. Consequently, we can give `forwarder` a polymorphic type. To begin with, we define $t = \texttt{?}[\alpha * t]$ as the protocol type for receiving an infinite stream of messages of type α. This is the type of the `src` argument of `forwarder`. It is now possible to derive

$$\texttt{forwarder} : \texttt{bool} \to t \to \texttt{!}[t + \texttt{?}[t * \texttt{<}\bullet\texttt{,}\bullet\texttt{>}]] \to \texttt{unit}$$

where α can be generalised. Note the three occurrences of t. The leftmost and rightmost occurrences are the type of `src`, which is both an argument of `forwarder` and a message sent on `dst`, whereas the middle occurrence describes the ghost continuation of `dst`. ■

We now investigate the relationship between well-typed programs and the following properties: 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*). The interest is not in these properties *per se* – they are standard for session type systems – but in the fact that they are guaranteed (to some extent) by a type system without built-in support for session types. 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 either try to use endpoints non-linearly causing runtime errors or discard endpoints whose use is necessary to prevent deadlocks; second, deadlocks may occur in well-typed programs even if endpoint linearity is respected. To take these facts into account, we must strengthen the statements of our results with additional hypotheses: that endpoints are used linearly, and that no deadlocks occur. It is worth to recall that avoiding deadlocks through static analysis requires non-trivial extensions of the type system such as the use of typing rules that prevent cycles in the network topology (Wadler, 2014) or a richer type structure (Padovani, 2014).

The semantics of our communication primitives allows for a simple definition of *linear endpoint usage*. Recall that each communication primitive K using an endpoint a^p *invalidates* every other occurrence of a^p before (possibly) returning a^p itself. Therefore, any occurrence of $a^\star$ reveals the previous simultaneous existence of a different occurrence of a^p that has been used. 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 along all possible reductions of a process.

Definition 8 (endpoint affine/linear process)**.** Let $\mathscr{A}, \mathscr{L}$ be the largest predicates such that:

1. if either $\mathscr{A}(P)$ or $\mathscr{L}(P)$ and $P \equiv \texttt{new}\ a_1 \cdots a_n\ \texttt{in}\ (\langle \mathscr{E}[K\ a^p] \rangle \mid Q)$, then $p \in \{+, -\}$;
2. if $\mathscr{L}(P)$ and $P \equiv \texttt{new}\ a\ \texttt{in}\ Q$ and $a^p \in \mathsf{fn}(Q)$ and $p \in \{+, -\}$, then $a^{\overline{p}} \in \mathsf{fn}(Q)$;
3. for every $\mathscr{X} \in \{\mathscr{A}, \mathscr{L}\}$, if $\mathscr{X}(P)$ and $P \xrightarrow{\ell} Q$, then $\mathscr{X}(Q)$.

We say that P is *endpoint affine* if $\mathscr{A}(P)$ holds; that it is *endpoint linear* if $\mathscr{L}(P)$ holds.

In words: condition 1 states that no process that satisfies $\mathscr{A}$ or $\mathscr{L}$ ever tries to use invalid endpoints; condition 2 states that no process in $\mathscr{L}$ ever discards a valid endpoint if its peer does occur; condition 3 closes $\mathscr{A}$ and $\mathscr{L}$ under process reductions. Note that $\mathscr{L}(P)$

implies $\mathscr{A}(P)$ and that $\mathscr{L}$ is *coarser* than the property enforced by standard substructural type systems. In particular, duplications of an endpoint are allowed provided that only valid endpoints are actually used. For example, the expression `send (fst (pair` a^+ a^+`))` may occur in a process that satisfies $\mathscr{L}$ even though a^+ occurs twice. The same expression would be ill typed in a substructural type system like that of Gay and Vasconcelos (2010).

The type associated with endpoints may change each time they are used (this is common in session type systems). To reflect this change in the statement of subject reduction, we define a suitable reduction relation for type environments mimicking that of processes.

Definition 9. Let $\xrightarrow{\ell}$ be the least relation between type environments such that:

$$\begin{aligned}
\Gamma &\xrightarrow{\tau} \Gamma \\
\Gamma, a^p : \mathtt{![}t * s\mathtt{]}, a^{\overline{p}} : \mathtt{?[}t * s\mathtt{]} &\xrightarrow{\mathtt{m}ap} \Gamma, a^p : s^{\perp}, a^{\overline{p}} : s \\
\Gamma, a^p : \mathtt{![}t_1 + t_2\mathtt{]}, a^{\overline{p}} : \mathtt{?[}t_1 + t_2\mathtt{]} &\xrightarrow{\mathtt{m}ap} \Gamma, a^p : t_i^{\perp}, a^{\overline{p}} : t_i \quad i \in \{1,2\} \\
\Gamma, a^p : \mathtt{<\bullet,\bullet>}, a^{\overline{p}} : \mathtt{<\bullet,\bullet>} &\xrightarrow{\mathtt{c}a} \Gamma
\end{aligned}$$

We write $\Gamma \xrightarrow{\ell}$ if $\Gamma \xrightarrow{\ell} \Gamma'$ for some Γ' 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 $a^+, a^- \notin \mathsf{ep}(\ell)$ (once session a has been closed, no more interactions are allowed in it).

The last notion 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 have dual types and invalid endpoints have type $\forall A\,.\,A$:

Definition 10 (balanced type environment)**.** We say that Γ is *balanced* if:

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

The type system guarantees that all type environments used in a typing derivation are balanced (*cf.* rule [T-NEW]). The empty type environment, used for typing closed programs, is trivially balanced.

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

Example 5. To appreciate the importance of condition (2) in Definition 10, let

$$\begin{aligned}
P \stackrel{\text{def}}{=}\ & \langle \mathtt{close}\ (\mathtt{send}\ 42\ a^+)\rangle \mid \langle \mathtt{close}\ (\mathtt{snd}\ (\mathtt{receive}\ a^-))\rangle \\
& \mid \langle \mathtt{close}\ (\mathtt{send}\ 31\ (\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ c^+\ \mathtt{else}\ a^+))\rangle
\end{aligned}$$

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

$$\begin{aligned}
P \xrightarrow{\mathtt{m}a^+}\ & \langle \mathtt{close}\ a^+\rangle \mid \langle \mathtt{close}\ (\mathtt{snd}\ (\mathtt{pair}\ 42\ a^-))\rangle \\
& \mid \langle \mathtt{close}\ (\mathtt{send}\ 31\ (\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ c^+\ \mathtt{else}\ a^*))\rangle
\end{aligned}$$

where one occurrence of a^+ has been invalidated and

$$\Gamma \xrightarrow{\mathtt{m}a^+} a^+ : \mathtt{<\bullet,\bullet>}, a^- : \mathtt{<\bullet,\bullet>}, c^+ : t, c^- : t^\perp, a^* : \forall A\,.\,A, c^* : \forall A\,.\,A \stackrel{\text{def}}{=} \Gamma'$$

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'. ■

Communication safety is a straightforward consequence of typing. Following Gay and Hole (2005), we can formulate it thus:

Proposition 1 (type safety)**.** *Let $\Gamma \vdash P$ and $\mathscr{A}(P)$. Then:*

1. *if $P \equiv \mathtt{new}\ a_1 \cdots a_n\ \mathtt{in}\ (\langle \mathscr{E}[\mathtt{send}\ v\ u] \rangle \mid Q)$, then there exists Γ' such that $\Gamma, \Gamma' \vdash v : t$ and $\Gamma, \Gamma' \vdash u : \mathtt{!}[t * s]$;*
2. *if $P \equiv \mathtt{new}\ a_1 \cdots a_n\ \mathtt{in}\ (\langle \mathscr{E}[\mathtt{left}\ u] \rangle \mid Q)$ or $P \equiv \mathtt{new}\ a_1 \cdots a_n\ \mathtt{in}\ (\langle \mathscr{E}[\mathtt{right}\ u] \rangle \mid Q)$, then there exists Γ' such that $\Gamma, \Gamma' \vdash u : \mathtt{!}[t + s]$.*

Protocol fidelity follows immediately from Theorem 2 and the observations below Definition 9: if an ℓ-labelled 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. Note that endpoint affinity suffices for proving both communication safety and protocol fidelity of well-typed programs. Concerning progress, endpoint linearity is necessary but not sufficient because of deadlocks, for which we provide the following syntactic characterisation.

Definition 11 (deadlock)**.** We say that P is *deadlocked* if

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

where $I \neq \emptyset$ and for every $i \in I$ there exists $j \in I$ such that $c_i^{\overline{p}_i} \in \mathsf{fn}(\mathscr{E}_j) \cup \mathsf{fn}(K_j)$.

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 $\mathscr{L}(P)$, then either there exists Q such that $P \xrightarrow{\tau} Q$ or $P \equiv \langle () \rangle$ or P is deadlocked.*

We now discuss some notable classes of processes that violate either $\mathscr{A}$ or $\mathscr{L}$.

Example 6 (affinity and linearity violations)**.** The condition $\neg\mathscr{A}(P)$ indicates that P attempts to use some endpoint more than once, in a way that disrespects the explicit threading of continuations required by the communication primitives. We call this event an *overlap*, of which two instances can be generated by the code fragments below:

```
let foo x =
  let _ = send 42 x in
  let x = send 43 x in
  close x
```

```
let bar y =
  let _ = send 42 y in
  let _, y = receive y in
  close y
```

The function `foo` can be typed giving `x` type `![int * <•,•>]`, 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 types (`<•,int * <•,•>>` for `send`, `<int * <•,•>,•>` 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 go undetected are those concerning multiple uses of the same communication primitive with the same message types. Section 5.2 discusses how these can be detected at runtime.

The condition $\mathscr{A}(P) \wedge \neg\mathscr{L}(P)$ indicates that P respects endpoint affinity but discards valid endpoints that may be necessary to have progress. For example, the program

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

is well typed but violates $\mathscr{L}$ because it 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). ■

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 generalised choices (Section 5.3). In the second half of the section we discuss whether and how the implementation scales to a distributed setting (Section 5.4), we show how to build a simple monadic API on top of `FuSe` primitives (Section 5.5), and we develop a microbenchmark to measure the overhead of the runtime checks and of the monadic API (Section 5.6).

5.1 The basics

The basic version of the `OCaml` module that implements the `FuSe` communication primitives is shown in full in Figure 1. The interface exports the abstract type • (line 1) and the abstract channel type `t` (line 2). In `OCaml`, the channel type `<t,s>` is written `(t,s) t` and we use the polymorphic variant type `[`L of t | `R of s]` to represent the sum type $t + s$. Polymorphic variants (Garrigue, 1998) easily generalise sums to arbitrary tags and support a form of subtyping that is consistent with subtyping for session types (Gay and Hole, 2005). We will see these features at work in Section 5.3.

The types of the primitives (lines 3–9) are essentially those shown in Table 3, so we only make a few remarks. First, all type variables are implicitly quantified. Second, as in `FuSe` types, we can switch from one channel type to its dual by flipping its two type parameters (see *e.g.* the type of `create` on line 3). Finally, in the type of `branch` (line 9), 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

Interface

```
1  type •           (* no message   *)
2  type (α,β) t     (* channel type *)
3  val create     : unit → (α,β) t * (β,α) t
4  val close      : (•,•) t → unit
5  val send       : φ → (•,φ * (α,β) t) t → (β,α) t
6  val receive    : (φ * (α,β) t,•) t → φ * (α,β) t
7  val left       : (•,[`L of (α,β) t | `R of (γ,δ) t]) t → (β,α) t
8  val right      : (•,[`L of (α,β) t | `R of (γ,δ) t]) t → (δ,γ) t
9  val branch     : ([`L of (α,β) t | `R of (γ,δ) t] as ε,•) t → ε
```

Implementation

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

Fig. 1. OCaml implementation of FuSe (basic version).

within t, the same construction is also used in OCaml for creating recursive types. We will see an instance of this feature in Example 7.

We have based the implementation of the primitives on the Event module in OCaml's standard library, which provides an API for communication and synchronisation in the style of Concurrent ML (Reppy, 1999). 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 communication to actually take place, both the sender and the receiver must *synchronise* by applying Event.sync to such events.

The representation of a channel type <α,β> is unit Event.channel (line 11), namely FuSe endpoints are Event.channels in OCaml. Note that α and β play no role in the representation of channel types, since neither of them faithfully describes the messages actually exchanged on the channel. For this reason, we just pretend that exchanged messages have type unit and will perform suitable casts in the communication primitives. Polarities are not represented either, they are an artefact of the formal model so that peer endpoints can be typed differently (we will partially reconsider this choice in Section 5.2).

The implementation of create (line 12) and close (line 13) is unremarkable: 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 14), we perform an unsafe cast on the message `x` using `Obj.magic : α → β` so that `x` appears to the type checker as having type `unit`. The cast cannot interfere with the internals of the `Event` module: since the `Event.channel` type is *parametric* in the type of messages, we know that `Event` functions make no assumptions on their content. 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. This latter cast is potentially dangerous, for it changes the type of an existing endpoint which might be used elsewhere according to its previous type. From Section 4 we know that this cast is safe provided that no other reference to `u` occurs in the program, namely if `u` is used linearly. In Section 5.2 we will implement the endpoint invalidation semantics of the formal model, so that any potentially unsafe linearity violation is at least signalled with a runtime exception. The trickery in `send` induces a corresponding implementation of `receive` (line 15): `OCaml` believes that the event created by `Event.receive` yields `()`, whereas an actual payload is received. We explicitly pair the endpoint `u` (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 16–18) 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 by the receiver and resorts to one last magic: since the internal `OCaml` representation of `` `T u `` – that is channel `u` injected through the `T` tag – is the same as that of the pair `` (`T, 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 received message and scales to arbitrary tag sets without requiring subsequent patches (see Section 5.4).

Example 7 (session type inference and duality)**.** Below are the types of `server` and `client` from Example 1 automatically inferred by `OCaml`:

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

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

These two types correspond exactly to those we have guessed in Example 3. Since the channel type in the type of `server` is dual of the channel type in the type of `client`, we deduce that `client` and `server` interact safely and the function `main` at the end of Example 1 is well typed.

Consider now a variation of `client` where the initial value for the partial result `acc` is `0.0` instead of `0`. The effect of this change is to turn `acc`'s type from `int` to `float`:

```
let client' =
  let rec aux acc n y = ... in aux 0.0
```

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 (•,•) t ] as α) t → float
```

However, the channel types of `client'` and `server` are no longer 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. ■

Example 8 (sample errors)**.** To show the effectiveness of the library in detecting programming errors involving session endpoints, let us consider once again the `forwarder` function of Example 2, which for convenience we report once again with numbered lines.

```
1  let forwarder mode src dst =
2    if mode then                          (* check modality      *)
3      let rec aux src dst =
4        let msg, src = receive src in     (* receive from source *)
5        let dst = send msg dst in         (* send to destination *)
6        aux src dst                       (* forever             *)
7      in aux src (left dst)               (* select forwarding   *)
8    else
9      close (send src (right dst))        (* select delegation   *)
```

We discuss a few non-obvious errors that may be introduced in `forwarder`:

- Omitting `left` (line 7) would cause a type error, for `dst` would be used for both an output operation (line 5) and a selection (line 9). The omission of `right` (line 9) would cause a type error for similar reasons.
- Omitting *both* `left` and `right` would also cause a type error, because `dst` would be used repeatedly for output operations in `aux` (line 5) and would be closed on line 9.
- Omitting `left`, `right` and also `close` would *not* cause a type error, because the type of the (unused) continuation of `dst` (line 9) would be unifiable with that of `dst` as used in `aux`. In general, closing unused endpoints explicitly helps detecting more errors in lack of a substructural type system.
- Swapping `msg` and `dst` (line 5) would not cause a type error, although `OCaml` would infer for `forwarder` a weird-looking type. However, this error would go undetected also in a substructural type system such as the one of Gay and Vasconcelos (2010).
- Omitting the rebinding of `dst` (*i.e.*, replacing the leftmost occurrence of `dst` on line 5 with `_`) would go unnoticed, because the type of `dst` is invariant at each recursion of `aux`. This is an instance of overlap which could be detected at runtime with the elaboration of the library that we are going to present in Section 5.2.

Although this list is by no means exhaustive and the notion of "non-obvious error" is subjective, the experience gained in coding examples (those presented in the paper and others included in `FuSe`) suggests that the session typing discipline realised by `FuSe` provides valuable feedback even if the host type system is uncapable of detecting all affinity and linearity violations at compile time. ■

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

```
1  exception InvalidEndpoint
2  type •                (* no representation *)
3  type (α,β) t        = unit Event.channel * Mutex.t
4  let check f         = if not (Mutex.try_lock f) then raise InvalidEndpoint
5  let fresh u         = (u, Mutex.create ())
6  let create ()       = let u = Event.new_channel () in (fresh u, fresh u)
7  let close (_, f)    = check f
8  let send x (u, f)   = check f; ...; Obj.magic (fresh u)
9  let receive (u, f)  = check f; Obj.magic (..., fresh u)
10 let left (u, f)     = check f; ...; Obj.magic (fresh u)
11 let right (u, f)    = check f; ...; Obj.magic (fresh u)
12 let branch (u, f)   = check f; Obj.magic (..., fresh u)
```

Fig. 2. `OCaml` implementation of `FuSe` (with invalid endpoint usage detection).

Figure 2 implements the invalidation semantics assuming that endpoints reside in shared memory, as is the case when using the `Event` module. The idea is to represent endpoints as pairs consisting of an `Event.channel` and a mutable flag that approximates the endpoint polarity and indicates whether the endpoint is valid or not (line 3). To account for the possibility that the endpoint is accessed concurrently by multiple threads, we represent the flag as a `Mutex` from `OCaml`'s standard library. Whenever a thread uses an endpoint, we attempt to acquire the mutex with `try_lock`. If the operation returns `true`, then the endpoint is valid and can be used; if `try_lock` returns `false`, then the endpoint has already been used and an exception is raised. The auxiliary function `check` (line 4) implements this behaviour.

Ideally, when a communication primitive returns a continuation endpoint, the mutex associated with the endpoint should be unlocked, but doing so on the existing mutex might induce other users of the endpoint into thinking that the endpoint 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 and a new (unlocked) mutex. This refreshing is implemented by the auxiliary function `fresh` (line 5). In essence, the cost we pay for detecting the usage of invalid endpoints is the allocation of a new pair and a mutex at each invocation of a communication primitive. As we will see in Section 5.6, this overhead is reasonable especially in a functional language, where the runtime system is optimised for the heap allocation of numerous small objects.

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 7–12). 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 choice of mutexes in the above discussion is aimed at describing an implementation of the endpoint invalidation semantics that is correct, simple and portable at the same time. However, more efficient mechanisms can be considered in specific settings. For example, one could replace the mutex with a boolean reference modified through an atomic compare-and-swap (CAS) operation. As one of the anonymous reviewers pointed out, in

the particular case of OCaml using a CAS is not even necessary, provided that no memory allocation intervenes between the moments the reference is read and written. The current implementation of FuSe follows this latter strategy.

5.3 Generalised 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 generalising 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 `select` primitive and revise `branch` so that `select` and `branch` have these types:

```
val select : ((α,β) t → φ) → (•,[>] as φ) t → (β,α) 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 $\langle\alpha,\beta\rangle\to\varphi$ which produces the message, of type φ, when applied to a continuation endpoint of type $\langle\alpha,\beta\rangle$. Typically, such function will be the η-expansion of a tag

```
fun x → `T 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`,`φ`) t` unifiable with the type `(`α`,`ψ `* (`γ`,`δ`) t) t` so that 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 f u = Event.sync (Event.send u (Obj.magic f)); Obj.magic u
let branch u   = Obj.magic (Event.sync (Event.receive u)) u
```

Example 9. 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 x
  | `Plus x → let n, x = receive x in
              let m, x = receive x in
              let x = send (n + m) x in
              server x
  | `Eq x →   let n, x = receive x in
```

```
    let m, x = receive x in
    let x = send (n = m) x in
    server x

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

For these two functions, `OCaml` infers the following types:

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

val client :
  int → (•,[> `Plus of (int * (int * (•,int * (•,α) t) t,•) t,•) t
           |  `Quit of (•,•) t ] as α) t → 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` uses 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 (Gay and Hole, 2005): 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 the encoding of session types (Dardha et al., 2012) to lift `OCaml`'s built-in subtyping of variant types at the level of channel (hence session) types. The implementation of `FuSe` pushes support

for subtyping even further, by taking advantage of OCaml's variance annotations for type parameters. In particular, the abstract channel type is declared thus

```
type (+α,-β) t
```

indicating that channel types are covariant in the input type and contravariant in the output type. These annotations induce, on decoded channel types, the same subtyping relation for session types described by Gay and Hole (2005). ■

5.4 Support for distributed communications

From a purely theoretical viewpoint, our approach to session type checking applies equally well to both local and distributed communications. In practice, however, the underlying communication API may rely on a specific distribution model and/or pose restrictions to the data that can be sent in messages. For example, the Event module relies on shared memory and therefore is unfit to support distributed communications. In this section, we argue that our approach remains feasible also in a distributed setting.

First of all, we have to consider that in a distributed system any data that cannot be serialised in a platform-neutral way might be meaningless if transmitted in a message. For a higher-order language such as OCaml, the typical example is that of functions, whose representation involves compiled code. In this respect, the handling of arbitrary tags that we have described in Section 5.3 does not scale well to a distributed setting, because it is realised by sending functions over channels. Two alternatives remain feasible though. The functions left, right, and branch in the basic version of the library (Section 5.1) handle binary choices and branches by transmitting tags only, and are therefore unproblematic in a distributed setting. These functions are also easy to generalise to arbitrary – but fixed – tag sets. For example, given the set $L = \{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$ of tags we can devise the following family of OCaml functions to handle choices involving the tags taken from L:

```
let select_{L,i} u = Event.sync (Event.send u (Obj.magic `T_i)); Obj.magic u
let branch_L  u = Obj.magic (Event.sync (Event.receive u), u)
```

These functions can be given the types

```
val select_{L,i} : (•,tags_L) t → (β_i,α_i) t
val branch_L  : (tags_L,•) t → tags_L
```

where $\mathtt{tags}_L$ abbreviates occurrences of the polymorphic variant type

```
[ `T_1 of (α_1,β_1) t | ⋯ | `T_n of (α_n,β_n) t ]
```

In the end, it is easy to define domain-specific versions of select and branch that involve the transfer of only a tag and that are suitable for distributed communications.

The mechanism described in Section 5.2 for the runtime detection of linearity violations is also delicate. Assuming that the underlying communication API allows endpoints to be sent in messages, the mechanism we have described would require keeping the flags associated with each endpoint coherent, which is difficult to do across different locations. We can adapt the mechanism to a distributed setting by devising dedicated primitives for the communication of endpoints. 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 effectively and efficiently detected using the code shown in Figure 2. This behaviour can be realised by the two functions below

```
let send_endpoint (u, f) (v, g) =
  check f; check g; Event.sync (Event.send v (Obj.magic u));
  Obj.magic (fresh v)
let receive_endpoint (u, f) =
  check f; let v = Event.sync (Event.receive u) in
  Obj.magic (fresh v, fresh u)
```

whose types are

```
val send_endpoint    : (γ,δ) t → (•,(γ,δ) t * (α,β) t) t → (β,α) t
val receive_endpoint : ((γ,δ) t * (α,β) t,•) t → (γ,δ) t * (α,β) t
```

Of course, it is up to the programmer to remember using the dedicated primitives for delegations, unless the type system has some feature that prevents channels to be sent as ordinary messages. The use of dedicated primitives for handling delegations may be motivated by other practical considerations, since the communication of endpoints in a distributed environment is likely to require some special handling anyway, and it has been considered in theoretical works on sessions as well (Honda et al., 1998).

5.5 A monadic interface for `FuSe` *primitives*

In this section we elaborate a monadic API on top of `FuSe` communication primitives. Following Pucella and Tov (2008), we define an indexed monad `(α,β,φ) m` that describes computations producing a value of type φ while using a session endpoint and turning its type from α to β. The monad encapsulates the session endpoint and hides it from the programmer, threading computations in such a way that the endpoint is guaranteed to be used linearly, unless an exception is thrown (affinity is always guaranteed though).

Figure 3 shows the `OCaml` module realising the monadic API. The interface of the module comprises the (abstract) monad type and the signatures of the usual monadic operations `return`, to create trivial computations not involving communications, and `>>=`, to compose computations sequentially. In addition, it is convenient to provide a specialisation `>>>` of `>>=` in which the continuation does not use the result of the first computation as well as a fixpoint operator `m_fix` (we will see in a moment why this is useful). Next we have the operations for session communications. Primitives `m_send` through `m_branch` correspond exactly to `FuSe` communication primitives, modulo the fact that output operations have an explicit `unit` return type signalling that they produce no result. The operation `m_connect` combines the functionality of `create` and `close`. When `m_connect` is applied to two monadic computations, corresponding to the "server" and "client" sides of the session, it creates a new session that connects server and client, spawns a new thread for the server and runs the client. Once the interaction is over, both endpoints of the session are closed and the result of the client computation is returned.

The implementation of the module is shown in the lower part of Figure 3. The concrete representation of the indexed monad `(α,β,φ) m` is $\alpha \to \varphi * \beta$, namely a function that

Interface

```
type (α,β,φ) m
val return          : φ → (α,α,φ) m
val (>>=)           : (α,β,φ) m → (φ → (β,γ,ψ) m) → (α,γ,ψ) m
val (>>>)           : (α,β,φ) m → (β,γ,ψ) m → (α,γ,ψ) m
val m_fix           : ((α,β,φ) m → (α,β,φ) m) → (α,β,φ) m
val m_connect       : ((α,β) t,(•,•) t,unit) m → ((β,α) t,(•,•) t,φ) m → φ
val m_send          : φ → ((•,φ * (α,β) t) t,(β,α) t,unit) m
val m_receive       : ((φ * (α,β) t,•) t,(α,β) t,φ) m
val m_left          : ((•,[`L of (α,β) t | `R of (γ,δ) t]) t,(β,α) t,unit) m
val m_right         : ((•,[`L of (α,β) t | `R of (γ,δ) t]) t,(δ,γ) t,unit) m
val m_branch        : ((α,β) t,ε,φ) m → ((γ,δ) t,ε,φ) m →
                      (([`L of (α,β) t | `R of (γ,δ) t],•) t,ε,φ) m
```

Implementation

```
type (α,β,φ) m      = α → φ * β
let return v u      = (v, u)
let (>>=) m f u     = let v, u = m u in f v u
let (>>>) m n       = m >>= (fun _ → n)
let rec m_fix f     = f (fun u → m_fix f u)
let m_connect s c   = let us, uc = create () in
                      let _ = fork (close ∘ snd ∘ s) us in
                      let x, uc = c uc in close uc; x
let m_receive       = receive
let m_send x u      = ((), send x u)
let m_left u        = ((), left u)
let m_right u       = ((), right u)
let m_branch l r u  = match branch u with
                      | `L u → l u
                      | `R u → r u
```

Fig. 3. `OCaml` implementation of `FuSe` (monadic version).

takes a session endpoint and produces a pair with the result of the computation and the endpoint with the updated type. This informal description essentially corresponds to that of the state monad, with the difference that in our case the state is a session endpoint whose type may change as the session goes on. Once the representation of the monad is fixed, the implementation of the operations follows directly. Just note the use of functional composition ∘ in `m_connect` and the explicit return of `()` in the output operations `m_send`, `m_left`, and `m_right`.

Example 10 (monadic mathematical server)**.** Below we show the monadic version of the mathematical server introduced in Example 1.

```
let server =
  m_fix                          (* recursive server    *)
    (fun server →
      m_branch                   (* wait for a request  *)
        (return ())              (* terminate session   *)
        (m_receive >>= fun n → (* receive 1st operand *)
```

```
      m_receive >>= fun m → (* receive 2nd operand *)
      m_send (n + m) >>>    (* send result         *)
      server))              (* serve more requests *)
```

Note the use of `m_fix` for creating a recursive server. In this case it would not be possible to define `server` recursively as we have done in Example 1, for the body of `server` is not a function (even if it evaluates to a function, as we know from Figure 3).

The monadic version of the client in Example 1 is shown below

```
let client =
  let rec aux acc n =          (* add n naturals      *)
    if n = 0 then
      m_left >>> return acc    (* terminate session   *)
    else
      m_right >>>              (* select + operation  *)
      m_send acc >>>           (* send 1st operand    *)
      m_send n >>>             (* send 2nd operand    *)
      m_receive >>= fun res →  (* receive result      *)
      aux res (n - 1)          (* possibly add more   *)
  in aux 0
```

and, with these definitions in place, client and server can be connected thus:

```
let main n =
  m_connect server (client n)  (* run session         *)
```

The slightly awkward syntax for binding results of computations on the right hand side of >>= is typical of monadic code but can be easily sugared into a more readable form. ■

As shown in Example 10, the monadic API can be convenient in simple (but possibly frequent) cases involving single sessions. Because endpoint affinity is guaranteed by the monad, the monadic API can be safely built on top of the basic version of `FuSe` primitives (Figure 1), thus sparing the overhead due to runtime checks. In fact, as we will see in Section 5.6, this results in a slightly more efficient implementation compared to that described in Section 5.2. Also, the chosen realisation of the `m_connect` operation is known to guarantee deadlock freedom, provided that no exceptions interrupt ongoing sessions (Wadler, 2014). The monadic API has some downsides as well. First and foremost, it disallows the interleaving of multiple sessions and, in particular, it does not support delegation. Multiple sessions can be opened and used according to a strictly nested discipline, pretty much like function calls. As shown by Pucella and Tov (2008), the monad can be generalised to support multiple sessions at the cost of a more complex API requiring the programmer to write significant amounts of boilerplate code. Also, it is difficult to provide a monadic API supporting generalised choices because not only the typing but also the implementation of `m_branch` depends on the set of tags that must be handled. As explained in Section 5.3, however, it is possible to provide versions of `m_branch` handling a specific set of tags.

5.6 Performance comparison

In this section we report on a series of microbenchmarks aimed at measuring the performance of different implementations of `FuSe` communication primitives. The program

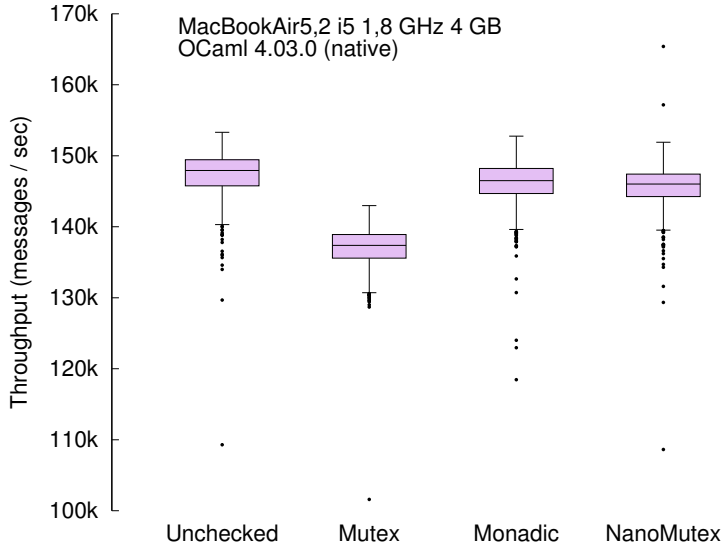

Fig. 4. Performance comparison of different implementations of `FuSe` primitives.

we use as benchmark is the one presented in Example 1 (and its monadic variant in Example 10) in which a client interacts with a mathematical server to compute the sum of the first n natural numbers. For the benchmarks we fix n at 1000 and we measure the message throughput of 1000 runs, considering that each execution involves the exchange of $4n+1$ messages. The reason for such a high number of runs is to contrast the broad variance observed in some execution times, which is due to uncontrollable factors such as the intervention of `OCaml`'s garbage collector and the involvement of concurrent threads using synchronisation primitives.

Figure 4 shows the results of the microbenchmarks corresponding to 4 implementations of the `FuSe` primitives. Each box contains the throughput of 750 runs, the horizontal line being the median and the individual dots stretching beyond the whiskers being outliers. The "Unchecked" column corresponds to the basic version of the library (Figure 1) which does not perform runtime checks to detect linearity violations. This column represents the ideal performance that could be achieved if `OCaml` had support for linear/affine types, thus eliminating the need for runtime checks or other mechanisms (such as the use of the monad) to enforce endpoint linearity/affinity. The "Mutex" column corresponds to the implementation shown in Figure 2, which checks for linearity violations using a mutex from `OCaml`'s standard library. The graph shows a 10% throughput decrease due to the overhead of runtime checks and the creation of a new mutex at each communication. The "Monadic" column corresponds to the implementation shown in Figure 3 in which the non-monadic primitives are those from Figure 1. The throughput in this case is almost ideal, despite the fair amount of closures that are created by this version of the library. Finally, the "NanoMutex" column corresponds once again to the implementation shown in Figure 2 except that the mutexes being used are `Nano_mutexes` taken from `Core` (Jane Street Developers, 2016), an alternative to `OCaml`'s standard library. `Nano_mutexes` are significantly cheaper than the OS-level mutexes used in the `Mutex` module of `OCaml`'s

standard library. In fact, the throughput achieved by this implementation of the library is almost ideal. This column shows that a careful handling of the validity flag associated with session endpoints can virtually eliminate the overhead due to the runtime checks.

6 Related work

Several Haskell libraries of binary sessions have been proposed (Neubauer and Thiemann, 2004; Sackman and Eisenbach, 2008; Pucella and Tov, 2008; Imai et al., 2010). In these works, session types are represented conventionally as sequences of I/O actions and branching points and duality constraints are expressed either using multiparameter type classes and functional dependencies (both of which are Haskell-specific features) or explicit duality proofs (Pucella and Tov, 2008). The lack of equi-recursive types in Haskell calls for an explicit representation also for recursive session types. To this aim, Pucella and Tov (2008) and Imai et al. (2010) use De Bruijn indexes and type-level Peano numerals. This representation requires programmers to write explicit monadic actions for unwinding recursions and induces an iso-recursive treatment of session types.

The standard way of expressing impure computations in Haskell is to embed them in a monad. All Haskell libraries of binary sessions follow this approach. Besides being a necessity dictated by the nature of the language, the monad for session communications guarantees affine access to session endpoints, which are hidden to the programmer, and tracks the changes in the type of session endpoints automatically without requiring the programmer to write explicit rebindings (compare the code in Examples 1 and 10). However, the monad has also a cost in terms of either expressiveness, usability, or portability: the monad defined by Neubauer and Thiemann (2004) supports communication on a single channel only and is therefore incapable of expressing session interleaving or delegation. Pucella and Tov (2008) propose a monad that stores a stack of endpoints (or, better, of their capabilities) allowing for session interleaving and delegation to some extent. The price for this generality is that the programmer has to write explicit monadic actions that literally dig into the stack to reach the channel/capability to be used; also for this reason delegation is severely limited. Imai et al. (2010) show how to avoid writing such explicit actions relying on a form of type-level computations that is unique to Haskell.

Alms (Tov and Pucella, 2011; Tov, 2012) is a general-purpose programming language equipped with an expressive type system that supports parametric polymorphism, abstract and algebraic data types, and affine types as well. Tov (2012) illustrates how to build a library of binary sessions on top of these features. Because Alms' type system is substructural, endpoint affinity is guaranteed statically by the fact that session types are qualified as affine. As in the works about Haskell, the representation of session types is conventional and recursive session types arise from Alms support for equi-recursive types. Duality constraints are expressed using a peculiar feature of Alms called *type functions* with which the programmer can define (lazy) type-level computations. In particular, Tov gives a type function corresponding to Definition 4 that allows the compiler to compute the dual of a session type. We observe that the mechanism of type functions is currently used in Alms' library for this purpose only. By using encoded session types, it would be possible to obtain a fully fledged library of binary sessions that makes no use of type functions.

The main source of inspiration for this work is the continuation-passing encoding of binary sessions given by Dardha et al. (2012) and partially studied by Kobayashi (2002) and Demangeon and Honda (2011). Dardha et al. (2012) motivate the encoding as a foundational approach to the study of session type systems and their properties. We have shown that the very same encoding has also practical relevance: one the one hand, it permits the integration of session type checking into a wide class of programming languages; on the other hand, it allows us to benefit from the features of the host language – parametric polymorphism, equi-recursive types, subtyping, type inference – lifting them at the level of (encoded) session types. From a technical standpoint, our work differs from that of Dardha et al. (2012) in two ways. First, we use a new representation of channel types that makes it possible to reduce type duality to type equality. Second, we consider the standard operational semantics of session communications (Gay and Vasconcelos, 2010), in which no continuation channels are explicitly created or exchanged, also taking into account endpoint invalidation. For these reasons, the soundness results given by Dardha et al. (2012) cannot be used as arguments for the soundness of our typing discipline.

Concerning the enforcement of linear/affine usage of session endpoints, our approach is based on runtime checks (Tov and Pucella, 2010; Hu and Yoshida, 2016). By definition, this approach cannot detect linearity violations statically, but it allows for greater flexibility when used in conjunction with constructs for non-local control flow (most notably exceptions and callbacks), which are known sources of issues for substructural type systems. The proposed runtime mechanism is reasonably lightweight (Section 5.6) and is independent of the particular representation of session types and/or the mechanism used for tracking the change in type of session endpoints, making it is easy to apply in a wide range of programming languages (Hu and Yoshida, 2016). Besides, the fact that the session type of an endpoint changes at each usage allows the detection of a fair number of linearity violations even if the underlying type system is not substructural (Example 6). Our approach can be seen as a particular instance of the framework of stateful contracts proposed by Tov and Pucella (2010) in that our representation of session endpoints comprises both the affine/linear data structure (the session endpoint) and its stateful contract (the mutex). We note two differences: first, we have to use a mutex (or an equivalent data type with atomic access) instead of a bare flag to account for multithreading; second, we use polymorphism to account for the possibility that multiple references to the same endpoint coexist but require different typings (Example 5). The endpoint invalidation semantics and the typing of invalid endpoints are original contributions of the present work.

A different but related technique is the *runtime monitoring* of sessions (Chen et al., 2011; Bocchi et al., 2013; Demangeon et al., 2015; Bartoletti et al., 2015). Runtime monitoring is achieved either by a service (Chen et al., 2011; Bocchi et al., 2013; Demangeon et al., 2015) or by an active communication middleware (Bartoletti et al., 2015) that compares the observable behaviour 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 do not necessarily imply corresponding protocol violations.

There is a long strand of works which follow a *top-down* methodology for integrating sessions into programming languages. In these works, the protocol comes first and is

explicitly described either formally or by means of a suitable DSL. From this description, the communication API (sometimes in the form of a class hierarchy) is generated, possibly with the aid of supporting tools. A comprehensive survey describing works that follow this methodology is given by Ancona et al. (2016). A notable omission from the survey that targets `Scala` and that is very related to our approach is a recent work by Scalas and Yoshida (2016). As we do, Scalas and Yoshida use a runtime mechanism to compensate for the lack of affine/linear types in `Scala` and work with the encoded representation of session types given by Dardha et al. (2012). Each (encoded) session type is represented as a `Scala` (case) class, which must be either provided by the programmer or generated from the protocol. Since `Scala`'s type system is nominal, the subtyping relation between session types is constrained by the (fixed) subclassing relation between the classes that represent them. Two more differences are worth noting. First, the framework of Scalas and Yoshida relies on the explicit creation and exchange of continuations. Second, it illustrates the sort of subtleties arising when channel types are represented using fixed capabilities `?` and `!` (*cf.* the discussion in Section 3). For instance, it is possible to write functions with types $\texttt{unit} \to \texttt{?}[\alpha] * \texttt{![}\alpha\texttt{]}$ and $\texttt{unit} \to \texttt{![}\alpha\texttt{]} * \texttt{?}[\alpha]$, but not a function with type $\texttt{unit} \to A * A^{\perp}$ that mentions completely unknown yet related session types. As we have seen in Section 4, our representation of channel types is key to express the most general type of the communication primitives and the functions built on them.

7 Concluding remarks

The implementation of `FuSe` supports other features not described in this paper, including `accept`/`request` primitives to open sessions via shared channels (Honda et al., 1998; Gay and Vasconcelos, 2010). The use of a single primitive `create` to open new sessions is technically simpler and has been inspired by Singularity OS (Hunt et al., 2005; Bono and Padovani, 2012). The implementation of `FuSe` also supports *sequential composition* and *iteration* for protocols. Sequential composition can be used for the enforcement of *context-free protocols* (Vasconcelos and Thiemann, 2016; Padovani, 2016) whereas iterative protocols provide alternative ways of realising arbitrarily long interactions without resorting to (equi-)recursive types. In general we argue that, by reducing session type duality to type equality, our approach makes it easy to conceive and implement high-level communication patterns that require the execution of complementary behaviours.

The choice of synchronous communication in `FuSe` was solely motivated by convenience. Asynchronous communication can be modelled using queues (Gay and Vasconcelos, 2010), adjusting the formal semantics so that the peers of a session are invalidated independently, and basing the implementation on a suitable asynchronous API.

An intriguing extension concerns *multiparty sessions* (Honda et al., 2008), those involving an arbitrary, possibly variable number of participants. The runtime mechanism to detect linearity violations is applicable without issues to the endpoints of multiparty sessions. Concerning the representation of the protocol structure, it is known that (some classes of) multiparty sessions can be encoded in terms of binary sessions either connecting pairs of participants (Padovani, 2014, extended version) or connecting each participant with a medium process (Caires and Pérez, 2016). As it stands, our approach would be unable

to recognise these individual sessions as part of a single multiparty session. Whether the approach can be extended to model "true" multiparty sessions remains an open question.

Acknowledgments. The author is grateful to the anonymous reviewers and to Hernán Melgratti for having provided detailed and constructive feedback on early versions of this paper. Hernán Melgratti also contributed to the development of `FuSe`.

A Supplement to Section 3

Lemma 1. *We have* $\bot_{\mathsf{ct}} \circ [\![\cdot]\!] = [\![\cdot]\!] \circ \bot_{\mathsf{st}}$ *and* $\bot_{\mathsf{st}} \circ \langle\!\langle \cdot \rangle\!\rangle = \langle\!\langle \cdot \rangle\!\rangle \circ \bot_{\mathsf{ct}}$.

Proof. We only show the first equality. In particular, we show that $[\![T]\!]^\bot = [\![T^\bot]\!]$ by case analysis on the shape of T.

$\boxed{T = ?t\,.\,S}$ We derive:

$$\begin{array}{rcll}
[\![T]\!]^\bot & = & [\![?t\,.\,S]\!]^\bot & \text{by definition of } T \\
& = & ?\,[t * [\![S]\!]]^\bot & \text{by definition of } [\![\cdot]\!] \\
& = & !\,[t * [\![S]\!]] & \text{by definition of duality on channel types} \\
& = & !\,[t * [\![S^{\bot\bot}]\!]] & \text{because duality is an involution} \\
& = & [\![!t\,.\,S^\bot]\!] & \text{by definition of } [\![\cdot]\!] \\
& = & [\![T^\bot]\!] & \text{by definition of duality on session types}
\end{array}$$

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

$$\begin{array}{rcll}
[\![T]\!]^\bot & = & [\![T_1 \oplus T_2]\!]^\bot & \text{by definition of } T \\
& = & !\,[[\![T_1^\bot]\!] + [\![T_2^\bot]\!]]^\bot & \text{by definition of } [\![\cdot]\!] \\
& = & ?\,[[\![T_1^\bot]\!] + [\![T_2^\bot]\!]] & \text{by definition of duality on channel types} \\
& = & [\![T_1^\bot \,\&\, T_2^\bot]\!] & \text{by definition of } [\![\cdot]\!] \\
& = & [\![T^\bot]\!] & \text{by definition of duality on session types.}
\end{array}$$

The remaining are either similar or trivial. □

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

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 and we consider only the case $T = !t\,.\,T'$, the others being analogous. We have to show that there exists S' such that $S = !t\,.\,S'$ and $S'\,\mathscr{R}\,T'$. We derive:

$$\begin{array}{rcll}
S = \langle\!\langle [\![T]\!] \rangle\!\rangle & = & \langle\!\langle [\![!t\,.\,T']\!] \rangle\!\rangle & \text{by definition of } T \\
& = & \langle\!\langle !\,[t * [\![T'^\bot]\!]] \rangle\!\rangle & \text{by definition of } [\![\cdot]\!] \\
& = & !t\,.\,\langle\!\langle [\![T'^\bot]\!]^\bot \rangle\!\rangle & \text{by definition of } \langle\!\langle \cdot \rangle\!\rangle \\
& = & !t\,.\,\langle\!\langle [\![T'^{\bot\bot}]\!] \rangle\!\rangle & \text{by Lemma 1} \\
& = & !t\,.\,\langle\!\langle [\![T']\!] \rangle\!\rangle & \text{because duality is an involution}
\end{array}$$

and we conclude by taking $S' \stackrel{\text{def}}{=} \langle\!\langle [\![T']\!] \rangle\!\rangle$ and observing that $S'\,\mathscr{R}\,T'$ by definition of $\mathscr{R}$.

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 $S \mathscr{R} T$. By definition of $\mathscr{R}$ we have $\langle\!\langle [\![T]\!] \rangle\!\rangle \mathscr{R} T$ 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 Subject reduction for expressions

We just recall the key type preservation result of Wright and Felleisen (1994), whose proof only requires minor adaptations concerning the set of values in `FuSe`.

Lemma 2 (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 a known result (Wright and Felleisen, 1994, Lemma 4.3), where the values include a few more cases. □

B.2 Subject reduction for processes

The next proposition establishes key properties of the reduction 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* $u \in \mathsf{dom}(\Gamma) \setminus \mathsf{ep}(\ell)$.

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 invalidated endpoints allows them to be typed with any instance of a channel type.

Lemma 3 (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 only interesting case being that of $e = a^p \in \mathsf{ep}(\ell)$. □

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 of Wright and Felleisen (1994) 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 4 (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 5 (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 6 (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 7 (congruence preserves typing)**.** *If* $\Gamma \vdash P$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q$.

Next is 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 $\mathscr{A}(P)$ 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* $\mathscr{A}(P)$ *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 $Q = \langle \mathscr{E}[\texttt{()}] \rangle \mid \langle v\ w \rangle$ and $\ell = \tau$. From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{fork}\ v\ w] : \texttt{unit}$. From Lemma 4 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 5 we deduce $\Gamma \vdash \mathscr{E}[\texttt{()}] : \texttt{unit}$. From [T-APP] we deduce $\Gamma \vdash v\ w : \texttt{unit}$. 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 $Q = \texttt{new}\ a\ \texttt{in}\ \langle \mathscr{E}[\texttt{pair}\ a^+\ a^-] \rangle$ where a is fresh and $\ell = \tau$. From [T-THREAD] we deduce $\Gamma \vdash \mathscr{E}[\texttt{create ()}] : \texttt{unit}$. From Lemma 4 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 5 we deduce $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A.A \vdash \mathscr{E}[\texttt{pair}\ a^+\ a^-] : \texttt{unit}$. We conclude with one application of [T-THREAD] and one 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 $Q = \langle \mathscr{E}_\ell[\texttt{()}] \rangle \mid \langle \mathscr{E}'_\ell[\texttt{()}] \rangle$ 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 4 and $\mathsf{TypeOf}(\texttt{close})$ we deduce $\Gamma(a^p) = \Gamma(a^{\overline{p}}) = \texttt{<•,•>}$ and $\Gamma \vdash \texttt{close}\ a^p : \texttt{unit}$ and $\Gamma \vdash \texttt{close}\ a^{\overline{p}} : \texttt{unit}$. Therefore, $\Gamma = \Gamma', a^p : \texttt{<•,•>}, a^{\overline{p}} : \texttt{<•,•>}$ for some Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$. From Lemma 5 we deduce $\Gamma' \vdash \mathscr{E}_\ell[\texttt{()}] : \texttt{unit}$ and $\Gamma' \vdash \mathscr{E}'_\ell[\texttt{()}] : \texttt{unit}$. We conclude with two applications of [T-THREAD] and one 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 $Q = \langle \mathscr{E}_\ell[a^p] \rangle \mid \langle \mathscr{E}'_\ell[\texttt{pair}\ v\ a^{\overline{p}}] \rangle$ and $\ell = \mathtt{map}$. 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 4 and $\mathsf{TypeOf}(\texttt{send})$ and the hypothesis that Γ is balanced we deduce that there exists Γ'' such that $\Gamma = \Gamma'', a^p : !\,[t * s], a^{\overline{p}} : ?\,[t * s]$ and $\Gamma \vdash \texttt{send}\ a^p\ v : s^\perp$ and $\Gamma \vdash v : t$. From Lemma 4 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 5 we derive $\Gamma' \vdash \mathscr{E}_\ell[a^p] : \texttt{unit}$. From one application of [T-PAIR] and Lemma 5 we derive $\Gamma' \vdash \mathscr{E}'_\ell[\texttt{pair}\ v\ a^{\overline{p}}] : \texttt{unit}$. 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 $Q = \langle \mathscr{E}_\ell[a^p] \rangle \mid \langle \mathscr{E}'_\ell[\texttt{inl}\ a^{\overline{p}}] \rangle$ and $\ell = \mathtt{map}$. 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 4 and $\mathsf{TypeOf}(\texttt{left})$ and the hypothesis that Γ is balanced we deduce that there exists Γ'' such that $\Gamma = \Gamma'', a^p : !\,[t + s], a^{\overline{p}} : ?\,[t + s]$ and $\Gamma \vdash \texttt{left}\ a^p : t^\perp$. From Lemma 4 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 5 we derive $\Gamma' \vdash \mathscr{E}_\ell[a^p] : \texttt{unit}$. From $\mathsf{TypeOf}(\texttt{inl})$ using [T-CONST], [T-APP], and Lemma 5 we derive $\Gamma' \vdash \mathscr{E}'_\ell[\texttt{inl}\ a^{\overline{p}}] : \texttt{unit}$. We conclude with two applications of [T-THREAD] and one application of [T-PAR].

[R-RIGHT] Analogous to the previous case.

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

[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 3 we deduce that $\Gamma' \vdash R_\ell$. We conclude with an application of [T-PAR].

[R-NEW] Then $P = \texttt{new}\ a\ \texttt{in}\ P'$ and $P' \xrightarrow{\ell'} Q'$ and $Q = \texttt{new}\ a\ \texttt{in}\ Q'$ and $\ell = \ell' \setminus a$. From [T-NEW] we deduce $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \vdash P'$ for some t. Observe that $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A$ is balanced if so is Γ. We distinguish three subcases:

- If $\mathsf{ep}(\ell') \cap \{a^+, a^-\} = \emptyset$, then by induction hypothesis we deduce $\Gamma', a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \vdash Q'$ for some Γ' such that $\Gamma \xrightarrow{\ell'} \Gamma'$ and we conclude with an application of [T-NEW].
- If $\ell' = \mathtt{c}a$, then by induction hypothesis we deduce $\Gamma, a^* : \forall A\,.\,A \vdash Q'$. By Lemma 6, 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 $\ell' = \mathtt{map}$, then by induction hypothesis we deduce that there exists s such that $\Gamma, a^+ : t, a^- : t^\perp, a^* : \forall A\,.\,A \xrightarrow{\mathtt{map}} \Gamma, a^+ : s, a^- : s^\perp, a^* : \forall A\,.\,A$ 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-STRUCT] A simple induction using Lemma 7. □

B.3 Partial progress

This section contains the proof of partial progress (Theorem 3). The first auxiliary result provides a syntactic characterisation 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. To characterise these expressions, we extend the syntactic category K introduced previously (Section 2):

$$\bar{K} \;::=\; \texttt{fork}\; v \;\mid\; \texttt{create} \;\mid\; K$$

Lemma 8. *We say that Γ is* ground *when its domain does not contain variables. If Γ is ground and $\Gamma \vdash e : t$ and $e \not\longrightarrow$, then either e is a value or $e = \mathscr{E}[\bar{K}\; v]$ for some $\mathscr{E}$, $\bar{K}$, and v.*

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.

$\boxed{e = x}$ This case is impossible because Γ is ground.

$\boxed{e = \texttt{let}\; x = e_1\; \texttt{in}\; e_2}$ Then $\Gamma \vdash e_1 : s$ for some s and $e_1 \not\longrightarrow$. Also, e_1 cannot be a value for otherwise e would reduce. By induction hypothesis we deduce that $e_1 = \mathscr{E}'[\bar{K}\; v]$ for some $\mathscr{E}'$, $\bar{K}$, and v. We conclude by taking $\mathscr{E} \stackrel{\text{def}}{=} \texttt{let}\; x = \mathscr{E}'\; \texttt{in}\; e_2$.

$\boxed{e = e_1\; e_2}$ Then $\Gamma \vdash e_1 : s \to t$ and $\Gamma \vdash e_2 : s$ for some s and $e_i \not\longrightarrow$ for $i = 1, 2$. We further distinguish three subcases. If e_1 is not a value, then by induction hypothesis we deduce that $e_1 = \mathscr{E}'[\bar{K}\; v]$ for some $\mathscr{E}'$, $\bar{K}$, and v and we conclude by taking $\mathscr{E} = \mathscr{E}'\; e_2$. If e_1 is a value but e_2 is not, then by induction hypothesis we deduce that $e_2 = \mathscr{E}'[\bar{K}\; v]$ for some $\mathscr{E}'$, $\bar{K}$, and v and we conclude by taking $\mathscr{E} = e_1\; \mathscr{E}'$. If both e_1 and e_2 are values, we deduce that:

- e_1 cannot be an abstraction or $\texttt{fix}$, for otherwise e would reduce;
- e_1 cannot be an endpoint or $()$, $\texttt{inl}\; v$, $\texttt{inr}\; w$, $\texttt{pair}\; v\; w$ because it has an arrow type;
- e_1 cannot be $\texttt{fork}$, $\texttt{pair}$, $\texttt{inl}$, $\texttt{inr}$, $\texttt{cases}$, $\texttt{send}$, $\texttt{pair}\; v$, $\texttt{cases}\; v$, or else e would be a value;
- e_1 cannot be $\texttt{fst}$ or $\texttt{snd}$, for otherwise e_2 would be a pair and e would reduce;
- e_1 cannot be $\texttt{cases}\; v\; w$, or else v would be an injected value and e would reduce.

Then, e_1 must be one of $\texttt{fork}\; w$, $\texttt{create}$, $\texttt{close}$, $\texttt{send}\; w$, $\texttt{receive}$, $\texttt{left}$, $\texttt{right}$, $\texttt{branch}$. We conclude by taking $\mathscr{E} = [\,]$, $\bar{K} = e_1$, and $v = e_2$. □

Theorem 3 (partial progress)**.** *If $\emptyset \vdash P$ and $\mathscr{L}(P)$, 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

$$P \equiv \texttt{new}\; a_1\; \texttt{in}\; \cdots \texttt{new}\; a_n\; \texttt{in}\; \textstyle\prod_{i \in I} \langle e_i \rangle$$

where $\prod_{i \in I} \langle e_i \rangle$ is a non-empty parallel composition of processes.

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 $\texttt{unit}$. Hence, from Lemma 8 we deduce that for every $i \in I$ either e_i is a value or there exist $\mathscr{E}_i$, $\bar{K}_i$, and v_i such that $e_i = \mathscr{E}_i[\bar{K}_i\; v_i]$.

Since the only value of type `unit` is `()`, we can assume that none of the e_i is `()`, for such threads could be removed by structural congruence. From the hypothesis $P \not\xrightarrow{\tau}$ we also deduce that none of the $\bar{K}_i$ is `create` or `fork` v, for otherwise P would be able to reduce according to the rules in Table 1. In summary, we can derive

$$P \equiv \texttt{new}\ a_1\ \texttt{in}\ \cdots \texttt{new}\ a_n\ \texttt{in}\ Q \qquad \text{where} \qquad Q = \prod_{i\in I}\langle \mathscr{E}_i[K_i\ v_i]\rangle$$

From the hypothesis $\emptyset \vdash P$ we deduce that for all $i \in I$ there exist c_i and p_i such that $v_i = c_i^{p_i}$. From the hypothesis $P \not\xrightarrow{\tau}$ 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 $\mathscr{L}(P)$ we deduce that $c_i^{\overline{p}_i}$ must occur somewhere in P. We reason by cases on K_i, and discuss only one case, when $K_i =$ `send` v, the others being analogous. Then, there exist t and s such that $\Gamma \vdash c_i^{p_i} : \,!\,[t * s]$ and $\Gamma \vdash c_i^{\overline{p}_i} : \,?\,[t * s]$. Suppose that $c_i^{\overline{p}_i} = c_j^{p_j}$ for some $j \in I$. It cannot be the case that $K_j \in \{\texttt{send}\ w, \texttt{left}, \texttt{right}\}$ because these would require $c_j^{p_j}$ to have a type with output capability. It cannot be the case that $K_j =$ `receive`, for otherwise P would be able to reduce. Finally, it cannot be the case $K_j =$ `branch`, because the message type in the type of $c_j^{p_j}$ would have a topmost $+$ type constructor, whereas the message type in the type of $c_i^{p_i}$ has a topmost $*$ type constructor. 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 $c_i^{\overline{p}_i} \in \mathsf{fn}(\mathscr{E}_j) \cup \mathsf{fn}(K_j)$ for some $j \in I$. □

References

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3:95–230, 2016.

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.

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.

Viviana Bono and Luca Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012.

Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Proceedings of FORTE'16*, LNCS 9688, pages 74–95. Springer, 2016.

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.

Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of PPDP'12*, pages 139–150. ACM, 2012.

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.

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.

Jacques Garrigue. Programming with polymorphic variants. In *Proceedings of ACM SIGPLAN Workshop on ML*, 1998.

Simon Gay and Malcolm Hole. Subtyping for Session Types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

Kohei Honda. Types for dyadic interaction. In *Proceedings of CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.

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.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL'08*, pages 273–284. ACM, 2008.

Raymond Hu and Nobuko Yoshida. Hybrid Session Verification through Endpoint API Generation. In *Proceedings of FASE'16*, LNCS 9633, pages 401–418. Springer, 2016.

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.

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session Type Inference in Haskell. In *Proceedings of PLACES'10*, EPTCS 69, pages 74–91, 2010.

Jane Street Developers. `Core` library documentation, August 2016. Available at `https://ocaml.janestreet.com/ocaml-core/latest/doc/core/`.

Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, LNCS 2757, pages 439–453. Springer, 2002. Extended version available at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System*, 2014. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/`.

Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proceedings of PADL'04*, LNCS 3057, pages 56–70. Springer, 2004.

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/`.

Luca Padovani. A Simple Library Implementation of Binary Sessions. Technical report, 2015. Available at `https://hal.archives-ouvertes.fr/hal-01216310`.

Luca Padovani. Context-Free Session Type Inference. Technical report, 2016. Available at `https://hal.archives-ouvertes.fr/hal-01385258`.

Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of HASKELL'08*, pages 25–36. ACM, 2008.

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, Imperial College London, 2008. Available at `http://pubs.doc.ic.ac.uk/session-types-in-haskell/`.

Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In *Proceedings of ECOOP'16*, LIPIcs 56, pages 21:1–21:28. Schloss Dagstuhl, 2016.

Jesse A. Tov. *Practical Programming with Substructural Types*. PhD thesis, Northeastern University, 2012.

Jesse A. Tov and Riccardo Pucella. Stateful Contracts for Affine Types. In *Proceedings of ESOP'10*, LNCS 6012, pages 550–569. Springer, 2010.

Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of POPL'11*, pages 447–458. ACM, 2011.

Vasco T. Vasconcelos and Peter Thiemann. Context-free session types. In *Proceedings of ICFP'16*, pages 462–475. ACM, 2016.

Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3): 384–418, 2014.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.