Session-ocaml: a Session-based Library with Polarities and Lenses

Keigo Imai[a], Nobuko Yoshida[b], Shoji Yuen[c]

[a] *Gifu University, Japan*
[b] *Imperial College London, UK*
[c] *Nagoya University, Japan*

Abstract

We propose `session-ocaml`, a novel library for session-typed concurrent/distributed programming in OCaml. Our technique solely relies on parametric polymorphism, which can encode core session type structures with strong static guarantees. Our key ideas are: (1) *polarised session types*, which give an alternative formulation of duality enabling OCaml to automatically infer an appropriate session type in a session with a reasonable notational overhead; and (2) a *parameterised monad* with a data structure called '*slots*' manipulated with *lenses*, which can statically enforce session linearity including delegations. We introduce a notational extension to enhance the session linearity for integrating the session types into the functional programming style. We show applications of `session-ocaml` to a travel agency use case and an SMTP protocol implementation. Furthermore, we evaluate the performance of `session-ocaml` on a number of benchmarks.

Keywords: Session types, Functional programming, Behavioural types, Parametric polymorphism, Polarity, Lenses, OCaml

1. Introduction

Session types [2], from their origins in the π-calculus [3], serve as rigorous specifications for coordinating *link mobility* in the sense that a communication link can move among participants, ensuring type safety. In session type

☆An earlier version of this article was presented at COORDINATION 2017 [1]
Email address: `keigoi@gifu-u.ac.jp` (Keigo Imai)

systems, such link mobility is called *delegation*. Once the ownership of a session is delegated (transferred) to another participant, the session cannot be used anymore at the sender. This property is ensured by *linearity* of sessions and appears in all session type systems. Furthermore, most session type implementations with delegation [4, 5, 6] explicitly rely on linearity to ensure the correct usage of each channel.

Linearity of session channels, however, is a major obstacle to adopt the session type disciplines in mainstream programming languages, as it requires special syntax extensions for session communications [7], or it depends on specific language features, such as type-level functions in Haskell [5, 8, 9, 4], or affine types in Rust [10]. In [6, 11, 12, 13], the check for linearity falls back to run-time and dynamic checking. For instance, a common technique in the Haskell implementations is to track linear channels using extra *symbol tables* to bookkeep the types of resources conveyed by a *parameterised monad*. A Haskell type for a session-typed function is of the form:

$$t_1 \to \cdots \to \mathtt{M}\,\{c_1 \mapsto s_1, c_2 \mapsto s_2, \cdots\}\,\{c_1 \mapsto s'_1, c_2 \mapsto s'_2, \cdots\}\,\alpha$$

where `M` is a monad type constructor of arity three; α is the result type and the two mappings $\{\cdots\}$ are symbol tables before (and after) evaluation which assign each channel c_i to its session type s_i (and s'_i respectively). This symbol table is represented at the *type level*, hence the channel c_i is not a value, but a *type* which reflects an identity of the channel. Since this static encoding is Haskell-specific using type-level functions, it is not directly extensible to other languages.

This paper proposes the `session-ocaml` library, which provides a fully static implementation of session types in OCaml (i.e. sessions are checked at compile-time) without any extra mechanisms nor any tools to the OCaml compiler. We extend the technique posted to the OCaml mailing list by Garrigue [14] where the linear usage of resources is enforced solely by the parametric polymorphism mechanism. According to [14], the type of a *file handler* guarantees linear accesses to *multiple resources* using a symbol table in a monad-like structure. Adopting this technique to session types, in `session-ocaml`, *multiple simultaneous sessions* are statically encoded in a parameterised monad. More specifically, we extend the monad structure to the *slot monad* and the file handlers to *lenses*. The slot monad is based on a type `(`pre`,`$post$`,`a`)monad` (hereafter we use postfix type constructors of OCaml) where *pre* and *post* are called *slots* which act like a symbol table. Slots are represented as a sequence of types represented by nested pair types s_1 `*` `(`s_2 `*` `···)`. Lenses are proposed as combinators for bi-directional transformations [15, 16]. A lens consists of two functions: `get` that extracts a *view* from a given source, and `put` that *updates* the source for a given view. `session-ocaml` uses lenses to manipulate a symbol table of the slot monad. These mechanisms provide an *idiomatic way* (i.e. codes do not require interposing combinators to replace standard syntactic elements of functional languages) to declare session delegations and labelled session branching/selections with the static guarantee of type safety and linearity. FuSe [6], another implementation of session types on top of OCaml, combines static and dynamic checking for linearity (see § 6).

This paper is a revised version extended significantly from the previous conference paper [1].

- In § 2, we give an overview on the session types and the *polarised session types* in `session-ocaml`.
- In § 3.4, we show the technical details to implement `session-ocaml` modules.
- In § 3.5, we develop macros with the *slot pattern*. The macros provide useful idioms and extensions for session-based programming.
- In § 3.6, we present the implementation technique with communication APIs utilising *ad hoc polymorphism*.
- In § 4.2, we use the extensions in § 3.5 and § 3.6 for the SMTP example. We illustrate how they improve the description compared to the version shown in [1].
- In § 5, we show the performance evaluation through a number of benchmarks. The benchmarks confirm that `session-ocaml` runs comparable to FuSe with the advantage of static type checking. We also discuss a performance issue of the the monadic computations in our framework.

Outline. The rest of the paper is as follows. Section 2 introduces session types and outlines programming with `session-ocaml`. Section 3 shows the design and implementation of `session-ocaml` with the polarised session types and the slot monads. Section 4 presents two examples, a travel agency use case and an implementation of the SMTP protocol. Section 5 gives performance benchmarks of `session-ocaml` in comparison with FuSe. Section 6 discusses session type implementations in functional languages. Section 7 concludes and discusses further applications of our technique. For an additional example, Appendix A shows an example of a database server. `session-ocaml` is available at `https://github.com/keigoi/session-ocaml` and the benchmark programs are available at `https://github.com/keigoi/FuSe-clone/`.

2. Programming with `session-ocaml`

In this section, we overview session types and the session-typed programming with `session-ocaml`, and summarise communication primitives in the library.

2.1. Session types and polarised session types

For readers who are not familiar with session types, we provide an overview on session types as follows. The original syntax of session types [2] is defined by the following grammar:

$$\begin{aligned} S ::= {} & !V;S \mid ?V;S \mid \texttt{end} \mid \mu\alpha.S \mid \alpha \mid ![S'];S \mid ?[S'];S \\ & \mid \&\{l_1 : S_1, \ldots, l_n : S_n\} \mid \oplus\{l_1 : S_1, \ldots, l_n : S_n\} \end{aligned}$$

$!V;S$ and $?V;S$ are sending and receiving of a value type V with the continuation of S, respectively. `end` is the terminated session. $\mu\alpha.S$ is a recursive session which is equivalent to $S[\mu\alpha.S/\alpha]$ (the type obtained by replacing free occurrences of α in S with $\mu\alpha.S$ itself); $![S'];S$ and $?[S'];S$ are delegation of a session S' with the continuation of S. $\&\{l_1 : S_1, \ldots, l_n : S_n\}$ offers labels $l_1, \ldots, l_n$ to its counterpart and continues to one of $S_1, \ldots, S_n$ according to the selected label. Conversely, $\oplus\{l_1 : S_1, \ldots, l_n : S_n\}$ selects a label l_i and continues to S_i. The *duality* is a one-to-one relation on session types which ensures that two parties perform complementary actions. It is defined inductively as follows:

$$\begin{array}{lll} \overline{!V;S} = ?V;\overline{S} & \overline{?V;S} = !V;\overline{S} & \overline{\texttt{end}} = \texttt{end} \\ \overline{![S'];S} = ?[S'];\overline{S} & \overline{?[S'];S} = ![S'];\overline{S} & \overline{\mu\alpha.S} = \mu\alpha.\overline{S[\overline{\alpha}/\alpha]} \\ \overline{\&\{l_i : S_i\}} = \oplus\{l_i : \overline{S_i}\} & \overline{\oplus\{l_i : S_i\}} = \&\{l_i : \overline{S_i}\} & \end{array}$$

where $\overline{(\overline{\alpha})} = \alpha$.

One of the problems in implementing session types is how to include the duality in the type inference mechanism of the target language. To enable *session-type inference* solely by the built-in type unification in OCaml, `session-ocaml` relies on *polarised session types* in order to reduce duality checking to

Listing 1 The xor server and its client

```
1  let xor_ch = new_channel ();;
2  Thread.create
3    (accept_ xor_ch (fun () ->
4      recv s >>= fun (x,y) ->
5      send s (xor x y) >>
6      close s)) ();;
7  connect_ xor_ch (fun () ->
8    send s (false,true) >>
9    recv s >>= fun b ->
10   print_bool b;
11   close s) ()
```

unification. In a polarised session type p^q, the *polarity* q is either `serv` (server) or `cli` (client). When a session is being initiated, polarities are assigned to each end of the session according to the primitives, namely `cli` for the proactive peer and `serv` for the passive peer. p is the *protocol type* (defined later) as an *objective* view of a communication. The duality of polarised session is defined as follows:

$$\overline{p^{\texttt{cli}}} = p^{\texttt{serv}} \qquad \overline{p^{\texttt{serv}}} = p^{\texttt{cli}}$$

The type inference in polarised session types is driven solely by type unification which checks whether a protocol type matches its counterpart or not. A protocol type is defined by the following grammar using two *communication directions* of `req` (request; client to server) and `resp` (response; server to client).

$$\begin{aligned} p ::= & \ \texttt{req}[V];p \mid \texttt{resp}[V];p \mid \texttt{close} \mid \mu\alpha.p \mid \alpha \mid \texttt{req}[p^q];p' \mid \texttt{resp}[p^q];p' \\ & \mid \texttt{req}\{l_1 : p_1, \ldots, l_n : p_n\} \mid \texttt{resp}\{l_1 : p_1, \ldots, l_n : p_n\} \end{aligned}$$

$\texttt{req}[V];p$ is a transmission of a message type V from a client to a server and $\texttt{resp}[V];p$ is from a server to a client. $\mu\alpha.p$ is recursion and `close` is the terminated session. $\texttt{req}[p^q];p'$ and $\texttt{resp}[p^q];p'$ are delegation. Note that a delegated type is annotated with polarities, and in § 3.3 we discuss the impact of having polarities in types rather than having them in the *syntax*, as in [17, 18]. $\texttt{req}\{l_1 : p_1, \ldots, l_n : p_n\}$ is the selection on the client side and the branching on the server side. Similarly, $\texttt{resp}\{l_1 : p_1, \ldots, l_n : p_n\}$ is the branching on the client side and the selection on the server side.

2.2. Send and receive primitives

Listing 1 shows a server and a client which communicate boolean values. The variable `xor_ch` (line 1) is a *shared channel* (or *service* channel) used to start the communication between a client and a server. `Thread.create` f x in the OCaml standard library creates a thread which evaluates the expression

`(`f x`)`, and is used to start the server thread (lines 2-6). The function `accept_`[1] accepts a session from a client (line 3) at `xor_ch`, then runs the body of the session inside `(fun () -> ..)` (lines 4-6). The global variable `s` used in the session refers to a *session endpoint* (or *session channel*) which is connected to the other endpoint[2]. At the beginning of the session, the server thread receives (`recv`) a pair of booleans (line 4). For inferring session types in OCaml, the communication primitives are concatenated by the *bind* operators `>>=` and `>>` of a parameterised monad [19] which conveys session endpoints. Intuitively, an expression op_1 `>>= fun x ->` op_2 executes op_1 and op_2 in order, binding free occurrences of `x` in op_2 to the result of op_1. op_1 `>>` op_2 is a shorthand of op_1 `>>= fun _ ->` op_2 discarding the result of op_1. The thread calculates the exclusive-or of the received values, transmits (`send`) back the resulting boolean, and finishes the session (`close`) (lines 5-6). The client (lines 7-11) connects to the server at `xor_ch` using `connect_`[3] (line 7), sends a pair of boolean values `(false,true)` (line 8), receives the exclusive-or of them from the server (line 9), prints it on the screen and finishes the session (lines 10-11).

A *channel type* of the form α `channel` is assigned to shared channels (e.g. `xor_ch`) where α is a protocol type. Protocol types are the primary language of communication specification in `session-ocaml`. The protocol type of `xor_ch` is `req[bool*bool];resp[bool];close` in the syntax introduced in § 1. It indicates that the server receives a request of type `bool * bool` before it sends a response of type `bool` back to the client. The channel type at `xor_ch` is represented in OCaml as follows:

```
[`msg of req * (bool * bool) * [`msg of resp * bool * [`close]]] channel
```

Type ``[` ``tag `of` τ`]` and ``[` ``tag`]` are the *polymorphic variant* types in OCaml. We use polymorphic variant type ``[`msg of`` r `*` v `*` p`]` to represent a protocol type r`[`v`];`p. The first component r is `req` (or `resp`) which indicates a communication direction from a client to a server (or from a server to a client, respectively). The second component v is the type of a message and the last component p is a protocol type denoting the continuation. ``[`close]`` is the end of a session.

[1]The suffixed underscore means that it runs immediately instead of returning a monadic action (see later). The sub-expression `accept_ (fun () -> ..)` is a *partial application* and the rest is fully applied in the new thread by `Thread.create` with argument `()`.

[2]The variable `s` is a *slot specifier* which we will introduce later in this section.

[3]We use the keyword `connect` instead of `request` used in most literature to avoid confusion with `req` in protocol types.

Listing 2 A logical operation server

```
1  let log_ch = new_channel ()
2  type binop = And | Or | Xor | Imp
3  let eval_op = function
4    | And -> (&&)
5    | Or -> (||)
6    | Xor -> xor
7    | Imp -> (fun a b -> not a || b)
8  let rec logic_server () =
9    branch ~left:(s, fun () ->
10       recv s >>= fun op ->
11       recv s >>= fun (x,y) ->
12       send s (eval_op op x y) >>= fun () ->
13       logic_server ())
14     ~right:(s, fun () -> close s);;
15 Thread.create
16   (accept_ log_ch logic_server) ();;
17 connect_ log_ch (fun () ->
18   select_left s >>
19   send s And >>
20   send s (true, false) >>
21   recv s >>= fun ans ->
22   (print_bool ans;
23   select_right s >>
24   close s)) ()
```

The reason for using the polymorphic variant types rather than normal type constructors like `(`r`,` v`,` p`) msg` in protocol types is that it allows *equi-recursive types* to directly implement session type recursion in OCaml type.

2.3. Branching and recursion

A combination of branching and recursion provides various useful idioms. Listing 2 shows a logical operation server. The channel type inferred for `log_ch` (line 1) is:

```
([`branch of req *
  [`left of [`msg of req * binop * [`msg of req * (bool * bool) *
        [`msg of resp * bool * 'a]]]
  |`right of [`close]]] as 'a) channel
```

The type ``[`branch of`` r `* [`$\cdots$ ``|` ``lab_i `of` p_i `|` $\cdots$`]]` ($r \in \{$`req`, `resp`$\}$) represents a protocol type $r\{\cdots, lab_i : p_i, \cdots\}$ that continues to p_i when label lab_i is communicated with a communication direction r. t `as 'a` is an equi-recursive type [20] of OCaml that represents recursive structure of a session where `'a` in t is instantiated by t `as 'a`. Lines 2-7 define a type and a function for logical operations over boolean values. The function `logic_server` (lines 8-14) describes the body of the server. An expression `branch ~left :(s,` $cont_1$`) ~right:(s,` $cont_2$`)` offers a choice between labels `left` and `right` to the peer[4], and it continues to $cont_1$ when label `left` is selected, or to $cont_2$ when `right` is selected. The evaluation of $cont_1$ and $cont_2$ is deferred by `(fun () -> ..)`. Upon receipt of `left`, the server receives a request for a logical operation of type `binop` (line 10) and its operands of type `bool * bool`

[4]The reason two `s`'s are required will be explained in § 3.3.

(line 11), sends back a response (line 12), and calls `logic_server` recursively (line 13). The recursive call of `logic_server` is again deferred inside `(fun () -> ..)`. Receiving `right` terminates the session (line 14). Lines 15-16 start the server in a new thread.

The function `select_left` (and `select_right`) selects the label `left` (and `right`) from the offered choices, respectively. A client using selection is shown in lines 17-24: It selects the label `left` (line 18), then it requests conjunction (line 19), sends the two operands (line 20), receives the response and prints it (lines 21-22). Then, it selects `right` (line 23) and the session ends (line 24).

Syntax extensions for the generalised choice. Although the branching among more than two labels can be simulated by the nested use of binary branches, it is more pragmatic if we have a *generalised choice* for an arbitrary number of labels. `session-ocaml` provides a syntax extension `match%branch` as follows:

```
match%branch s with
| `lab_1 -> e_1
| ..
| `lab_n -> e_n
```

The expression above offers labels $lab_1, \ldots, lab_n$ and continues to e_i when lab_i is selected. The symbol `%` stands for an *extension point* of the OCaml syntax handled by the `session-ocaml` preprocessor. Similarly, the selection of lab_i is done by ``[%select s `lab_i]``. The Travel Agency usecase in § 4.1 demonstrates the use of the generalised choices.

2.4. Handling multiple sessions

We explain how *multiple sessions* are handled in `session-ocaml`, and exhibit an interesting communication pattern of link mobility, called *session delegation*. Session delegation dynamically changes the communication counterparts during a session. A typical pattern utilising delegations incorporates a main thread accepting a connection with worker threads doing the actual work to increase responsiveness of a service.

To handle multiple sessions, we explicitly assign each session endpoint to a distinct *slot* via *slot specifiers* `_0`, `_1`, ... In the xor server (Listing 1) and the logical operation server (Listing 2), a special identifier `s` has been used for referring a single session endpoint in that context as the *0-th slot*.

Listing 3 shows an example of a responsive server using delegation. The server accepts repeated connection requests on the channel `log_ch` defined in Listing 2, with six worker threads. The main thread (lines 2-7) accepts

Listing 3 A responsive server using delegation (`log_ch` is from Listing 2)

```
1  let worker_ch = new_channel ()
2  let rec main () =
3    accept log_ch ~bindto:_0 >>
4    connect worker_ch ~bindto:_1 >>
5    deleg_send _1 ~release:_0 >>
6    close _1 >>= fun () ->
7    main ();;
8  Thread.create (run main) ();;
9
10 let rec worker () =
11   accept worker_ch ~bindto:_1 >>
12   deleg_recv _1 ~bindto:_0 >>
13   close _1 >>
14   logic_server () >>= fun () ->
15   worker ();;
16 for i = 0 to 5 do
17   Thread.create (run worker) ()
18 done
```

a connection from a client (`accept`) with `log_ch` and assigns the established session to the *0-th* slot (`~bindto:_0`)[5]. Next, it connects (`connect`) to a worker waiting for a delegation at the channel `worker_ch` (line 1) and assigns the session to the *1st* slot (`~bindto:_1`). Finally, it delegates the m-th session session with the client to the worker over the n-th session by `deleg_send _`n `~release:_`m. Then the server ends the session for the worker and accepts the next connection. Line 8 starts the main thread. Here `run` is a function that executes a monadic action of `session-ocaml`.

A worker thread (lines 10-15) accepts a session from the main thread (line 11). Then it receives the delegated session and assigns the session to the 0-th slot (line 12), calls `logic_server` in Listing 2 (line 14) and recurses to itself (line 15). Note that the received session endpoint is implicitly passed to `logic_server` via the 0-th slot and consumed by it. Lines 16-18 start the six worker threads.

The type of `worker_ch` is inferred as follows:

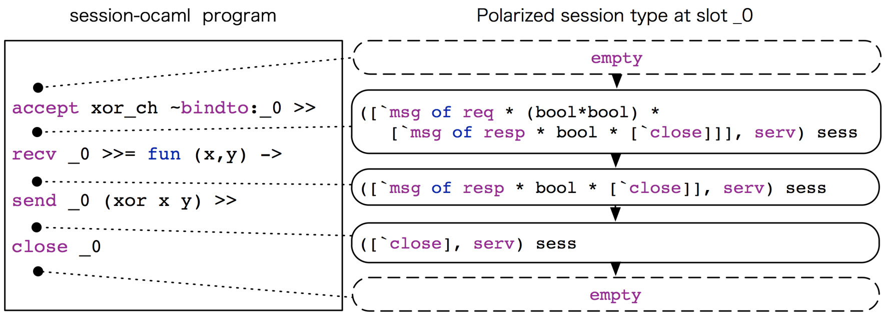
```
[`deleg of req * (logic_p, serv) sess * [`close]] channel
```

where ``[`deleg of`` r `*` `(`p`,`q`) sess *` p'`]` represents the protocol type for delegation $r[p^q];p'$ with a communication direction r. `(`p`,` q`) sess` is a polarised session type for the delegated session which consists of protocol type p and polarity q, and p' is a protocol type denoting the continuation. The type above denotes the delegation of a session with the direction `req` and it continues to the session ``[`close]``. `(logic_p, serv) sess` is a polarised session type for

[5]The slot number starts from zero and we say the 0-th slot for the first slot in a slot sequence. We use named arguments to indicate a slot to be allocated (`bindto`) or released (`release`). Names can be omitted; e.g. `deleg_send _0 _1` is equivalent to `deleg_send _0 ~release:_1`. Hence the use of named arguments does not affect safety/correctness in our encoding of session types.

Table 1: Correspondence between polarities and communication directions

	send	select	deleg_send	recv	branch	deleg_recv
cli	req	req	req	resp	resp	resp
serv	resp	resp	resp	req	req	req



```
accept xor_ch ~bindto:_0 >>
recv _0 >>= fun (x,y) ->
send _0 (xor x y) >>
close _0
```

```
([`msg of req * (bool*bool) *
   [`msg of resp * bool * [`close]]], serv) sess
([`msg of resp * bool * [`close]], serv) sess
([`close], serv) sess
```

Figure 1: Session type changes in `xor_server`

the delegated session as we explain the details next. `logic_p` is the protocol and `serv` is the polarity of the delegated session, respectively. Here we assume `logic_p` is the protocol type of `log_ch`. By inferring the protocol types, `session-ocaml` can statically guarantee the safety of higher-order protocols including delegations.

2.5. Tracking sessions with polarised session types

Communication safety is checked by matching the protocol types inferred at both ends. In the polarised session type `(`p`,` q`) sess`, p is a protocol type, and $q \in \{$`serv`,`cli`$\}$ is the polarity determined at session initiation. `serv` is assigned to the `accept` side and `cli` to the `connect` side.

The polarised session type gives a simple way to let the type checker infer a uniform protocol type according to a communication direction and a polarity assigned to the endpoint. For example, as we have seen, we deduce `resp` (response) from server transmission (`send`) and client reception (`recv`). Table 1 shows the correspondence between polarities and communication directions.

To track the entire session, a polarised session type changes in its protocol part as the session progresses. Figure 1 shows the changes of the session type in the 0-th slot of the xor server (here we use `_0` instead of `s`). The server first accepts a connection and assigns the session type to the 0-th slot, where the type before acceptance is `empty`. The reception of the pair of booleans is followed by the transmission of the xor of those booleans. This interaction consumes `req` and `resp`, and the slot becomes `empty` again at the end of the session. As for Listing 3, similar type changes occur on both `main` and `worker` and their types are inferred as follows:

```
unit -> (empty * (empty * 'ss), 'tt, 'a) session
```

where the type `(`*pre*`,` *post*`,` *a*`) session` specifies that it expects a slot sequence *pre* at the beginning, and returns another slot sequence *post* with a value of type a. The nested pair type `empty * (empty * 'ss)` denotes that the 0-th and 1st slots are empty at the beginning. Since function `main` and `worker` never return the answer (i.e. the recursion runs infinitely), the remaining types `'tt` and `'a` are left as variables.

Table 2 summarises the communication behaviour of the `session-ocaml` communication primitives, and shows the protocol types before and after execution. A *pre-type* is the type required before execution and *post-type* is the type guaranteed after execution. In the first column of "primitive", $\_n$ and $\_m$ are slot specifiers, e is an expression of a base type, and `` ` ``*lab* is a polymorphic variant. In the second and third columns of "pre-type" and "post-type", v is a base type, p is a protocol type, s is a polarised session type (of *any* polarity), t is either a polarised session type or `empty` and *ch* is a shared channel. The protocol types shown in the table are at polarity `cli` except for the rows of `accept` and `connect` which have a fixed polarity in the allocated session. The protocol type at `serv` is obtained by simultaneously replacing `req` with `resp` and `resp` with `req`. For example, the session `send` $\_n$ e has ``([`msg of req * ``v`` * ``p``], cli) sess`` as the pre-type at `cli` and ``([`msg of resp * ``v`` * ``p'``], serv) sess`` as the pre-type at `serv` where $\_n$ is a slot specifier, e is an expression, v is the type of e, and p and p' are protocol types (where p' is obtained from p by swapping `req` and `resp`).

The primitives for the session initialisation `accept_` *ch* *f* `()` and `connect_` *ch* *f* `()` in the previous listings are shorthands of "`run (fun () -> accept` *ch* `~bindto:_0 >>` *f* `()) ()`" and "`run (fun () -> connect` *ch* `~bindto:_0 >>` *f* `()) ()`", respectively. They establish a session at *ch* and assign it to the 0-th slot, then run the given monadic computation `(`*f* `())`.

The selection primitives `select_left`, `select_right` and `[%select` $\_n$ `` ` ``*lab*`]` have *open* polymorphic variant type `[>...]` as the pre-type to simulate the *subtyping* of the labelled branches. For example, in the session type theory,

Table 2: session-ocaml primitives and polarised session types at cli^{*1}

Primitive	Pre-type	Post-type	Synopsis
send _n e	([`msg of req * v * p], cli) sess	(p, cli) sess	Send $e : v$ at the n-th slot
recv _n	([`msg of resp * v * p], cli) sess	(p, cli) sess	Receive a value of type v at the n-th slot.
select_left _n	([`branch of req * [> `left of p]], cli) sess	(p, cli) sess	Select left at the n-th slot
select_right _n	([`branch of req * [> `right of p]], cli) sess	(p, cli) sess	Select right at the n-th slot
[%select _n `lab_i]	([`branch of req * [> `lab_i of p_i]], cli) sess	(p_i, cli) sess	Select lab_i at the n-th slot
branch ~left:(_n, f_1) ~right:(_n, f_2)	([`branch of resp * [`left of p_0 |`right of p_1]], cli) sess	t	branch offers left and right and match%branch offers $lab_1, \ldots, lab_n$ at the n-th slot provided that each pre-type of f_i () (or e_i) is (p_i, cli) sess and each post-type of f_i () (or e_i) is t.
match%branch _n with |`$lab_0$ -> $e_0$ | ⋯ |`lab_m -> e_m	([`branch of resp * [`lab_0 of p_0 | ⋯ |`lab_m of p_m]], cli) sess	t	
deleg_send _n ~release:_m	n:([`deleg of req * s * p], cli) sess m:s^{*2}	n:(p, cli) sess m:empty^{*2}	Delegate session at the m-th slot along the session at the n-th slot
deleg_recv _n ~bindto:_m	n:([`deleg of resp * s * p], cli) sess m:empty^{*2}	n:(p, cli) sess m:s^{*2}	Receive delegation along the n-th slot and assign it to the m-th slot
close _n	([`close], cli) sess	empty	Close session at the n-th slot
accept ch ~bindto:_n	empty	(p, serv) sess	Accept a connection at channel ch of protocol type p; assign a new session of polarity serv to n
connect ch ~bindto:_n	empty	(p, cli) sess	Connect to channel ch of protocol type p; assign a new session of polarity cli to n

∗1: accept and connect have the fixed polarity serv or cli in the post-type. ∗2: deleg_send and deleg_recv involve two slots; hence each column contains types for the two slots.

$\oplus\{$left $: S;$ right $: S'\}$ (a selection of either left or right) is a subtype of $\oplus\{$left $: S\}$ (selection of left). This is simulated in session-ocaml that the protocol type [`branch of [`left of p | `right of $p'$]] is subsumed by [`branch of [>`left of p]] according to the OCaml type system.

3. Design and implementation of session-ocaml

In this section, we present the technical details of session-ocaml. We first show the design of the polarised session types associated with the communication primitives (§ 3.1); then introduce the *slot monad* which conveys multiple session endpoints in a sequence of slots (§ 3.2). In § 3.3, we introduce the *slot specifier* to look up a particular slot in a slot sequence with *lenses* as a polymorphic data manipulation technique known in functional programming languages. We present the syntax extension for branching and selection, and explain a restriction on the polarised session types[6]. In § 3.4, we show the implementation details. In § 3.5, we introduce a macro for *slot-based linearity*, and shows Gay-Vasconcelos style session programming based on the macro, which unifies various aspects of `session-ocaml` including generalised choice. In § 3.6, we introduce the distributed implementation using ad hoc polymorphism.

3.1. Polarity polymorphism

In the framework of polarised session types, the type for sending would be either ``[`msg of req * 'v * 'p]`` or ``[`msg of resp * 'v * 'p]``, and one of them must be chosen according to the polarity from which the message is sent.

The ability of a session primitive to be used at both polarities is called *polarity polymorphism*. In order to relate the polarities to the directions, we define the types `cli` and `serv` as type aliases of communication direction pairs in the form *sending* $*$ *receiving*, as follows:

```
type cli = req * resp     type serv = resp * req
```

In the type signature of a session primitive, we introduce fresh type variables `'r1` for sending and `'r2` for receiving, and put `'r1 * 'r2` in the polarity. We further put `'r1` (`'r2`) in the communication direction if modality is sending (receiving, respectively), so that the direction is consistent with both polarity and modality described in Table 1. As a result, if the polarity of a session is `cli`, `'r1` and `'r2` are unified with `req` and `resp` respectively; while when the polarity is `serv`, `'r1` and `'r2` are unified with `resp` and `req` respectively. For example, the pre-type of `send` is ``([`msg of 'r1*'v*'p],'r1*'r2) sess`` because sending is `req` at `cli` while it is `resp` at `serv`, and that of `recv` is ``([`msg of 'r2*'v*'p],'r1*'r2) sess`` because receiving is `resp` at `cli` while it is `req` at `serv`. The same discipline applies to branching and delegation primitives.

3.2. The slot monad carrying multiple sessions

The key factor to achieve the session channel linearity is to keep session endpoints securely inside a monad. In `session-ocaml`, multiple sessions are

[6]The Travel Agency usecase in § 4.1 depends only on the materials from § 3.1 to § 3.3; thus readers can skip the subsequent sub-sections.

Listing 4 The slot monad

```
1 type ('pre,'post,'a) session
2 val return : 'a -> ('pre,'pre,'a) session
3 val (>>=) : ('pre,'mid,'a) session -> ('a -> ('mid,'post,'b) session)
4     -> ('pre,'post,'b) session
5 val (>>) : ('pre,'mid,'a) session -> ('mid,'post,'b) session -> ('pre,'post,'b) session
6 type empty and all_empty = empty * 'a as 'a
7 val run : ('a -> (all_empty,all_empty,'b) session) -> 'a -> 'b
```

conveyed in slots using the *slot monad* of type

$$(s_0 * (s_1 * \cdots),\ t_0 * (t_1 * \cdots),\ \alpha)\ \texttt{session}$$

where α is the type of the result, s_i's are pre-types and t_j's are post-types of the computation. Given a slot monad, we refer to the slots before and after the computation as *pre-slots* and *post-slots*, respectively. The type signature of the slot monad is shown in Listing 4. The operators `>>=` and `>>` (lines 3-5) compose the computation sequentially while enforcing consistent use of the concatenated sessions in the slots by sharing `'mid` appearing in the both post-slots on the left-hand side and the pre-slots on the right-hand side. It realises type changes in a session via *unification*. For example, in `send s And >> send s (true, false)` (from Listing 2) the left-hand side (`send s Add`) has the following type:

```
(([`msg of req*binop*'p1],cli) sess * 'ss1, ('p1,cli) sess * 'ss1, unit)
  session
```

The type of the right-hand side (`send s (true, false)`) is:

```
(([`msg of req*(bool*bool)*'p2],cli) sess*'ss2, ('p2,cli) sess*'ss2, unit)
  session
```

By unifying the post-type in the preceding computation with the pre-type in the following computation (and the rest of slots `'ss1` with `'ss2`), `>>=` produces the protocol type in the pre- and post-slots, and value type, as follows:

```
(([`msg of req*binop*[`msg of req*(bool*bool)*'p2]],cli) sess*'ss2,
    ('p2,cli) sess*'ss2, unit) session
```

In line 7, `run` takes a function returning a slot monad which requires all slots being `empty` before and after the execution, thus it precludes the use of unallocated slots and mandates that all sessions are finally closed. The type `all_empty` (line 6) is a type alias for OCaml equi-recursive type `empty * 'a as 'a`[7], enabling use of *arbitrarily* many slots.

[7]In order to have such a type, we compile the code with the `-rectypes` option. If we

Table 3: Types for slot specifiers

Specifier	Type
`_0`	`('a, 'b, 'a * 'ss, 'b * 'ss) slot`
`_1`	`('a, 'b, 's0 * ('a * 'ss), 's0 * ('b * 'ss)) slot`
`_2`	`('a, 'b, 's0 * ('s1 * ('a * 'ss)), 's0 * ('s1 * ('b * 'ss))) slot`
`_`n	`('a, 'b, 's0*(`$\cdots$`*('s`$_{n-1}$`*('a*'ss))`$\cdots$`), 's0*(`$\cdots$`*('s`$_{n-1}$`*('b*'ss))`$\cdots$`)) slot`

3.3. Lenses focusing on linear channels

In order to provide accesses to session endpoints conveyed inside a slot monad computation, we apply *lenses* [15, 16] to slot specifiers `_0`,`_1`, $\cdots$ to manipulate polymorphic data structures. The following shows the type of a slot specifier which modifies the n-th slot in a slot sequence (recall that the slot number starts from zero):

```
('a, 'b, 's_0 *(···('s_{n-1} *('a *'ss))···), 's_0 *(···('s_{n-1} *('b *'ss))···)) slot
```

This type says that `'a` of the n-th slot in the slot sequence `'s`$_0$`*(`$\cdots$`('s`$_{n-1}$`*('a*'ss))`$\cdots$`)` is replaced with `'b` by the lens. The resulting slot sequence type becomes `'s`$_0$`*(`$\cdots$`('s`$_{n-1}$`*('b*'ss))`$\cdots$`)`. The type of each slot specifier (`_0`, `_1`, $\cdots$) is shown in Table 3. We only provide a small number of slot specifiers (currently four) as default in `session-ocaml`. However, this is sufficient since in reality the number of sessions in a thread is relatively small in many usecases.

We present how to embed slot type changes into a pair of slot sequences in a slot monad where the position of the slot is specified by a slot specifier. The following is the type of `close` which takes a slot specifier for the session that will be terminated:

```
val close : (([`close],'r1*'r2) sess, empty, 'pre, 'post) slot
  -> ('pre, 'post, unit) session
```

In this type, the first and second type arguments of type `slot` prescribe *how* the slot type changes. The third and fourth arguments do not specify a slot in the slot sequence conveyed by the slot monad. For example, the type of `close _1` is given by the following type substitution:

(change of the slot type specified by `close`)
  `'a` $\mapsto$ `` ([`close],'r1*'r2) sess ``, `'b` $\mapsto$ `empty`,
(change of the slot sequence type specified by `_1`)
  `'pre` $\mapsto$ `` 's0 * (([`close],'r1*'r2) sess * 'ss) ``, `'post` $\mapsto$ `'s0 * (empty * 'ss)`

choose types for slots using objects or polymorphic variants, there is no need to use this option.

Listing 5 Signatures for communication primitives in `session-ocaml`

```
 1 val accept : 'p channel -> bindto:(empty, ('p, serv) sess, 'pre, 'post) slot
 2     -> ('pre, 'post, unit) session
 3 val connect : 'p channel -> bindto:(empty, ('p, cli) sess, 'pre, 'post) slot
 4     -> ('pre, 'post, unit) session
 5 val close : (([`close],'r1*'r2) sess, empty, 'pre, 'post) slot
 6     -> ('pre, 'post, unit) session
 7 val send : (([`msg of 'r1*'v*'p],'r1*'r2) sess, ('p,'r1*'r2) sess, 'pre, 'post) slot -> 'v
 8     -> ('pre, 'post, unit) session
 9 val recv : (([`msg of 'r2*'v*'p],'r1*'r2) sess, ('p,'r1*'r2) sess, 'pre, 'post) slot
10     -> ('pre, 'post, 'v) session
11 val deleg_send :
12     (([`deleg of 'r1*('pp,'qq) sess*'p],'r1*'r2) sess, ('p,'r1*'r2) sess, 'pre, 'mid) slot
13     -> release:(('pp,'qq) sess, empty, 'mid, 'post) slot -> ('pre, 'post, unit) session
14 val deleg_recv :
15     (([`deleg of 'r2*('pp,'qq) sess*'p],'r1*'r2) sess, ('p,'r1*'r2) sess, 'pre, 'mid) slot
16     -> bindto:(empty, ('pp,'qq) sess, 'mid, 'post) slot -> ('pre, 'post, unit) session
17 val select_left :
18     (([`branch of 'r1 * [>`left of 'p1]],'r1*'r2) sess, ('p1,'r1*'r2) sess, 'pre, 'post) slot
19     -> ('pre, 'post, unit) session
20 val select_right :
21     (([`branch of 'r1 * [>`right of 'p2]],'r1*'r2) sess, ('p2,'r1*'r2) sess, 'pre, 'post) slot
22     -> ('pre, 'post, unit) session
23 val branch :
24        left: (([`branch of 'r2 * [`left of 'p1 | `right of 'p2]],'r1*'r2) sess,
25                  ('p1,'r1*'r2) sess, 'pre, 'mid1) slot
26               * (unit -> ('mid1, 'post, 'a) session)
27     -> right:(([`branch of 'r2 * [`left of 'p1 | `right of 'p2]],'r1*'r2) sess,
28                  ('p2,'r1*'r2) sess, 'pre, 'mid2) slot
29               * (unit -> ('mid2, 'post, 'a) session)
30     -> ('pre, 'post, 'a) session
```

And the type completing the session in the first slot is:

```
close _1: ('s0 * ([`close],'r1*'r2) sess * 'ss), 's0 * (empty * 'ss), unit)
    session
```

Listing 5 exhibits the type signatures of the communication primitives by using lenses, the polarity polymorphism (§ 3.1), the slot monad (§ 3.2), and pre- and post-types in Table 2 (§ 2.5). Note that `release:` in the type of `deleg_send` and `bindto:` in the types of `accept`, `connect`, and `deleg_recv` are the named parameters of each primitive.

The session-establishment primitives `accept` and `connect` (lines 1-4) assign new session channels to the `empty` slot, whereas `close` (lines 5-6) finishes the session and leaves the slot `empty`. `send` and `recv` (lines 7-10) remove the prefix `` [`msg of `` $r * v * p$`]` by communicating a value to proceed a session to p.

The primitives for delegations `deleg_send` and `deleg_recv` (lines 11-16) update a pair of slots; one is for the transmission/reception of the delegating session and the other is for the delegated session. To update the slots at a time, an intermediate slot sequence `'mid` is shared in the pair of slot specifiers. `deleg_send` releases the ownership of the delegated session by replacing the slot type with `empty`, while `deleg_recv` allocates the slot for the acquired session.

The primitives for binary selection `select_left` and `select_right` communicate `left` and `right` labels with branching `branch` (lines 17-30), respectively. The type signature of `select_left` (or `select_right`) says that they remove the prefix ``[`branch of [>`left of`` p`]]` (or ``[`branch of [>`right of`` p`]]`) to proceed a session to p, where the symbol `>` of the open polymorphic variant (§ 2.5) denotes that the counterpart offers other labels. The function `branch` takes two pairs, each consists of a slot specifier and a continuation function either for `left` or for `right`. It receives a branch label first using one of the two slot specifiers[8]. Then, according to the received label, `branch` assigns the continuation session to the slot, and invokes the continuation function. We need distinct slot specifiers for different continuations because the pre-types in the two continuations can be different from each other, and a slot specifier assigning to the slot cannot have two different types in the third parameter.

Generalised choice. Since the OCaml type system does not allow to parameterise type labels, we provide macros for generalised choice. Listing 6 provides the helper functions for the macros. The type signatures in this listing are not enough to ensure that the type checking is sound. The macro ``[%select _``n`` ` ``lab_i`]` is expanded to `_select _`n` (fun x -> `` ` ``lab_i`(x))`, where the helper function `_select` transmits label lab_i on the slot n. The expanded code adopts FuSe's style (§ 5.3 of [6]) which uses the η-expansion of a variant tag `(fun p -> `` ` ``lab_i`(p))`[9]. The macro "`match%branch _`n` with | `` ` ``lab_1` -> `e_1` |` $\cdots$ `|`` ` ``lab_k` -> `e_k" manually extracts the polarised session type at slot n and explicitly assigns it to a slot in the expanded code, as follows.

```
_recvlabel _n >>= ((function |`lab₁(p1),q -> _set_sess _n (p1,q) >> e₁ | ···
                              |`labₖ(pk),q -> _set_sess _n (pk,q) >> eₖ)
                              : [`lab₁ of _|···|`labₖ of _]*_ ->_)
```

[8]It does not matter whether `branch` uses the left or right slot specifier for reception since they point to the same slot (i.e. the two slot specifiers has the same pre-type `'pre`).

[9]Note that in the argument type `'p -> ([>] as 'br)` of `_select`, the type `([>] as 'br)` says that `'br` can be an arbitrary polymorphic variant type and it does not force `'p` being the payload type of `'br`, and the macro takes care of this.

Listing 6 The helper functions for a generalised choice

```
val _select : (([`branch of 'r1 * 'br],'r1*'r2) sess, ('p,'r1*'r2) sess, 'pre, 'post) slot
    -> ('p -> ([>] as 'br)) -> ('pre, 'post, unit) session
val _recvlabel : (([`branch of 'r2 * 'br], 'r1*'r2) sess, empty, 'pre, 'post) slot
    -> ('pre, 'post, 'br * ('r1*'r2)) session
val _set_sess : (empty, ('p,'r1*'r2) sess, 'pre, 'post) slot
    -> 'p * ('r1*'r2) -> ('pre, 'post, unit) session
```

The helper functions `_recvlabel` and `_set_sess` have the types shown in Listing 6. The function `_recvlabel` $\_n$ first receives lab_i ($i \in \{1 \ldots k\}$), and then returns a pair of value `` ` ``lab_i`(p`i`)` of type `` [` ``lab_1 `of` p_i `|` $\cdots$ `|` `` ` ``lab_k `of` p_k`]` (which is unified with `'br`) and a value `q` of type `'r1*'r2`. Here `p`i has type p_i and p_i is the protocol type for the continuation of `` ` ``lab_i. `q` is a *witness* value of the polarity `'r1*'r2`. At the same time, the n-th slot is made `empty`. After that, the anonymous function matches on the pair `` (` ``lab_i`(p`i`),q)`, then by `_set_sess` $\_n$ `(p`i`,q)` it assigns the session of type `(`p_i`,'r1*'r2) sess` to the n-th slot and it continues to e_i. As for `[%select ..]`, the expanded code above plays a crucial role to enforce the type `'br` of `_recvlabel` to be a polymorphic variant type of branching labels.

The type annotation `` [` ``lab_i `of _|`$\cdots$`|`` ` ``lab_k `of _]*_ ->_` in the expanded code erases the row type variable `[<`$\cdots$`]` generated by the anonymous function. The annotation is necessary because the row type variable turns into a useless *monomorphic* row type variable `_[<`$\cdots$`]` in the inferred protocol type. This may cause a problem since the compiler requires monomorphic type variables not to escape from the compilation units.

The function `branch` is a specialised form of the macro `match%branch`, and an analogous function can be defined in the same way as the expanded code of the macro above, as follows (the type annotation is omitted).

```
let branch' _n0 ~left:(_n1,f1) ~right:(_n2,f2) =
  _recvlabel _n0 >>= (function | `left(p1),q -> _set_sess _n1 (p1,q) >> f1 ()
                               | `right(p2),q -> _set_sess _n2 (p1,q) >> f2 ())
```

It takes an extra slot specifier `_n0` which is used to receive a label by using `_recvlabel`. This is mandatory since the type of the slot specifier passed to `_recvlabel` cannot be the same as the ones passed to `_set_sess`. The reason that `branch` does not need this extra slot specifier is that it internally uses the unused *getter* component of the left slot specifier to receive a label. The implementation of `branch` will appear in § 3.4.

A note on delegation. The type signature of `deleg_send` allows to use the same slot in both arguments like `deleg_send _0 ~release:_0`. The expression `deleg_send _0 ~release:_0` delegates the continuation of the delegating session itself, and its type is inferred as follows (at polarity `cli`):

```
deleg_send _0 ~release:_0
: ([`deleg of req * ('pp,cli) sess * 'pp], cli) sess * '_ss,
   empty * '_ss, unit) session
```

where the type of the slot sequence is inferred as follows:

```
'pre  ↦ ([`deleg of req * ('pp,cli) sess * 'pp], cli) sess * 'ss
'mid  ↦ ('pp, cli) sess * 'ss
'post ↦ empty * 'ss
```

On the other hand, `deleg_recv _0 ~bindto:_0` is not typeable since by `deleg_recv _0` we would have

```
'pre  ↦ ([`deleg of req * ('pp,serv) sess * 'pp], serv) sess * 'ss
'mid  ↦ ('pp, serv) sess * 'ss
```

Because the 0-th type of `'mid` is not empty, we cannot assign the delegated session to that slot.

The delegation ``[`deleg of`` $r * s * p$`]` distinguishes the polarities in the delegated session s. This results in a situation that two sessions exhibiting the same communicating behaviour cannot be delegated at a single point in a protocol, if they have different polarities from each other. This is illustrated by the following untypeable example.

```
if b then connect ch1 ~bindto:_1 >> deleg_send _0 ~release:_1
else      accept ch2 ~bindto:_1  >> deleg_send _0 ~release:_1
```

Recall that `connect` yields an endpoint of polarity `cli` while `accept` gives polarity `serv`. Due to the different polarities in the delegated session types, the types of `then` and `else` clause conflict, even if they have identical behaviour. In [18], since a polarity is not a type but a syntactic construct, such a restriction does not exist. A similar restriction to ours exists in GV [21] where polarity in end exists as $\mathsf{end}_!$ and $\mathsf{end}_?$.

In principle, it is possible to automatically assign numbers to slot specifiers *locally* in a function instead of writing them explicitly. However, since the sequential composition of the `session` monad requires each post- and pre-type to be unified with each other, the *global* assignment of slot specifiers would require a considerable amount of work.

Listing 7 Basic operations on slot monad

```
 1 (* The slot monad *)
 2 type ('pre,'post,'a) session = 'pre -> 'post * 'a
 3 let return a = fun pre -> pre, a
 4 let (>>=) m f = fun pre -> let mid, a = m pre in f a mid
 5 (* Empty slots and the run function *)
 6 type empty = Empty
 7 type all_empty = empty * all_empty
 8 let rec all_empty = Empty, all_empty
 9 let run f x = snd (f x all_empty)
10 (* Slot specifiers *)
11 type ('a,'b,'pre,'post) slot = ('pre -> 'a) * ('pre -> 'b -> 'post)
12 let _0 = (fun (a,_) -> a), (fun (_,ss) b -> (b,ss))
13 let _1 = (fun (_,(a,_)) -> a), (fun (s0,(_,ss)) b -> (s0,(b,ss)))
14 let _2 = (fun (_,(_,(a,_))) -> a), (fun (s0,(s1,(_,ss))) b -> (s0,(s1,(b,ss))))
```

3.4. Implementing the slot monad and the communication primitives

3.4.1. Implementing the slot monad

Listing 7 shows an implementation of the basic operations for the slot monad. The slot monad is defined as a varying-type state monad `'pre -> 'post * 'a` (line 2) where `'pre` and `'post` are types of the initial and final states, respectively, and `'a` is the result type of a monadic action. `return` (line 3) is an operation that takes a value and returns a "pure" action without any effects on the state. `>>=` (line 4) returns a composed action: `>>=` takes a state `pre`, runs the action `m`, applies the result value `a` and the intermediate state `mid` to the continuation `f`. `run` (line 9) takes the empty slots of type `all_empty` (line 8) as the initial state to run the whole sessions.

The type of slot specifiers (line 11) is a pair of a getter and a setter for accessing on a sequence of slots. Slot specifiers `_0`, `_1`, $\cdots$ (lines 12-14) are defined as getters and setters for specific positions.

3.4.2. Implementing the communication primitives

Since session types allow channels to send and receive messages with heterogeneous types, they cannot be implemented directly by OCaml types. A solution is to use untyped channels as an underlying communication medium as in FuSe [6], which we could adopt to achieve better run-time performance. It is straightforward to extend the technique to slots.

Instead, we build session types based on the encoding of the linear types by Kobayashi et al [22]. This implementation is inherently safe since it does not use dangerous operations on untyped channels. A shortcoming is that the encoding incurs some overhead to generate a channel each time when a

Listing 8 An implementation of polarised session types

```
1  (* Communication direction and polarity *)
2  type req = Req and resp = Resp
3  type cli = resp * req and serv = req * resp
4  (* The polarised session type and the message wrapper *)
5  type ('p, 'q) sess = 'p wrap Channel.t * 'q
6  and 'p wrap =
7    Msg : ('v * 'p wrap Channel.t) -> [`msg of 'r * 'v * 'p] wrap
8  | BranchL : 'p1 wrap Channel.t -> [`branch of 'r * [> `left of 'p1]] wrap
9  | BranchR : 'p2 wrap Channel.t -> [`branch of 'r * [> `right of 'p2]] wrap
10 | Chan : (('pp, 'qq) sess * 'p wrap Channel.t) -> [`deleg of 'r * ('pp, 'qq) sess * 'p] wrap
11 (* The service channel *)
12 type 'p channel = 'p wrap Channel.t Channel.t
```

Listing 9 Buffered channel module

```
module Channel : sig
  type 'a t
  val create : unit -> 'a t
  val send : 'a t -> 'a -> unit
  val receive : 'a t -> 'a
end
```

message is sent or received.

Listing 8 shows the definitions of service channels (line 12) and session channels (line 5) where the signature of underlying `Channel` module is given in Listing 9. Hereafter channels from the `Channel` module are called *plain channels*. A session channel `('p, 'q) sess` (line 5) is a pair of a plain channel `'p wrap Channel.t` and a polarity `'q`, where `wrap` (line 6) is the type of the *wrapper* of the message payload (if any) and a plain channel for the subsequent communications. The wrapper uses generalised algebraic data types (GADTs) [23] to associate session messages with the three kinds of protocol types, ``[`msg ..]``, ``[`branch ..]`` and ``[`deleg .]``. The constructor `Msg` (lines 7) is communicated by `send` and `recv`, and has the protocol type of ``[`msg ..]`` and the payload is the OCaml's value type (`'v`). `BranchL` and `BranchR` (lines 8-9) is for branching (`branch`, `select_left` and `select_right`) and the protocol type is ``[`branch of [`left .. | `right ..]]``. The wrapper labels have no payload since each constructor represents a branching label itself. `Chan` (line 10) is for delegation (`deleg_send` and `deleg_recv`) and the protocol type is ``[`deleg ..]``. It has a delegated session channel `('pp, 'qq) sess` as a payload.

Listings 10 and 11 show the implementation of the communication primitives. `new_channel` (line 1 in Listing 10) creates a new shared channel. `connect` (lines 2-8) and `accept` (lines 9-11) create a new session channel and share

Listing 10 Implementation of communication in the slot monad (1)

```
 1 let new_channel = Channel.create
 2 let connect ch ~bindto:(_,set) = fun pre ->
 3   (* Generates a session channel *)
 4   let ch' = Channel.create () in
 5   (* Establish a connection and share the session channel with the server *)
 6   Channel.send ch ch';
 7   (* Then put it in a slot and return *)
 8   set pre (ch',(Resp,Req)), ()
 9 let accept ch ~bindto:(_,set) = fun pre ->
10   let ch' = Channel.receive ch in
11   set pre (ch',(Req,Resp)), ()
12 let send (get,set) v = fun pre ->
13   (* Extract the session channel (q is polarity) *)
14   let ch,q = get pre
15   (* Generate a session channel for subsequent communication *)
16   and ch' = Channel.create () in
17   (* wrap the message and the above channel in Msg and transmit it *)
18   Channel.send ch (Msg(v,ch'));
19   (* Then put it in a slot and return  *)
20   set pre (ch',q), ()
21 let recv (get,set) = fun pre ->
22   let ch,q = get pre in
23   let Msg(v,ch') = Channel.receive ch in
24   set pre (ch',q), v
```

the session over the service channel. `send` (lines 12-20), `recv` (lines 21-24), `select_left` (lines 1-5 in Listing 11) and `select_right` (lines 6-10) take a session channel from the slot using the lenses; and communicate using the wrapper, storing the new session channel in the slot. `branch` (lines 11-15) matches on the received message and branches to the appropriate continuation. `deleg_send` (lines 16-21) transmits the session channel stored in the second slot and removes it from the slot. `deleg_recv` (lines 22-26) receives the delegated channel and stores it in the second slot. Finally, `close` just throws away the session channel and fill in the slot with `Empty`.

3.5. Syntactic extension for functional programming

In this section, we integrate the syntax of `session-ocaml` with the functional programming style in an *idiomatic* way. A slot assignment is of the form *expr* `~bindto:_`n where n is a slot number of a new session. We develop the *slot pattern*[10] `#`s which assigns a session to the slot indicated by a slot specifier s.

[10]Here we abuse and replace OCaml's *type pattern* `#typ` for slot patterns.

Listing 11 Implementation of communication in the slot monad (2)

```
let select_left (get,set) = fun pre ->
  let ch,q = get pre
  and ch' = Channel.create () in
  Channel.send ch (BranchL(ch'));
  set pre (ch',q), ()
let select_right (get,set) = fun pre ->
  let ch,q = get pre
  and ch' = Channel.create () in
  Channel.send ch (BranchR(ch'));
  set pre (ch',q), ()
let branch ~left:((get1,set1),f1) ~right:((get2,set2),f2) = fun pre ->
  let (ch1,q) = get1 pre in
  match Channel.receive ch1 with
  | BranchL(ch1') -> f1 () (set1 pre (ch1',q))
  | BranchR(ch2') -> f2 () (set2 pre (ch2',q))
let deleg_send (get0,set0) ~release:(get1,set1) = fun pre ->
  let ch0,q1 = get0 pre and ch0' = Channel.create () in
  let mid = set0 pre (ch0',q1) in
  let ch1,q2 = get1 mid in
  Channel.send ch0 (Chan((ch1,q2),ch0'));
  set1 mid Empty, ()
let deleg_recv (get0,set0) ~bindto:(get1,set1) = fun pre ->
  let ch0,q0 = get0 pre in
  let Chan((ch1',q1),ch0') = Channel.receive ch0 in
  let mid = set0 pre (ch0',q0) in
  set1 mid (ch1',q1), ()
let close (get,set) = fun pre ->
  set pre Empty, ()
```

Based on these constructs, we propose a new idiomatic style, which is reminiscent of Gay-Vasconcelos [24] where session types are reformulated using *linear types*. With the new syntax, we can write `let%lin` #*s* = `accept` *ch* in which the intention of assigning a session to slot *s* is much clearer than merely writing `accept` *ch* `~bindto:`*s*. In § 4.2, we illustrate how this extension is integrated into the functional programming style.

3.5.1. Idioms in session-typed programming

The syntax extension for slot assignment. Through a few idioms in Gay-Vasconcelos style, we describe the syntax extensions and sketch the new communication primitives based on the new syntax. Firstly, we revise the slot-assigning primitives like `accept` returning a new session, rather than assigning it to a given slot, and utilise the `let%lin` syntax to assign a new session to a slot. For example, to accept a session at shared channel *ch* and assign it to the slot specified by *_n*, we previously wrote `accept` *ch* `~bindto:`*_n* `>>` *expr*. Instead, we now write it as follows:n

```
let%lin #_n = accept ch in expr
```

Equivalently,

```
accept ch >>= (fun%lin #_n -> expr)
```

The above code will be expanded to the following code:

```
accept ch >>= _lbind (fun tmp -> _put _n tmp >> expr)
```

where `tmp` is a fresh variable generated by the macro which is distinct from any variables in that program. The inserted code is highlighted with red colour. The operator `>>=` is a monadic bind operator as before, with the exception that it accepts functions of type `((.. -> ..) lbind)` in the right-hand side instead of ordinary functions `(.. -> ..)`. The function `_lbind` does nothing but wraps the argument function type f as `(`f `lbind)`, as required by `>>=`. The inserted `_put` $\_n$ `tmp` before *expr* assigns the new session bound to the variable `tmp` into the n-th slot provided that the slot is empty beforehand. The type f `lbind` states that a function of that type puts its session parameter immediately into a slot, thus ensures linearity. The operator `>>=` rejects normal (i.e. non-`lbind`) OCaml functions because it would duplicate the endpoint returned by the left-hand side.

Gay-Vasconcelos-style primitives. We further revise our library by following the design principle of Gay-Vasconcelos style as follows: (1) Communication primitives except for `close` returns a fresh session channel (or *continuation*) for the subsequent interaction rather than re-using the existing channels, and (2) output primitives (`send`, `select` and `deleg_send`) take a session channel as their *second* arguments. Thus, for example, we write `send 100` $\_n$ instead of previous `send` $\_n$ `100`, and we write

```
let%lin #_n = send "Hello" _n in
let%lin #_n = send "World" _n in
expr
```

instead of `send` $\_n$ `"Hello" >> send` $\_n$ `"World" >>` *expr*. This idiom is frequently seen in ML programming. `send` consumes the argument $\_n$, and the `let`-construct binds the continuation to another fresh variable $\_n$ which hides the previous occurrence of $\_n$.

As in Gay-Vasconcelos style, we revise the primitive of reception to `receive` which returns a pair of a received value *and a continuation*. We require pattern *pat* for value wrapped by `(W` *pat*`)` to prevent them from matched against continuations. Thus, the value returned by `receive` is matched with a pattern `(W` *pat*`), #_`n, as follows:

```
let%lin (W x), #_n = receive _n
in expr
```

Equivalently,

```
receive _n >>= (fun%lin ((W x), #_n) -> expr)
```

which binds the received value to the variable `x`. Both are expanded to

```
receive _n >>= _lbind (fun ((W x), tmp) -> _put _n tmp >> expr)
```

as we have shown in the `accept` case above.

Generalised choice. The `%lin` extension also integrates generalised choice in a consistent manner. We make the new `branch` function returning a continuation *wrapped* by a selected label. Branching among labels $lab_1, \ldots, lab_n$ at session $\_n$ is written as[11]:

```
match%lin branch _n with
  `lab1( #_n ) -> e1
| ..
| `labn( #_n ) -> en
```

equivalently,

```
branch _n >>= (function%lin
                 `lab1( #_n ) -> e1
               | ..
               | `labn( #_n ) -> en)
```

Once `branch` is called, the n-th slot is made empty and it waits for the peer to select a label. When lab_i is selected, `branch` returns the continuation wrapped with that label, and the slot pattern assigns the continuation to the slot n. The code above is expanded to:

```
branch _n >>=
  _lbind (function
            `lab1( tmp ) -> _put _n tmp >> e1
          | ..
          | `labn( tmp ) -> _put _n tmp >> en)
```

[11]Note that the OCaml construct `function` pat_1 `->` e_1 `| .. |` pat_n `->` e_n is an anonymous function which returns e_i when the argument matches pat_i.

Listing 12 The linearity monad

```
1  type ('pre, 'post, 'a) lmonad
2  type 'f lbind
3  val (>>=) : ('pre, 'mid, 'a) lmonad
4      -> ('a -> ('mid, 'post, 'b) lmonad) lbind
5      -> ('pre, 'post, 'b) lmonad
6  val return : 'a -> ('pre, 'pre, 'a) lmonad
7  type 'a data = W of 'a
8  val _lbind : 'f -> 'f lbind
9  val _put  : (empty, 'a, 'pre, 'post) -> 'a
10     -> ('pre, 'post, unit) lmonad
```

Again, `tmp` is fresh. Note that the slot number in the patterns can be different from n and different from each other, provided that the slot of that position is empty.

Selection. To select a label `` `L ``, we again adopt FuSe's style (§ 3.3) which uses the η-expansion of a variant tag ``(fun p -> `L(p))``, as follows:

```
select (fun p -> `L(p)) _n
```

It is the programmer's responsibility to pass an appropriately η-expanded function.

Furthermore, the current style generalises branching/selection to convey *payloads* within labels. For example, if a web server receives a URL payload with a `_GET` label, one can write

```
match%lin branch _n with
| `_GET(W url, #s) -> ..
```

The syntax extension requires `W` constructor pattern right before variable pattern `url`. Similarly, on the selecting side, the payload can be sent along with a label:

```
select (fun p -> `_GET(W "http://www.example.com", p)) _n
```

where the payload must be wrapped with `W` constructor, as in the receiver side.

3.5.2. Implementation by the linearity monad

We revise the type signature of the monad in Listing 12 and refer to this as the *linearity monad.* The only difference from the slot monad (Listing 4 in § 3.2) is that `>>=` (line 4) requires that the right-hand function argument must be `%lin` function of type `lbind`, as we have implied above. Thus, the linearity monad is a restricted (or *constrained* [25]) monad. The constructor of `W` has type `data` as defined in line 7. The signature of `_lbind` and `_put` are given in lines 8-10.

Listing 13 Gay-Vasconcelos-based session primitives in `session-ocaml`

```
val accept : 'p channel -> ('pre, 'pre, ('p, serv) sess) monad
val connect : 'p channel -> ('pre, 'pre, ('p, cli) sess) monad
val close : (([`close], 'r1*'r2) sess, empty, 'pre, 'post) slot
    -> ('pre, 'post, unit) monad
val send : 'v -> (([`msg of 'r1 * 'v * 'p], 'r1*'r2) sess, empty, 'pre, 'post) slot
    -> ('pre, 'post, ('p, 'r1*'r2) sess) monad
val receive : (([`msg of 'r2 * 'v * 'p], 'r1*'r2) sess, empty, 'pre, 'post) slot
    -> ('pre, 'post, 'v data * ('p, 'r1*'r2) sess) monad
val select : (('p,'r2*'r1) sess -> [>] as 'br)
    -> (([`branch of 'r1 * 'br],'r1*'r2) sess, empty, 'pre, 'post) slot
    -> ('pre, 'post, ('p,'r1*'r2) sess) monad
val branch : (([`branch of 'r2 * [>] as 'br], 'r1*'r2) sess, empty, 'pre, 'post) slot
    -> ('pre, 'post, 'br) monad
val deleg_send : (('pp, 'qq) sess, empty, 'mid, 'post) slot
    -> (([`deleg of 'r1 * ('pp, 'qq) sess * 'p], 'r1*'r2) sess, empty, 'pre, 'mid) slot
    -> ('pre, 'post, ('p, 'r1*'r2) sess) monad
val deleg_recv :
    (([`deleg of 'r2 * ('pp, 'qq) sess * 'p], 'r1*'r2) sess, empty, 'pre, 'post) slot
     -> ('pre, 'post, ('pp,'qq) sess * ('p,'r1*'r2) sess) monad
```

Listing 13 shows type signatures of the Gay-Vasconcelos-style primitives. The second type argument of each slot specifier type (`slot`) is `empty`, indicating that the session channel is "consumed" according to the linearity typing discipline.

3.5.3. A mathematical server example

Listing 14 shows a "mathematical server" example in FuSe [6]. We compare the FuSe description in the left column with our `session-ocaml` description in the right column. In FuSe (left), the `server` receives a tag labelled by one of `Quit`, `Plus` and `Eq`, and each label conveys a continuation. `Quit` closes the session. `Plus` and `Eq` receive a pair of integers and sends back the answer (of type `int` for `Plus`, or `bool` for `Eq`). `receive` is the replacement for `recv` and it returns a pair of a message and a continuation. `send` returns a continuation as well. Each continuation is bound to a variable shadowing the session channel variable used *e.g.* `let n, s = receive s`.

In the right column, the newly introduced extensions are highlighted with red colour. Remember that we have a variable `s` as an alias to lens `_0`. The argument to the `server` is changed to `()` because session channels are passed implicitly via slots to the next recursive call. The slot specifier `s` cannot be passed as an argument; otherwise polymorphism is lost and the slot specifier cannot have a different type at each occurrence. The significant difference lies in the syntactic extension `%lin` on the pattern matching (`match`) and the

Listing 14 The mathematical server in FuSe (left) and `session-ocaml` (right)

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

```
let rec server () =
  match%lin branch s with
  | `Quit #s -> close s
  | `Plus #s ->
    let%lin (W n), #s = receive s in
    let%lin (W m), #s = receive s in
    let%lin #s = send (n+m) s in
    server ()
  | `Eq #s ->
    let%lin (W n), #s = receive s in
    let%lin (W m), #s = receive s in
    let%lin #s = send (n=m) s in
    server ()
```

`let`-binding.

Thanks to polarised session types, we enjoy a natural, *prefixing*-style session types while FuSe needs a translation tool between FuSe types and the session type notation, as we will mention in Section 6.2.

We believe that this linear-pattern based syntax is *idiomatic* in the sense that it is a functional formulation of session types. We write `let%lin #s = accept ch in` e and `let%lin #s = connect ch in` e instead of `accept ch ~bindto:s >>` e and `connect ch ~bindto:s >>` e, respectively.

Notes on polarity. The resulting protocol type of Listing 14 is the following (assuming that `server`'s polarity is `serv`):

```
[`branch of req *
  [ `Quit of ([`close], serv) sess
  | `Plus of ([`msg of req * int * [`msg of req * int *
               [`msg of resp * int * 's]]], serv) sess
  | `Eq of   ([`msg of req * int * [`msg of req * int *
               [`msg of resp * bool * 's]]], serv) sess]] as 's
```

A drawback is that a polarity from the receiving endpoint is inserted in the continuation type in the labels of the protocol type. The continuation types of the labels above are polarised by `serv`. This is due to the fact that we cannot say anything in the continuation of `branch` (type `'br` in lines 12-13 of Listing 13) since the pattern determines its whole structure, and the polarised session type in the selecting side enforces the polarity in the session of the label. This is regarded as a notational overhead since the polarity polymorphism (§ 3.1) applies here.

3.6. Distributed implementation with ad hoc polymorphism

In the implementation of communications between processes over the network stack such as TCP, the messages need to be treated uniformly as packets in order to use existing communication APIs. Thus, communicating messages are converted to a fixed type at sending and the type information is recovered when received. This issue is solved via *ad hoc polymorphism*. It offers a way to specify different behaviours depending on the type of arguments, and widely utilised in functional programming, as in the type classes in Haskell [26] and implicit parameters in Scala [27]. Recently, the two implementations has been proposed for ad hoc polymorphism in OCaml as follows: (1) *Modular-implicits* [28]; and (2) *Ppx_implicits* [29]. Although Modular-implicits is planned to be finally integrated into mainstream OCaml, but does not currently work in the latest OCaml equipped with Flambda optimiser. On the other hand, Ppx_implicits works with Flambda optimiser. In what follows, we use Ppx_implicits.

Ppx_implicits is a type-aware preprocessor of OCaml, which provides ad hoc polymorphism based on the implicit parameters. It exploits optional arguments in vanilla OCaml rather than extending the current syntax and types. Implicit values are of type `(`τ`, [%imp` *module*`]) Ppx_implicits.t` where τ is the type of a function to be passed and *module* is the name of the module where the implicit values are defined. Followings are the types for such implicit values[12]:

```
type ('c,'v) sender = ('c -> 'v -> unit, [%imp Senders]) Ppx_implicits.t
type ('c,'v) receiver = ('c -> 'v, [%imp Receivers]) Ppx_implicits.t
```

`('c, 'v) sender` is for sending values and `('c, 'v) receiver` are for receiving values where `'c` is the type of communication medium and `'v` is the type of the messages. The communication primitives are defined as follows:

```
val send : ?sender:('c, 'v) sender -> 'v
   -> (([`msg of 'r1 * 'v * 'p], 'r1*'r2, 'c) dsess, empty, 'pre, 'post) slot
   -> ('pre, 'post, ('p, 'r1*'r2, 'c) dsess) monad
val receive : ?receiver:('c ,'v) receiver
   -> (([`msg of 'r2 * 'v * 'p], 'r1*'r2, 'c) dsess, empty, 'pre, 'post) slot
   -> ('pre, 'post, ('v * ('p, 'r1*'r2, 'c) dsess) lin) monad
```

where the red part is the only difference from Listing 13. We extend the `lsess` type to `('p, 'q, 'c) dsess` to carry the medium type `'c`. The named

[12]This description is based on version `0.3.0`.

optional arguments `?sender` in `send` and `?receiver` in `receive` is filled by the Ppx_implicits preprocessor.

For example, to implement the mathematical server shown in Listing 14 in the distributed processes, the arguments are inserted by looking up the modules with the name `Sender` or `Receiver`.

```
module Sender = struct
  let write_int (_,oc) (i:int) = output_value oc i
end
module Receiver = struct
  let read_int (ic,_) : int   = input_value ic
  let read_bool (ic,_) : bool = input_value ic
end
```

4. Applications

We present two applications of `session-ocaml`: the *Travel Agency usecase* and an *SMTP client*. The Travel Agency usecase (§ 4.1) involves session delegation; and it is shown that static typing of `session-ocaml` can effectively find errors in the protocol. The SMTP client (§ 4.2) captures a flow-sensitive behaviour in a real-world network protocol and it uses message payloads in the SMTP commands and response code based on the technique developed in § 3.5. It is implemented on the distributed implementation in § 3.6. The material in § 4.1 depends only on the materials from § 3.1 to § 3.3.

4.1. Travel agency

We demonstrate programming in `session-ocaml` using the Travel agency scenario from [7], which consists of typical patterns found in business and financial protocols. The scenario is played by three participants: `customer`, `agency` and `service` (Listing 15). `customer` knows `agency` while `customer` and `service` initially do not know each other, and `agency` mediates a deal between `customer` and `service` by the session delegation.

(1) `customer` begins a session for ordering a ticket with `agency` and binds it to the 0-th slot (line 2). Then `customer` requests and receives the price for the desired journey after sending the `quote` label (lines 3-5). In our scenario, `customer` requests `"London to Paris"` (line 4) and `agency` replies with a fixed price `80.00` (lines 5 and 22).

Then `customer` might send the `agree` label to proceed the transaction with the current price (lines 10-15). If `customer` does not agree with the price,

Listing 15 Travel agency

```
1  let customer cst_ch =
2    connect cst_ch ~bindto:_0 >>
3    [%select _0 `quote] >>
4    send _0 "London to Paris" >>
5    recv _0 >>= fun cost ->
6    if cost > 100. then
7      [%select _0 `reject] >>
8      close _0
9    else
10     [%select _0 `agree] >>
11     send _0 (Address("London")) >>
12     recv _0 >>= fun (d : date) ->
13     close _0 >>
14     (Printf.printf "cost: %f\n" cost;
15     return ())
16 let agency cst_ch svc_ch =
17   accept cst_ch ~bindto:_0 >>
18   let rec loop () =
19     match%branch _0 with
20     | `quote  ->
21       recv _0 >>= fun dest ->
22       send _0 80.00 >>
23       loop ()
24     | `reject -> close _0
25     | `agree  ->
26       connect svc_ch ~bindto:_1 >>
27       deleg_send _1 ~release:_0 >>
28       close _1
29   in loop ()
30 let service svc_ch =
31   accept svc_ch ~bindto:_1 >>
32   deleg_recv _1 ~bindto:_0 >>
33   recv _1 >>= fun (Address(addr)) ->
34   send _0 (now()) >>
35   close _0 >>
36   close _1
```

`customer` can cancel the transaction by sending the `reject` label (lines 7-8). Alternatively, `customer` can send `quote` again and this will be repeated an arbitrary number of times for different journeys. In our program, `customer` agrees with `agency` at a price less than `100.0`, or otherwise rejects it and terminates the transaction.

Next, if `customer` agrees with the price, `agency` opens the session with `service` and binds it to the 1st slot (line 26). Then it delegates the interactions with `customer` remaining in the 0-th slot to `service` (line 27). `customer` sends the billing address being unaware that the customer is now talking to `service`. `service` replies with the dispatch date (`now()`) for the purchased tickets to close the sessions (lines 33-36).

The type of protocol between `customer` and `agency` is inferred as:

```
[`branch of req *
 [`quote of [`msg of req * string * [`msg of resp * float * 'a]]
 |`reject of [`close]
 |`agree of [`msg of req * addr * [`msg of resp * date * [`close]]]]] as 'a
```

Delegation from `agency` to `service` is inferred in the channel of `service` as:

```
[`deleg of req *
  ([`msg of 'r1 * addr * [`msg of 'r2 * date * [`close]]], 'r1*'r2) sess * [`close]]
```

The delegated type is polymorphic on the polarities and communication directions (§ 3.1). Hence the `service` can handle both polarities. After `agree` in the protocol above `'r1` becomes `req` and `'r2` becomes `resp`. Delegations

Listing 16 SMTP client session types

```
1  type 'p cont = ('p,cli,stream) dsess
2  type 'p contR = ('p,serv,stream) dsess
3  type smtp =
4    [`branch of resp * [`_200 of string list data *
5    [`branch of req * [`EHLO of string * [`branch of resp * [`_200 of string list data *
6      mailloop cont]] contR]] cont]]
7  and mailloop = [`branch of req *
8      [`MAIL of string * [`branch of resp * [`_200 of string list data * rcptloop cont]] contR
9      |`QUIT of [`close] contR]]
10 and rcptloop = [`branch of req *
11     [`RCPT of string * [`branch of resp *
12         [`_200 of string list data * rcptloop cont
13         |`_500 of string list data * [`branch of req * [`QUIT of [`close] contR]] cont]] contR
14     |`DATA of [`branch of resp *
15         [`_354 of string list data *
16           [`msg of req * mailbody * [`branch of resp * [`_200 of string list data *
17             mailloop cont]]] cont]] contR]]
```

with the polarised session types and slots effectively give a way to coordinate *higher order* communications by the link mobility.

Static session type checking of delegations eases the protocol analysis even with the indirect nature of delegation. Consider a case that `service` changes its behaviour to receive the payment method in addition to the billing address typed as `addr * paymeth`. Now the inferred protocol type at `service` would be:

```
[`deleg of req *
  ([`msg of 'r1 * (addr * paymeth) * [`msg of 'r2 * date * [`close]]], 'r1*'r2) sess *
    [`close]]
```

If `customer` remains unchanged, there is a type error in the communication between `customer` and `service`. `session-ocaml` detects this error by checking the duality through slots. Without static typing, the run-time error would be deferred until the beginning of actual `client-service` communication.

4.2. SMTP protocol

This section shows an SMTP client implementation using `%lin` extended syntax as well as ad hoc polymorphism for the distributed implementation introduced in § 3.5 and § 3.6. Listing 16 shows session types for a SMTP client. Lines 1 and 2 declare the types `cont` and `contR` as a shorthand for polarised session types with a specific polarity, which is required for the continuation type of labelled branches (see § 3.5). The type `stream` specifies TCP streams as a communication medium. In the SMTP, a client sends a command and receives the reply code for the command from the server.

Listing 17 SMTP client

```
 1 let sendmail from to_ mailbody ()
 2   : (((smtp,cli,stream) dsess * 'p) lin, (empty * 'p) lin, unit lin) lmonad =
 3   let%lin `_200(W _,#s) = branch s in                    (* 220 greeting *)
 4   let%lin #s = select s (fun x -> `EHLO(W "me.example.com",x)) in  (* send EHLO command *)
 5   let%lin `_200(W _,#s) =  branch s in                   (* 250 Ok *)
 6   let%lin #s = select s (fun x -> `MAIL(W from,x)) in    (* MAIL FROM: <sender> *)
 7   let%lin `_200(W _,#s) = branch s in                    (* 250 Ok *)
 8   let%lin #s = select s (fun x -> `RCPT(W to_,x)) in     (* RCPT TO: <recipient> *)
 9   begin match%lin branch s with                          (* branch according to reply *)
10   | `_200(W _,#s) ->                                     (* if 250 Ok *)
11     let%lin #s = select s (fun x -> `DATA(x)) in           (* DATA *)
12     let%lin `_354(W _, #s) = branch s in                   (* 354 end data with CRLF.CRLF*)
13     let%lin #s = send s (MailBody mailbody) in             (* mail body *)
14     let%lin `_200(W _, #s) = branch s in                   (* 250 queued *)
15     let%lin #s = select s (fun x -> `QUIT(x)) in           (* QUIT *)
16     return ()
17   | `_500 (W txt,#s) ->                                  (* if 550 recipient rejected *)
18     (print_endline "Email sending failed. Detail:";        (* print error messages *)
19      List.iter print_endline txt; return ()) >>
20     let%lin #s = select s (fun x -> `QUIT(x)) in           (* QUIT *)
21     return ()
22   end >> close s                                         (* disconnect *)
23 let smtp_client host port from to_ mailbody () =
24   let%lin #s = tcp_connect ~host ~port in
25   sendmail from to_ mailbody ()
```

Since the reply codes are categorized by the hundreds digit, we write `_$n$00` for the label to receive a reply code with the number of n-hundred. Lines 4-6 specify the initial negotiation in SMTP that comprises the first `220` greeting reply matched by `` `_200 `` from the server (line 4), `EHLO` command from the client (line 5) and continuation to the main part (line 6). Lines 7-9 are branching between `MAIL FROM` (which specifies the sender) and `QUIT` commands. Lines 10-17 specify arbitrary many `RCPT TO` (specifies a recipient) commands and their replies. The reply can be either `250 Ok` or an error like `550 Recipient address rejected`. Once an error occurs, the session terminates. `250 Ok` is followed by the `DATA` command, `354` reply, the body of e-mail, and recursion until the next `MAIL FROM` or `QUIT`. Listing 17 shows the SMTP client. Lines 1-22 declare the function `sendmail`. Each line with `branch` matches a linear pattern using either `let%lin` or `match%lin` constructs. Patterns for non-linear values are surrounded by `W`, while linear values (for continuations) are prefixed by `#`. After a TCP connection is established via `tcp_connect` on lines 23-25, `sendmail` is invoked.

The introduction of the ad hoc polymorphism enables communications to be implemented directly using APIs for TCP communication. This direct

Listing 18 SMTP client (de)serialisers

```
(* Instance declaration for SMTP command serialisers *)
module Senders = struct
  let write out str = output_string out str; flush out
  let _ehlo c (`EHLO (W v,_) : [`EHLO of _]) = write c  "EHLO " ^ v ^ "\r\n"
  let _mailbody c (MailBody s) = write c  s ^ "\r\n.\r\n"
  let _mail c : [`MAIL of _] -> unit = function
    | `MAIL(W v,_) -> write c  "MAIL FROM:" ^ v ^ "\r\n"
  let _quit c : [`QUIT of _] -> unit = function
    | `QUIT(_) -> write c "QUIT\r\n"
  let _rcpt_or_data c = (function
    | `RCPT(W v,_) -> write c  "RCPT TO:" ^ v ^ "\r\n"
    | `DATA(_) -> write c "DATA\r\n" : [`RCPT of _ | `DATA of _] -> unit)
  let _mail_or_quit c = (function
    | `MAIL(_) as m -> _mail c m
    | `QUIT(_) as m -> _quit c m : [`MAIL of _ | `QUIT of _] -> unit)
end
(* Instance declaration for SMTP reply deserialisers *)
module Receivers = struct
  let r200 = ('2', (fun tcp str -> `_200(W str, _mksess tcp)))
  let r354 = ('3', (fun tcp str -> `_354(W str, _mksess tcp)))
  let r500 = ('5', (fun tcp str -> `_500(W str, _mksess tcp)))
  let _200 : stream -> [`_200 of _ * _] = fun c -> parse_reply [r200] c
  let _200_or_500 : stream -> [`_200 of _ * _ | `_500 of _ * _]  =
    fun c -> parse_reply [r200; r500] c
  let _354 : stream -> [`_354 of _ * _] = fun c -> parse_reply [r354] c
end
```

correspondence is important in the sense that reliable communications are ensured by the session based programming. Listing 18 presents the serialiser and the deserialiser for the SMTP commands and replies. Ad hoc polymorphism improves the implementation of the protocol in that there is no need to write an *adapter*. Otherwise, we need an adapter to mediate between the heterogeneous session-based stream and the homogeneous TCP streams, as in [1].

One of the advantages of the OCaml implementation over the existing Haskell implementation in [4] is that `session-ocaml` can utilise *equi-recursive types* in OCaml to encode the session type recursion, keeping from adding `unwind` annotations repeatedly to unfold iso-recursive types in Haskell.

5. Performance

In this section, we evaluate the run-time performance of `session-ocaml`. We measure the performance of `session-ocaml` through benchmarks in two different settings: Synchronous/inter-thread (§ 5.1) and asynchronous/inter-

Listing 19 The benchmark program in `session-ocaml`

```
0  let s = _0
1  let rec server ch () =
2    let rec aux () =
3      match%lin branch s with
4      | `True(#s) ->
5          let%lin W n, #s = receive s in
6          let%lin W m, #s = receive s in
7          let%lin #s      = send (n + m) s in
8          aux ()
9      | `False(#s) ->
10        close s
11     in aux ()
12 let client ch cnt () =
13   let rec aux acc n =
14     if n = 0 then begin
15       let%lin #s =
16           select (fun x -> `False(x)) s in
17       close s >>
18       return acc
19     end else begin
20       let%lin #s =
21           select (fun x -> `True(x)) s in
22       let%lin #s = send acc s in
23       let%lin #s = send n s in
24       let%lin W res, #s = receive s in
25       aux res (n - 1)
26     end
27   in aux 0 cnt
```

process (§ 5.2). We present our benchmark results following the way shown in [6].

The first benchmark is employed using the *mathematical server* program in [6] and its transliteration to `session-ocaml` in Listing 19. The server first offers branching (line 3) on the labels `` `True `` and `` `False ``. If the client selects `` `False ``, the session finishes (lines 9-10). If the client selects `` `True `` (line 4-8), the server receives two integers, sends back their sum, and recurs. The client computes the sum of the first `n` numbers using the server. If `n` is 0, the client selects `` `False `` and the session finishes (line 14-18). Otherwise (line 19-25), the client selects `` `True ``, sends a pair of the accumulated sum and `n`, then receives the new sum and recurs.

For *delegation*, we use the benchmark in Listing 20. This is a revised version of Listing 19. Changes from Listing 19 are highlighted in red. After communicating the first integer, the client creates a pair of new session endpoints `t` and `u` (line 25) with `create`, then delegates `u` to the server (line 26). It sends the second integer on `t` (line 27), receives the sum on `t` (line 28) then closes `t` and recurs (lines 29-30). The server receives the delegated session and assigns it to `t` (line 6), then receives the next integer and sends back the sum on `t` (lines 7-8), and closes `t` then recurs (lines 9-10).

We measure throughputs of benchmark programs by the numbers of communicated messages per second. In Listing 19, each iteration communicates four messages and the total number of communicated messages is $4n + 1$ for calculating the first n sum. We present the throughputs of 1,000 runs

Listing 20 The benchmark program with delegation

```
let s = _0 and t = _1 and u = _2
let rec server ch () =
  let rec aux () =
    match%lin branch s with
    | `True(#s) ->
        let%lin W n, #s = receive s in
        let%lin #t,  #s = deleg_recv s in
        let%lin W m, #t = receive t in
        let%lin #t      = send (n + m) t in
        close t >>
        aux ()
    | `False(#s) ->
       close s
  in aux ()
let client ch cnt () =
  let rec aux acc n =
    if n = 0 then begin
      let%lin #s =
          select (fun x -> `False(x)) s in
      close s >>
      return acc
    end else begin
      let%lin #s =
          select (fun x -> `True(x)) s in
      let%lin #s = send acc s in
      let%lin #t, #u = create () in
      let%lin #s = deleg_send u s in
      let%lin #t = send n t in
      let%lin W res, #t = receive t in
      close t >>
      aux res (n - 1)
    end
  in aux 0 cnt
```

where each run computes the sum of 8,000 numbers. We used the native `ocamlopt` compiler of OCaml 4.05.0 with the new Flambda optimiser[13]. For comparison with FuSe, we modified the original FuSe 0.7 to support asynchronous and inter-process communications. The whole program is available at https://github.com/keigoi/FuSe-clone/. The benchmark programs were executed on MacBook Pro with 2.7 GHz quad core CPU and 16 GB of memory (model MacBookPro10,1).

5.1. Synchronous and inter-thread communication

Figure 2 shows the results of the benchmark using the standard *boxplot* diagram to illustrate the variance in execution time. The unit of throughput is the numbers of messages per second. The throughputs of 500 runs are in the boxes and the horizontal line in each box shows the median. The length of each whisker is $1.5IQR$ and circles on the whiskers are outliers.

The first column of Figure 2 is the throughputs of FuSe with no run-time checking. In this case, it should exhibit the ideal performance that could be achieved if OCaml would be equipped with linear types. The second column of Figure 2 is the throughputs of FuSe with the run-time linearity checking. The third column of Figure 2 is the *monadic* version of FuSe which checks

[13] It generates faster code than the previous optimiser in general.

types statically but creates a numerous number of closures. As reported in [6], these three FuSe versions display little difference.

The fourth column of Figure 2 is the throughputs of `session-ocaml`. Its performance is seen on par with the three FuSe versions. Although we first suspected that function calls on lenses in `session-ocaml` slow down the throughput, the benchmark result shows that the lens manipulations have little impact on performance.

For delegation benchmark, the fifth column of Figure 2 is FuSe and the sixth column is `session-ocaml`. `Session-ocaml` performs slightly less, and the throughput difference is less than 1 percent. We argue that this performance degradation does not have any significant effect; although we expect more degradation for a very large number of slots due to the nested structure of slots, Listing 20 uses three slots and we suspect that it is unrealistic to have so many slots in a thread, as we discussed in § 3.3. Note that with delegation FuSe possibly raises a run-time error when the linearity is violated, while `session-ocaml` statically points out the error before execution.

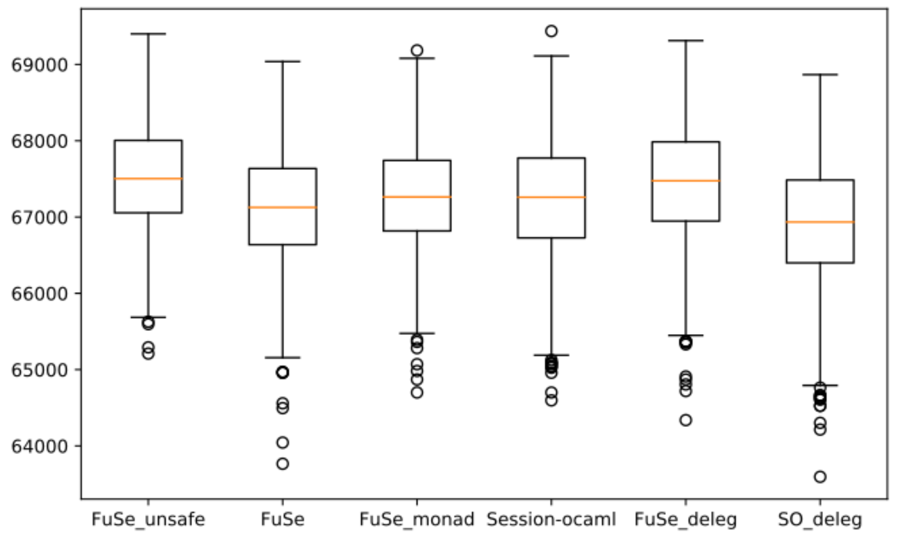


Figure 2: Performance benchmark between FuSe and `session-ocaml`

5.2. Asynchronous and inter-process communication

The inter-process communication is instrumented using `Unix.pipe`. We invoke `client` process and `server` process is forked using `Unix.fork` with two inter process pipes for both directions, where the communications are *asynchronous*. OCaml messages are (de)serialised in a standard way (`Pervasives.output_value` and `input_value`).

Figure 3 is the results of benchmarks for the three FuSe versions in § 5.1 and `session-ocaml`. The overall throughputs in all programs are increased by ca. 60 percent than the benchmarks in the previous section, since communications are asynchronous. Difference between the safe FuSe versions and `session-ocaml` is less than 1 percent; however, the unchecked version is slightly faster (ca. 4 percent). From this result, one can see that monadic operations still incur a cost, although static checking of linearity removes runtime costs for dynamic checking. This is the *real* cost paid for linearity checking in OCaml *statically*, which is incurred by monads. By our benchmark results, we remark that the dynamic linearity check affects the performance without using the monad techniques.

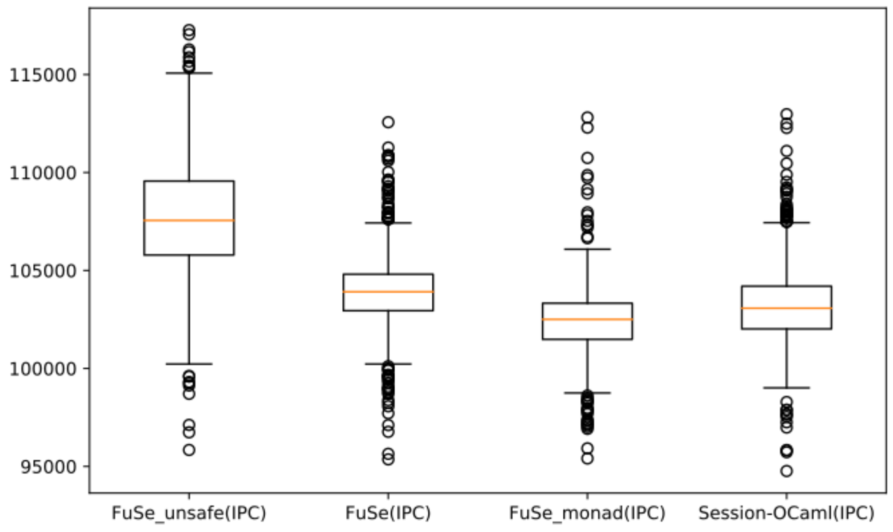


Figure 3: Performance benchmark between FuSe and `session-ocaml` (IPC with pipes)

The reason for taking an additional inter-process benchmark is slightly OCaml-specific: Two OCaml threads in a process cannot run simultaneously in parallel because the garbage collector of OCaml does not run concurrently (albeit it performs really well in single-threaded setting [30]). This fact would improve the entire throughput of the system in an *asynchronous* setting. In fact, the implementation using inter-process communication is a way to take the advantage of performance by parallelism with the sequential garbage collectors in OCaml.

In summary, `session-ocaml` enjoys performance equivalent to the dynamic linearity checking, with the benefits of static checking on session types. It does not make any significant difference in both synchronous/asynchronous and inter-thread/inter-process setting. A comparison with an *unsafe* implementation highlighted the cost of monads, which can only be eliminated by dropping linearity checking.

6. Related work

6.1. Implementations in Haskell

The first implementation of session types by Neubauer and Thiemann [31] deals with the first-order single-channel session types with recursions. Using parameterised monads, Pucella and Tov [9] provide multiple sessions, but checking session types is not automatic where manual reordering of symbol tables is required. Imai et al. [4] extend [9] with delegation, automatically handling multiple sessions in a user-friendly manner by using type-level functions. Orchard and Yoshida [8] use an embedding of the effect systems in Haskell via graded monads by encoding the session-typed π-calculus into PCF with an effect system. Lindley and Morris [5] provide an embedding of the GV session-typed functional calculus [21] into Haskell, built on a linear λ-calculus embedding by Polakow [32]. In [5, 8, 9, 4], duality inference of session types is represented by the multi-parameter type classes with functional dependencies [33]; For instance, `class Dual t t'| t -> t', t' -> t` declares that `t` can be inferred from its dual `t'` and vice versa. Since all of the above works depend on type-level features in Haskell, the techniques are not directly applicable to OCaml. See [34] for a detailed survey.

6.2. Implementations in OCaml

Padovani [6] implements multiple sessions with dynamic linearity checking and its single-session version with static checking in OCaml. FuSe is capable of checking a single session statically, but for multiple sessions with delegation, dynamic checking is necessary. Our `session-ocaml` achieves static typing for multiple sessions with delegation by introducing session manipulations using lenses. `session-ocaml` provides an idiomatic way to declare branching with arbitrary labels. The FuSe implementation has been recently extended to *context-free session types* [35] by adding an *endpoint* attribute to session types [36]. Furthermore, Melgratti and Padovani [37] developed a monitoring technique for higher-order sessions on top of FuSe.

The following example shows that `session-ocaml` can statically check a linearity violation, while FuSe is only able to dynamically detect the violation at the runtime.

```
let rec loop () = let s = send "*" s in
                  match branch s with `stop s -> close s |`cont _ -> loop ()
```

`loop` sends `"*"` repeatedly until it receives label `stop`. Although the endpoint `s` should be used linearly, the condition is violated at the beginning of the second iteration since the endpoint is disposed of by using the wildcard `_` at the end of the `loop`. In FuSe 0.7, `loop` is well-typed but terminates in error `InvalidEndpoint` at runtime. In `session-ocaml`, this results in an error at static type checking as the slot `s` passed to the recursive call of `loop` is `empty`, while `loop` expects `s` to have a session endpoint.

On duality inference, a simple approach in OCaml is introduced by Pucella and Tov [9]. The idea in [9] is to keep a pair of the current session and its dual at every step; therefore the notational size of a session type is twice as big as that in [2]. FuSe [6] reduces its size by almost half using the encoding technique in [38] by modelling binary session types as a chain of linear channel types as follows. A session type in FuSe `('a,'b) t` prescribes input (`'a`) and output (`'b`) capabilities. A transmission and a reception of a value `'v` followed by a session `('a,'b) t` are represented as `(_0,'v*('a,'b) t) t` and `('v*('a,'b) t,_0) t` respectively, where `_0` means "no message"; then the dual of a session type is obtained by swapping the top pair of the type. For example, in a simplified variant of the logic operation server in Listing 2, the protocol type of `log_ch` becomes:

```
[`msg of req*binop*[`msg of req*(bool*bool)*[`msg of resp*bool*[`close]]]]
```

In FuSe, at server side, the channel should be inferred as:

```
(binop*((bool*bool)*(_0,bool*(_0,_0) t) t,_0) t,_0) t
```

Due to a sequence of flipping capability pairs, more effort is needed to understand the protocol. The difficulty arises when multiple nestings are present, which is inevitable even without recursion and branch. To recover the readability, FuSe supplies the translation tool called *Rosetta* between FuSe types and session type notation with the prefixing style. Our polarised session types are directly represented in a *prefixing* manner with the slight restriction shown in § 3.3.

7. Conclusion

We have shown `session-ocaml`, a library for session-typed communications which supports multiple simultaneous sessions with static type checking including delegation in OCaml. `session-ocaml` generalises the authors' previous work in Haskell [4] by replacing type-level functions in Haskell with lenses in OCaml. Our contributions in this paper are summarised as follows. (1) Based on lenses and the slot monad, we achieved fully static checking of session types by the OCaml type system without adding any extension to the language. To the authors' knowledge, this is the first implementation which combines lenses and a parameterised monad. In the existing implementations [9, 6], static type checking has been limited to single sessions; (2) We proposed two macros for arbitrarily labelled branches, `%branch` and `%lin`. The `%branch` macro "patches up" only the branching part where linear variables are inevitably exposed due to limitation on polymorphic variants, keeping the original Honda-Vasconcelos-Kubo style [2] session programming which re-uses the same session endpoints in a series of communications, while `%lin` macro offers *linear-pattern matching*, aiming at a more idiomatic formulation based on Gay-Vasconcelos style [24]; (3) We proposed the *polarised session types* for the session type inference solely based on the built-in type unification of OCaml. This encoding efficiently ensures communication safety by checking the equivalence of protocol types inferred at both ends with different polarities; and (4) We presented a performance comparison between FuSe and `session-ocaml`, showing that our lens-based formulation for static checking does not incur significant overhead in practice. Furthermore, we found that certain overhead for linearity checking emerges in asynchronous communication, which is not stated in [6].

Generalisability. The `session-ocaml` library without macros depends only on the parametric polymorphism and is easily portable to functional programming languages such as F#, Standard ML, and Haskell. Type inference plays a key role in using lenses without the burden of writing any type annotations. These languages have a nearly complete type inference system, hence it is relatively easy to apply the method presented in this paper to such languages. For recursive type definitions, since equi-recursive types appear only in OCaml, we cannot directly encode recursive sessions of the form $\mu\alpha.p$ and the type of infinitely many slots `all_empty = empty * all_empty` in the other programming languages. Following an iso-recursive encoding of the session type recursion in a language-independent manner, it is possible to encode the recursive types as required [9]. As for slots, a fixed number suffices for most cases. Other programming languages such as Scala, Java and C# have a limited type inference system, and our technique is not portable to these languages. Also, it is not apparent how the macros in `session-ocaml` can be implemented in the other programming languages. Notably, generalised-branching/selection and linearity extensions are only possible with macros. The binary-labelled choices are portable to functional programming languages and can be used in place of the generalised choices.

Notational overhead. We discuss the notational overhead in `session-ocaml`, in terms of the programming style in OCaml. (1) *Monadic programming.* There is a notational overhead in using the bind operator e_1 `>>= fun` x `->` e_2 in a sequential composition, instead of standard `let` x `=` e_1 `in` e_2 construct of OCaml. As there are several well-known OCaml libraries using monads such as `Lwt` [39] and `Async` [40], this style is gaining popularity among many OCaml programmers. Then, we believe this is not a significant issue. (2) *Lenses and slots.* Slot manipulation using lenses enables programmers to flexibly specify slots storing sessions. Despite its conceptual indirectness, the notational overhead on using lenses is minor. On the other hand, [9] requires manual reordering of slots by `swap` and `dig`. (3) *Branching.* The construct for generalised branching `match%branch` provides a handy way to write branching sessions, sacrificing portability to the other programming languages. (4) *Linearity.* In § 3.5 we developed a macro `fun%lin` which enables programming with linear types, writing `#_`n for every linear pattern and `w` for other variable patterns .

Our approach using slots to deal with simultaneous multiple sessions resembles parameterised session types [41, 42], and it is smoothly extendable to the multiparty session type framework [43]. For future work, we plan to investigate code generations from Scribble [44] (a protocol description language for the multiparty session types) along the line of [13, 12] integrating `session-ocaml` with parameterised features [41, 42].

Acknowledgements

We thank the anonymous reviewers for the thorough review and constructive comments. We thank Raymond Hu for his comments on the early version of the paper. We thank Jun Furuse for his advice on using his `ppx_implicits`. The third author thanks the JSPS bilateral research with

NFSC for fruitful discussion. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1; by EU FP7 612985 (UPSCALE), and COST Action IC1405 (RC); by JSPS International Fellowships (S15051), and KAKENHI JP17K12662, JP25280023 and JP17H01722 from JSPS, Japan.

References

[1] K. Imai, N. Yoshida, S. Yuen, Session-ocaml: A session-based library with polarities and lenses, in: Coordination Models and Languages - 19th IFIP WG 6.1 International Conference, COORDINATION 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings, 2017, pp. 99–118. doi:10.1007/978-3-319-59746-1_6.

[2] K. Honda, V. T. Vasconcelos, M. Kubo, Language Primitives and Type Discipline for Structured Communication-Based Programming, in: Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, 1998, pp. 122–138. doi:10.1007/BFb0053567.

[3] R. Milner, Communicating and Mobile Systems: the π-Calculus, Cambridge University Press, 1999.

[4] K. Imai, S. Yuen, K. Agusa, Session Type Inference in Haskell, in: Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010., 2010, pp. 74–91. doi:10.4204/EPTCS.69.6.

[5] S. Lindley, J. G. Morris, Embedding Session Types in Haskell, in: Haskell 2016: Proceedings of the 9th International Symposium on Haskell, ACM, 2016, pp. 133–145. doi:10.1145/2976002.2976018.

[6] L. Padovani, A simple library implementation of binary sessions, J. Funct. Program. 27 (2017) e4. doi:10.1017/S0956796816000289.

[7] R. Hu, N. Yoshida, K. Honda, Session-Based Distributed Programming in Java, in: ECOOP 2008 - Object-Oriented Programming, 22nd European

Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings, 2008, pp. 516–541. doi:10.1007/978-3-540-70592-5_22.

[8] D. Orchard, N. Yoshida, Effects as sessions, sessions as effects, in: POPL 2016: 43th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2016, pp. 568–581. doi:10.1145/2837614.2837634.

[9] R. Pucella, J. A. Tov, Haskell Session Types with (Almost) No Class, in: Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell, ACM, 2008, pp. 25–36. doi:10.1145/1411286.1411290.

[10] T. B. L. Jespersen, P. Munksgaard, K. F. Larsen, Session Types for Rust, in: WGP 2015: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, ACM, 2015, pp. 13–22. doi:10.1145/2808098.2808100.

[11] A. Scalas, N. Yoshida, Lightweight Session Programming in Scala, in: ECOOP 2016: 30th European Conference on Object-Oriented Programming, Vol. 56 of LIPIcs, Dagstuhl, 2016, pp. 21:1–21:28. doi:10.4230/LIPIcs.ECOOP.2016.21.

[12] R. Hu, N. Yoshida, Hybrid Session Verification Through Endpoint API Generation, in: Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, 2016, pp. 401–418. doi:10.1007/978-3-662-49665-7_24.

[13] R. Hu, N. Yoshida, Explicit connection actions in multiparty session types, in: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, 2017, pp. 116–133. doi:10.1007/978-3-662-54494-5_7.

[14] J. Garrigue, Safeio (a mailing-list post), available at https://github.com/garrigue/safeio (2006).

[15] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, ACM, New York, NY, USA, 2005, pp. 233–246. doi:10.1145/1040305.1040325.
URL http://doi.acm.org/10.1145/1040305.1040325

[16] M. Pickering, J. Gibbons, N. Wu, Profunctor Optics: Modular Data Accessors, The Art, Science, and Engineering of Programming 1 (2). doi:10.22152/programming-journal.org/2017/1/7.

[17] S. Gay, M. Hole, Subtyping for Session Types in the Pi Calculus, Acta Informatica 42 (2) (2005) 191–225. doi:10.1007/s00236-005-0177-z.

[18] N. Yoshida, V. T. Vasconcelos, Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication, Electr. Notes Theor. Comput. Sci. 171 (4) (2007) 73–93. doi:10.1016/j.entcs.2007.02.056.

[19] R. Atkey, Parameterised notions of computation, J. Funct. Program. 19 (3-4) (2009) 335–376. doi:10.1017/S095679680900728X.

[20] B. C. Pierce, Recursive Types, in: Types and Programming Languages, MIT Press, 2002, Ch. 20, pp. 267–280.

[21] P. Wadler, Propositions as sessions, in: ICFP '12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ACM, 2012, pp. 273–286. doi:10.1145/2364527.2364568.

[22] N. Kobayashi, Type Systems for Concurrent Programs, in: 10th Anniversary Colloquium of UNU/IIST, Vol. 2757 of Lecture Notes in Computer Science, 2002, pp. 439–453. doi:10.1007/978-3-540-40007-3_26.

[23] J. Garrigue, J. L. Normand, Adding GADTs to OCaml: the direct approach, in *ACM SIGPLAN Workshop on ML 2011*. Slides available at https://www.math.nagoya-u.ac.jp/~garrigue/papers/ml2011-show.pdf (Septempber 2011).

[24] S. j. Gay, V. T. Vasconcelos, Linear Type Theory for Asynchronous Session Types, Journal of Functional Programming 20 (1) (2010) 19–50. doi:10.1017/S0956796809990268.

[25] N. Sculthorpe, J. Bracker, G. Giorgidze, A. Gill, The Constrained-Monad Problem, in: ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013, 2013, pp. 287–298. doi:10.1145/2500365.2500602.

[26] M. J. Simon Peyton Jones, E. Meijer, Type classes: an exploration of the design space, in: Proceedings of the Second Haskell Workshop, 1997.

[27] M. Odersky, Poor Man's Type Classes, in: IFIP WG2.8 working group meeting, 2006, available at http://scala-lang.org/old/sites/default/files/odersky/wg2.8-boston06.pdf.

[28] L. White, F. Bour, J. Yallop, Modular implicits, in: ML'14: ACM SIGPLAN ML Family Workshop 2014, Vol. 198 of Electronic Proceedings in Theoretical Computer Science, 2015, pp. 22–63. doi:10.4204/EPTCS.198.2.

[29] J. Furuse, Typeful PPX and Value Implicits, in: OCaml 2015: The OCaml Users and Developers Workshop, 2015, available at https://bitbucket.org/camlspotter/ppx_implicits.

[30] Y. Minsky, OCaml for the Masses, Commun. ACM 54 (11) (2011) 53–58. doi:10.1145/2018396.2018413.

[31] M. Neubauer, P. Thiemann, An Implementation of Session Types, in: Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings, 2004, pp. 56–70. doi:10.1007/978-3-540-24836-1_5.

[32] J. Polakow, Embedding a Full Linear Lambda Calculus in Haskell, in: Haskell '15: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell, ACM, 2015, pp. 177–188. doi:10.1145/2804302.2804309.

[33] M. P. Jones, Type Classes with Functional Dependencies, in: Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences

on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings, 2000, pp. 230–244. doi:10.1007/3-540-46425-5_15.

[34] D. Orchard, N. Yoshida, Session types with linearity in Haskell, in: S. J. Gay, A. Ravara (Eds.), Behavioural Types: from Theory to Tools, River Publishers, 2017, pp. 219–241. doi:10.13052/rp-9788793519817.

[35] P. Thiemann, V. T. Vasconcelos, Context-Free Session Types, in: ICFP '16: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, 2016, pp. 462–475. doi:10.1145/2951913.2951926.

[36] L. Padovani, Context-Free Session Type Inference, in: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, 2017, pp. 804–830. doi:10.1007/978-3-662-54434-1_30.

[37] H. C. Melgratti, L. Padovani, Chaperone contracts for higher-order sessions, PACMPL 1 (ICFP) (2017) 35:1–35:29. doi:10.1145/3110279.

[38] O. Dardha, E. Giachino, D. Sangiorgi, Session Types Revisited, in: PPDP '12: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, ACM, New York, NY, USA, 2012, pp. 139–150. doi:10.1145/2370776.2370794.

[39] J. Vouillon, Lwt: a cooperative thread library, in: Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008, 2008, pp. 3–12. doi:10.1145/1411304.1411307.

[40] Jane Street Developers, Core library documentation, available at https://ocaml.janestreet.com/ocaml-core/latest/doc/core/ (2016).

[41] M. Charalambides, P. Dinges, G. A. Agha, Parameterized, Concurrent Session Types for Asynchronous Multi-Actor Interactions, Science of Computer Programming 115-116 (2016) 100–126. doi:10.1016/j.scico.2015.10.006.

[42] N. Ng, J. G. de Figueiredo Coutinho, N. Yoshida, Protocols by Default - Safe MPI Code Generation Based on Session Types, in: Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, 2015, pp. 212–232. doi:10.1007/978-3-662-46663-6_11.

[43] K. Honda, N. Yoshida, M. Carbone, Multiparty Asynchronous Session Types, J. ACM 63 (1) (2016) 9:1–9:67. doi:10.1145/2827695.

[44] Scribble Project homepage, www.scribble.org.

Appendix A. Further application: a database server

As a more practical application with the delegation, we show an implementation of a database server with a worker thread pool of queries for better responses. The main thread authenticates client connections to server and delegates a session established to one of the worker threads.

The database server in Listing 21 behaves as follows.

1. A client starts a session with the server on `db_ch` and sends the authentication information (`cred: credential`) (line 8).

2. The main thread accepts or rejects the client according to the authentication result. If it rejects, the session ends (lines 10-11).

3. The main thread connects to a worker thread waiting for the connection on `worker_ch` and delegates the session to the thread (line 15).

4. The worker thread either terminates the session (line 25) or receives a query (`query: query`) (line 27). The worker thread returns the query result (`res: result`) and repeats the loop (lines 29-30).

5. A client first communicates with the main thread, and once the connection is authorised, a client starts communicating with the delegated worker thread.

The protocol type of the database channel (`db_ch`) is shown in Listing 22. Lines 2-5 are for receiving credential and branching between the label `left` for

Listing 21 The implementation of a database server

```
1  let db_ch = new_channel ()
2  and worker_ch = new_channel ()
3
4  let rec main () =
5    accept db_ch ~bindto:_0 >>
6    recv _0 >>= fun cred ->
7    if bad_credential cred then
8      select_left _0 >>
9      close _0
10   else
11     select_right _0 >>
12     connect worker_ch ~bindto:_1 >>
13     deleg_send _1 ~release:_0 >>
14     close _1 >>=
15     main
16
17 let rec worker () =
18   accept worker_ch ~bindto:_0 >>
19   deleg_recv _0 ~bindto:_1 >>
20   close _0 >>
21   let rec loop () =
22     branch
23       ~left:(_1, fun () -> close _1)
24       ~right:(_1, fun () ->
25           recv _1 >>= fun query ->
26           let res = do_query query in
27           send _1 res >>=
28           loop)
29   in loop () >>= worker
```

rejection and the label `right` for acceptance. Lines 7-10 correspond to `loop` in the Listing 21 (lines 23-31). The `left` branch finishes the protocol, while the `right` branch is for communicating the repetition of a query and its result.

Delegation appears in the type of `worker_ch` as follows.

```
  val worker_ch :
    [`deleg of req * (dbprotocol, serv) sess * [`close]]
```

where the delegated session has the polarity `serv`, because the delegated session is originally established by `accept`.

Listing 22 The protocol of a database server

```
type dbprotocol =
  [`msg of req * credential *
    [`branch of resp *
      [ `left of [`close],
      | `right of query_loop ]]]
and query_loop =
  [`branch of req *
    [ `left of [`close]
    | `right of [`msg of req * query *
                  [`msg of resp * result * query_loop]]]]
```