



**HAL**  
open science

## Session-ocaml: A Session-Based Library with Polarities and Lenses

Keigo Imai, Nobuko Yoshida, Shoji Yuen

► **To cite this version:**

Keigo Imai, Nobuko Yoshida, Shoji Yuen. Session-ocaml: A Session-Based Library with Polarities and Lenses. 19th International Conference on Coordination Languages and Models (COORDINATION), Jun 2017, Neuchâtel, Switzerland. pp.99-118, 10.1007/978-3-319-59746-1\_6 . hal-01657342

**HAL Id: hal-01657342**

**<https://inria.hal.science/hal-01657342v1>**

Submitted on 6 Dec 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

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

Keigo Imai<sup>1</sup>, Nobuko Yoshida<sup>2</sup>, and Shoji Yuen<sup>3</sup>

<sup>1</sup> Gifu University, Japan

<sup>2</sup> Imperial College London, UK

<sup>3</sup> 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 and delegations. We show applications of `session-ocaml` including a travel agency usecase and an SMTP protocol.

## 1 Introduction

Session types [5], from their origins in the  $\pi$ -calculus [17], serve as rigorous specifications for coordinating *link mobility* in the sense that a communication link can move among participants, while ensuring type safety. In session type systems such mobility is called *delegation*. Once the ownership of a session is delegated (transferred) to another participant, it cannot be used anymore at the sender side. This property is called *linearity* of sessions and appears indispensable for all session type systems.

Linearity of session channels, however, is a major obstacle to adopt session type disciplines in mainstream languages, as it requires special syntax extensions for session communications [9], or depends on specific language features, such as type-level functions in Haskell [11, 16, 20, 26], and affine types in Rust [13], or even falling back on run-time and dynamic checking [7, 8, 22, 27]. For instance, a common way in Haskell implementations is to track linear channels using an extra *symbol table* which denotes types of each resource conveyed by a *parameterised monad*. A Haskell type for a session-typed function is roughly of the form:

$$t_1 \rightarrow \cdots \rightarrow \mathbf{M} \{c_1 \mapsto s_1, c_2 \mapsto s_2, \dots\} \{c_1 \mapsto s'_1, c_2 \mapsto s'_2, \dots\} \alpha$$

where  $\mathbf{M}$  is a monad type constructor of arity three,  $\alpha$  is a result type and the two  $\{\dots\}$  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 a channel. Since this static encoding is Haskell-specific using type-level functions, it is not directly extendable to other languages.

This paper proposes the `session-ocaml` library, which provides a fully static implementation of session types in OCaml without any extra mechanisms or tools (i.e. sessions are checked at compile-time). We extend the technique posted to the OCaml mailing list by Garrigue [4] where linear usage of resources is enforced solely by the parametric polymorphism mechanism. According to [4], the type of a *file handle* guarantees linear access to *multiple resources* using a symbol table in a monad-like structures. Adapting 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 a *slot monad* and the file handles to *lenses*. The slot monad is based on a type  $(p, q, a)$  monad (hereafter we use postfix type constructor of OCaml) where  $p$  and  $q$  are called *slots* which act like a symbol table. Slots are represented as a sequence of types represented by nested pair types  $s_1 * (s_2 * \dots)$ . Lenses [15] are combinators that provide access to a particular element in nested tuples and are used to manipulate a symbol table in the slot monad. These mechanisms can provide an *idiomatic way* (i.e. code does 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 (unlike FuSe [22] which combines static and dynamic checking for linearity, see § 5).

To enable *session-type inference* solely by unification in OCaml, `session-ocaml` is equipped with *polarised session types* which give an alternative formulation of *duality* (binary relation over types which ensures reciprocal use of sessions). In a polarised session type  $(p, q)$  *sess*, the *polarity*  $q$  is either `serv` (server) or `cli` (client). The usage of a session is prescribed in the *protocol type*  $p$  which provides an *objective* view of a communication based on a *communication direction* of `req` (request; client to server) and `resp` (response; server to client). For example, the protocol type for sending of a message type `'v` from client to server is `[`msg of req * 'v * 's]` and the opposite is `[`msg of resp * 'v * 's]`. Duality is not necessary for protocol types as it shows a protocol common to both ends of a session rather than biased by either end. Then the session type inference can be driven solely by type unification which checks whether a protocol matches its counterpart or not. For instance, the dual of  $(p, \text{cli})$  *sess* is  $(p, \text{serv})$  *sess* and vice versa. When a session is being initiated, polarities are assigned to each end of a session according to the primitives used, namely `cli` for the proactive peer and `serv` for the passive peer. The protocol types also provide a usual prefixing declaration of session types, which is more human-readable than FuSe types [22] (see § 5).

The rest of the paper is as follows. Section 2 outlines programming with `session-ocaml`. Section 3 shows the library design with the polarised session type and the slot monads. In Section 4, we present two examples, a travel agency usecase and SMTP protocol implementations. Section 5 discusses comparisons with session type implementations in functional languages. Section 6 concludes and discusses further application of our technique. Technical report [10] includes the implementation of `session-ocaml` modules and additional examples. `Session-ocaml` is available at <https://github.com/keigo/session-ocaml>.

**Listing 1** The xor server and its client

---

```

1 open Session0
2 let xor_ch = new_channel ();;
3 Thread.create (fun () ->
4   accept_ xor_ch (fun () ->
5     let%> x,y = recv () in
6     send (xor x y) >>
7     close ())) ();;
8 connect_ xor_ch (fun () ->
9   send (false,true) >>
10  let%> b = recv () in
11  print_bool b;
12  close ()) ()

```

---

**2 Programming with session-ocaml**

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

**Send and receive primitives** Listing 1 shows a server and client which communicate boolean values. The module `Session0`<sup>4</sup> introduces functions of `session-ocaml` in the scope. `xor_ch` (line 2) is a *service channel* (or *shared channel*) that is used to start communication by a client connecting (`connect_`) to the server waiting (`accept_`) at it.<sup>5</sup> The server (lines 3-7) receives (`recv`) a pair of booleans, then calculates the exclusive-or of these values, transmits (`send`) back the resulting boolean, and finishes the session (`close`). These communication primitives communicate on an implicit *session endpoint* (or *session channel*) which is connected to the other endpoint. For inferring session types by OCaml, communication primitives are concatenated by the *bind* operations `>>` and `>>=` of a parameterised monad [1] which conveys session endpoints. The syntax `let%> pat = e1 in e2` binds the value returned by `e1` to the pattern `pat` and executes `e2`, which is shorthand for `e1 >>= fun pat -> e2` (the `%` symbol indicates a syntax extension point in an OCaml program). The client (lines 8-12) sends a pair of boolean, receives from the server and finishes the session, as prescribed in the following type. These server and client behaviours are captured by the *protocol type* argument of the channel type inferred at `xor_ch` as follows:

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

The protocol type is the primary language of communication specification in `session-ocaml`. Here, `[`msg of r * v * p]` is a protocol that represents communication of a message of type `v` before continuing to `p`. `r ∈ {req,resp}` indicates a *communication direction* from client to server and vice versa, respectively. `[`close]` is the end of a session. Thus the above type indicates that by a session established at `xor_ch`, (1) the server receives a request of type `bool * bool` and then (2) sends a response of type `bool` back to the client.

**Branching and recursion** A combination of branching and recursion provides various useful idioms such as exception handling. As an example, Listing 2 shows a logical operation server. The protocol type inferred for `log_ch` is:

<sup>4</sup> The suffix `0` means that it only uses the slot 0 (see later in this section).

<sup>5</sup> The suffixed underscore means that they run immediately instead of returning a monadic action (see later).

**Listing 2** A logical operation server

---

```

1 open Session0
2 type binop = And | Or | Xor | Imp
3 let log_ch = new_channel ()
4 let eval_op = function
5   | And -> (&& | Or -> (||)
6   | Xor -> xor
7   | Imp -> (fun a b -> not a || b)
8 let rec logic_server () =
9   match%branch0 () with
10  | `bin -> let%$ op = recv () in
11            let%$ x,y = recv () in
12            send (eval_op op x y) >>=
13            logic_server
14  | `fin -> close ();;
15
16 Thread.create
17   (accept_log_ch logic_server) ();;
18 connect_log_ch (fun () ->
19   [%select0 `bin] >>
20   send And >>
21   send (true, false) >>
22   let%$ ans = recv () in
23   (print_bool ans;
24   [%select0 `fin] >>
25   close ())) ()

```

---

```

[ `branch of req * [ `bin of [ `msg of req * binop *
  [ `msg of req * (bool * bool) * [ `msg of resp * bool * 'a]]]
  | `fin of [ `close]]] as 'a

```

`[ `branch of  $r$  * [... |  $lab_i$  of  $p_i$  | ...]]` represents a protocol that branches to  $p_i$  when label  $lab_i$  is communicated. Here  $r$  is a communication direction.  $t$  as 'a is an equi-recursive type [24] of OCaml that represents recursive structure of a session where 'a in  $t$  is instantiated by  $t$  as 'a. Lines 8-14 describe the body of the server. It receives one of the labels `bin` or `fin`, and branches to a different protocol. `match%branch0 () with | ... |  $lab_i$  ->  $e_i$  | ...` is the syntax for branching to the expression  $e_i$  after label  $lab_i$  is received. Upon receipt of `bin`, the server receives requests for a logical operation from the client (type `binop` and `bool * bool`), sends back a response and returns to the branch (note that the server is recursively defined by `let rec`). In the case of `fin`, the session is terminated. `[%select0 `lab]` is a syntax to select one of branches with a label  $lab$ .<sup>6</sup> A client using selection is shown in lines 18-25: it selects the label `bin`, requests conjunction, and selects `fin`; then the session ends.

For the branching primitive on arbitrary labels, `session-ocaml` uses OCaml polymorphic variants and syntax extensions. By using equi-recursive types, recursive protocols are also directly encoded into OCaml types.

**Link mobility with delegation** Link mobility with *session delegation* enables one to describe a protocol where the communication counterpart dynamically changes during a session. A typical pattern utilising delegation incorporates a main thread accepting a connection and worker threads doing the actual work to increase responsiveness of a service.

In `session-ocaml`, a program using delegation handles *multiple sessions* simultaneously. We explicitly assign each session endpoint to a *slot* using *slot specifiers* `_0, _1, ...` which gives an idiomatic way to use linear channels. Listing 3 shows an example of a highly responsive server using delegation. The server receives

<sup>6</sup> Here the bracket is another form of a syntax extension point applied to an expression (see the OCaml manual).

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

---

```

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

```

---

repeated connection requests on channel `log_ch`, consisting of the main thread and six worker threads. The module `SessionN` provides slot specifiers and accompanying communication primitives, where the suffix `N` means that it can handle on arbitrary number of sessions. The main thread (lines 3-7) accepts a connection from a client (`accept`) with `log_ch` and assigns the established session to slot 0 (`~bindto:_0`).<sup>7</sup> Next, it connects (`connect`) to a worker waiting for delegation at channel `worker_ch` (line 2) and assigns the session to slot 1 (`~bindto:_1`). Finally it delegates the session with the client to the worker (`deleg_send`), then ends the session with the worker and accepts the next connection. The worker thread (lines 8-12) receives the delegated session from the main thread (`deleg_recv`) and assigns the session to slot 0, then continues to `logic_server` (Listing 2). Here, `Session0` module used by `logic_server` implicitly allocates the session type to slot 0, hence can be used with `SessionN` module. Line 14 starts the main thread and workers. Here `run` is a function that executes `session-ocaml` threads.

The protocol type of `worker_ch` is inferred as follows:

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

Here `logic_p` is the protocol type of `log_ch` and `[`deleg of r * s * p]` is the delegation type. `r` is a communication direction, `s` is a polarised session type (a type with protocol and polarity which we explain next) for the delegated session and `p` is a continuation. By inferring the protocol types, `session-ocaml` can statically guarantee safety of higher-order protocols including delegations.

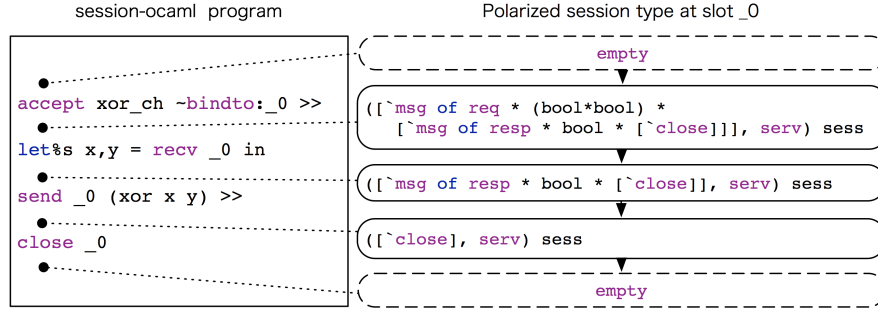
**The polarised session types** Communication safety is checked by matching each protocol type inferred at both ends. The *polarised session type*  $(p, q)$  `sess` given to each endpoint plays a key role for protocol type inference. Here  $p$  is a protocol type, and  $q \in \{\text{serv}, \text{cli}\}$  is the *polarity* determined at session initiation. `serv` is assigned to the `accept` side and `cli` to the `connect` side. `serv` and `cli` are *dual* to each other.

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 correspondences between polarities and communication directions.

<sup>7</sup> `~arg:e` is a *labelled argument*  $e$  for a named parameter `arg`.

**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

**Fig. 1.** Session type changes in `xor_server`

To track the entire session, a polarised session type changes in its protocol part as a session progresses. Fig. 1 shows changes of the session type in slot 0 of the xor server (here we use the `SessionN` module). The server first `accepts` a connection and assigns the session type to slot 0, where the type before acceptance is `empty`. After the subsequent reception of the pair of booleans and transmission of the xor values of those booleans, `req` and `resp` are consumed, and becomes `empty` again at the end of the session. Similar type changes occur on both `main` and `worker` and their types would be:

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

Here the type  $(s, t, a)$  session specifies that it expects slot sequence  $s$  at the beginning, and returns another slot sequence  $t$  and a value of type  $a$ . The type `empty * (empty * 'ss)` denotes that slot 0 and 1 are empty at the beginning, and since they never return the answer (i.e. the recursion runs infinitely), the rest of types `'tt` and `'a` are left as variables.

The type of `logic_server` in Listing 2 has a session type:

```
((logic_p, serv) sess * 'ss, empty * 'ss, unit) session
```

Here `logic_server` expects a session assigned at slot 0 before it is called, hence it expects the session type  $(\text{logic\_p}, \text{serv})\text{sess}$  in its pre-type. A difference from `main` and `worker` above is that since each of them establishes or receives sessions by their own (by using `accept`, `connect` or `deleg_recv`), they expect that slots 0 and 1 are `empty`.

Table 2 shows the type and communication behaviour before and after the execution of each `session-ocaml` communication primitive. Each row has a *pre-type* (the type required before execution) and *post-type* (the type guaranteed after execution). The protocol type at `serv` is obtained by replacing `req` with `resp` and `resp` with `req`. For example, the session `send _n e` has pre-type  $([\text{`msg of req} * v * p], \text{cli}) \text{sess}$  at `cli` and  $([\text{`msg of resp} * v * p], \text{serv}) \text{sess}$  at `serv` where  $_n$  is a slot specifier,  $e$  is an expression,  $v$  is a value type and  $p$  is a protocol type.

**Table 2.** session-ocaml primitives and protocol types

Primitive	Pre-type (at <code>cli</code> <sup>*1</sup> )	Post-type	Synopsis
<code>send _n e</code>	<code>[`msg of req * v * p]</code>	<code>p</code>	sending $e : v$ at slot $n$
<code>let%<i>s</i> pat = recv _n</code> <code>in ...</code>	<code>[`msg of resp * v * p]</code>	<code>p</code>	Reception at slot $n$ , binding to pattern $pat : v$
<code>[%select _n `lab<sub><i>i</i></sub>]</code>	<code>[`branch of req * [&gt; `lab<sub><i>i</i></sub> of p<sub><i>i</i></sub>]]</code>	<code>p<sub><i>i</i></sub></code>	Select label $lab_i$ at slot $n$
<code>match%branch _n with</code> <code>  `lab<sub>0</sub> -&gt; e<sub>0</sub></code> <code>  ...</code> <code>  `lab<sub><i>m</i></sub> -&gt; e<sub><i>m</i></sub></code>	<code>[`branch of resp * [ `lab<sub>0</sub> of p<sub>0</sub>   ...   `lab<sub><i>m</i></sub> of p<sub><i>m</i></sub>]]</code>	<code>t</code>	Branch at $n$ with labels $lab_i$ (protocol type $p_i$ is that of pre-type of $e_i$ ; $t$ is a post-type of all $e_i$ )
<code>deleg_send _n</code> <code>~release:_m</code>	<code>n:[`deleg of req * s * p]<sup>*2</sup></code> <code>m:s<sup>*2</sup></code>	<code>n:p</code> <code>m:empty<sup>*3</sup></code>	Delegate session at $m$ with type $s$ along $n$
<code>deleg_recv _n</code> <code>~bindto:_m</code>	<code>n:[`deleg of resp * s * p]<sup>*2</sup></code> <code>m:empty<sup>*3</sup></code>	<code>n:p</code> <code>m:s<sup>*2</sup></code>	Reception of delegation along $n$ and assign it to $m$
<code>close _n</code>	<code>[`close]</code>	<code>empty<sup>*3</sup></code>	Close session at slot $n$

$s$  is a polarised session type,  $_n$  and  $_m$  are slot specifiers,  $e$  is an expression of a base type,  $ch$  is a service channel,  $`lab$  is a polymorphic variant and  $pat$  is a binding pattern.

\*1: At `serv`, `req` and `resp` are exchanged; \*2:  $s$  is a session type (not a protocol type);  
\*3: Slot type changes to `empty`.

Primitive	Pre-type	Post-type	Synopsis
<code>accept ch ~bindto:_n</code>	<code>empty</code>	<code>(p, serv) sess</code>	Accept a connection at channel $ch$ ; assign a new session of polarity <code>serv</code> to $n$
<code>connect ch ~bindto:_n</code>	<code>empty</code>	<code>(p, cli) sess</code>	Connect to channel $ch$ ; assign a new ses- sion of polarity <code>cli</code> to $n$

Selection `[%select _n]` has *open* polymorphic variant type `[>...]` in pre-type to simulate *subtyping* of the labelled branches.

### 3 Design and implementation of session-ocaml

In this section, we first show the design of polarised session types associated with communication primitives (§ 3.1); then introduce the *slot monad* which conveys multiple session endpoints in a sequence of slots and constructs the whole session type for a session (§ 3.2). In § 3.3, we introduce the *slot specifier* to look up a particular slot in a slot sequence with *lenses* which are 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. This section mainly explains the *type signatures* of the communication primitives. Implementation of the communication *behaviours* is left to [10].



### 3.1 Polarity polymorphism

The *polarity polymorphism* is accompanied within all session primitives in that the appropriate direction type is assigned according to the polarity. This resolves a trade-off of having two polarised session types for one transmission. For instance, a transmission of a value could have two candidates, [`msg of req * 'v * 's`] and [`msg of resp * 'v * 's`] but they are chosen according to the polarity from which the message is sent. In order to relate the polarities to the directions, `cli` and `serv` are defined by type aliases as follows:

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

For each communication primitive, we introduce fresh type variables  $r_{req}$  and  $r_{resp}$  representing the communication direction, and put  $r_{req} * r_{resp}$  as the polarity in its session type. When its polarity is `cli`, we put  $r_{req}$  for `req` and  $r_{resp}$  for `resp`, while when it is `serv`, we put  $r_{req}$  for `resp` and  $r_{resp}$  for `req`. For example, the pre-type of `send` is (`[msg of 'r1*'v*'p]`, `'r1*'r2`) `sess` and that of `recv` is (`[msg of 'r2*'v*'p]`, `'r1*'r2`) `sess`. The same discipline applies to branching and delegation. The actual typing is deferred to the following subsections.

### 3.2 The slot monad carrying multiple sessions

The key factor to achieve linearity is to keep session endpoints securely inside a monad. In `session-ocaml`, multiple sessions are conveyed in slots using the *slot monad* of type

```
(s0 * (s1 * ...), t0 * (t1 * ...), α) session
```

which denotes a computation of a value of type  $\alpha$ , turning each pre-type  $s_i$  of slot  $i$  to post-type  $t_i$ . We refer to slots before and after computation as *pre*- and *post*-slots, respectively. The type signature of the slot monad is shown in Listing 4. The operators `>>=` and `>>` (lines 3-4) compose computation sequentially while propagating type changes on each slot by demanding the same type 'q in the post-slots on the left-hand side and the pre-slots on the right-hand side. Usually they construct compound session types via *unification*. For example, in `send` `And >> send (true, false)` (from Listing 2) the left hand side (`send` `Add`) has the following type:

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

While the type of the right hand side (`send (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 monad with the pre-type in the following monad (and the rest of slots 'ss1 with 'ss2), the bind operation produces a chain of protocol type in the pre-slots as follows:

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

In line 5, `run` executes the slot monad and requires all slots being `empty` before and after execution, thus it precludes use of unallocated slots, and mandates that all sessions are finally closed (which corresponds to the absence of *contraction*

---

**Listing 4** The slot monad
 

---

```

1 type ('p,'q,'a) session and empty and all_empty = empty * 'a as 'a
2 val return : 'a -> ('p,'p,'a) session
3 val (>>=) : ('p,'q,'a) session -> ('a -> ('q,'r,'b) session) -> ('p,'r,'b) session
4 val (>>) : ('p,'q,'a) session -> ('q,'r,'b) session -> ('p,'r,'b) session
5 val run : (all_empty,all_empty,unit) session -> unit
    
```

---

**Table 3.** Types for slot specifiers

Specifier	Type
<code>_0</code>	<code>('a, 'b, 'a * 'ss, 'b * 'ss) slot</code>
<code>_1</code>	<code>('a, 'b, 's0 * ('a * 'ss), 's0 * ('b * 'ss)) slot</code>
<code>_2</code>	<code>('a, 'b, 's0 * ('s1 * ('a * 'ss)), 's0 * ('s1 * ('b * 'ss))) slot</code>
<code>_n</code>	<code>('a, 'b, 's0*(...*( 's_{n-1}*( 'a*'ss))...), 's0*(...*( 's_{n-1}*( 'b*'ss))...)) slot</code>

in linear type systems). The type `all_empty` (line 1) is a type alias for OCaml equi-recursive type `empty * 'a as 'a`,<sup>8</sup> enabling use of *arbitrarily* many slots.

### 3.3 Lenses focusing on linear channels

In order to provide access to session endpoints conveyed inside a slot monad, we apply *lenses* [15] to slot specifiers `_0, _1, ...` which are combinators to manipulate a polymorphic data structure. The following shows the type of a slot specifier which modifies slot  $n$  of a slot sequence:

```
type ('a, 'b, 's0*(...('s_{n-1}*( 'a*'ss))...), 's0*(...('s_{n-1}*( 'b*'ss))...)) slot
```

The type says that it replaces the type `'a` of slot  $n$  in the slot sequence `'s0 * (...('s_{n-1}*( 'a*'ss))...)` with `'b` and the resulting sequence type becomes to `'s0*(...('s_{n-1}*( 'b*'ss))...)`. The type of each slot specifier (`_0, _1, ...`) is shown in Table 3.

Listing 5 exhibits type signatures of `accept`, `connect`, `close`, `send`, `recv`, `deleg_send` and `deleg_recv` which are compiled from lenses, the polarised session types (§ 3.1), slot monads (§ 3.2), and pre- and post-types in Table 2 (§ 2). Note that `bindto`: and `release`: are named parameters of a primitive.

`accept` and `connect` (lines 1-4) assign a new session channel to the `empty` slot, whereas `close` (lines 5-6) finishes the session and leave the slot `empty` again. `send` and `recv` (lines 7-10) proceed the protocol type by removing a `msg` prefix.

`deleg_send` and `deleg_recv` (lines 11-16) update a pair of slots; one is for the transmission/reception and the other is for the delegated session. To update the slots twice, they take a pair of slot specifiers which share an intermediate slot sequence `'mid`. They embody an aspect of linearity: `deleg_send` releases the ownership of the delegated session by replacing the slot type to `empty`, while `deleg_recv` allocates another `empty` slot to the acquired session.

---

<sup>8</sup> In order to have such a type, we compile the code with the `-rectypes` option. If we chose 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
8   -> 'v -> ('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*'s*'p], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'mid) slot
13   -> release:('s, empty, 'mid, 'post) slot -> ('pre, 'post, 'v) session
14 val deleg_recv :
15   ((['deleg of 'r2*'s*'p], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'mid) slot
16   -> bindto:(empty, 's, 'mid, 'post) slot -> ('pre, 'post, 'v) session
17 val select_left : ((['branch of 'r1 * [>`left of 's1]], 'r1*'r2) sess,
18   ('s1, 'r1*'r2) sess, 'pre, 'post) slot -> ('pre, 'post, unit) session
19 val select_right : ((['branch of 'r1 * [>`right of 's2]], 'r1*'r2) sess,
20   ('s2, 'r1*'r2) sess, 'pre, 'post) slot -> ('pre, 'post, unit) session
21 val branch2 : ((['branch of 'r2 * [`left of 's1 | `right of 's2]], 'r1*'r2) sess,
22   ('s1, 'r1*'r2) sess, 'pre, 'mid1) slot * (unit -> ('mid1, 'post, 'a) session)
23 -> ((['branch of 'r2 * [`left of 's1 | `right of 's2]], 'r1*'r2) sess,
24   ('s2, 'r1*'r2) sess, 'pre, 'mid2) slot * (unit -> ('mid2, 'post, 'a) session)
25 -> ('pre, 'post, 'a) session

```

---

The primitives for binary selection `select_left`, `select_right` and branching `branch2` (lines 17-25) communicate left and right labels. `branch2` takes a pair of continuations as well as a pair of slot specifiers. According to the received label, one of the continuations is invoked after the pre-type of the invoked continuation is assigned to the corresponding slot.

Finally, 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 applying a slot specifier. In each type signature, the first and second type arguments of type `slot` prescribes *how* a 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 function application `close _1` is given by the following type substitution:

```

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

```

And the type completing the session at slot 1 is:

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

**A note on delegation and slot assignment** The delegation `[`deleg of r * s * p]` distinguishes polarity in the delegated session `s`. This results in a situation where 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. It 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 a `cli` endpoint while `accept` gives a `serv`. Due to the different polarities in the delegated session types, the types of `then` and `else` clause conflict with each other, even if they have the identical behaviour. In [32], where polarity is not a type but a syntactic construct, such a restriction does not exist. A similar restriction exists in GV [30] which has polarity in `end` (`end!` and `end?`).

In principle, it is possible to automatically assign numbers to slot specifiers *locally* in a function instead of writing them explicitly. However, since sequential composition of the session monad requires each post- and pre-type to match with each other, the *global* assignment of slot specifiers would require a considerable amount of work and can be hard to predict its behaviour. As shown in Listing 3, one can handle two sessions by just using two slot specifiers.

**Syntax extension for arbitrarily labelled branch** Since the OCaml type system does not allow to parameterise type labels (polymorphic variants), we provide macros for arbitrarily-labelled branching. Listing 6 provides helper functions for the macros. For selection, the macro `[%select _n `labi]` is expanded to `_select _n (fun x -> `labi(x))`, where the helper function `_select` transmits label `labi` on the slot `n`. `match%branch _n with | `lab1 -> e1 | ... | `labk -> ek` is expanded to:

```
_branch_start _n ((function | `lab1(p1),q -> _branch _n (p1,q) (e1) | ...
                    | `labk(pk),q -> _branch _n (pk,q) (ek)
                    : [ `labi of 'p1|...|`labk of 'pk]*'x->'y)
```

The helper functions `_branch_start` and `_branch` have the type shown in Listing 6. The anonymous function will have type

```
[ `lab1 of p1 | ... | `labk of pk ] * q -> (pre, post, v) session
```

where `q` is the polarity and `pi` is the protocol type in the pre-type at slot `n` in `ei`. When a label `labi` ( $i \in \{1 \dots k\}$ ) is received, `_branch_start _n f` passes a pair of *witness* ``labi(pi)` and `q` of a polarised session type `(pi, q) sess` to the function `f`. The anonymous function extracts the witness and by `_branch` it rebuilds the session type `('pi, 'q) sess` and passes the continuation `ei` as the pre-type. The type annotation `[ `labi of 'p1|...|`labk of 'pk]*'x->'y` erases the row type variable `[<...]` generated by the anonymous function. The annotation is necessary because the row type variable turns into a useless *monomorphic* row type variable `[_<...]` in the inferred protocol type. This may cause a problem while compiling since the compiler requires monomorphic type variables to not escape from compilation units.

**Listing 6** The helper functions for branching/selection with arbitrary labels

---

```

1 val _select :
2     (([ `branch of 'r2 * 'br], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'post) slot
3     -> ('p -> 'br) -> ('pre, 'post, unit) session
4 val _branch_start : (([ `branch of 'r1 * 'br], 'r1*'r2) sess, 'x, 'pre, 'dummy) slot
5     -> ('br * ('r1*'r2) -> ('pre, 'post, 'v) session) -> ('pre, 'post, 'v) session
6 val _branch :
7     (([ `branch of 'r1 * 'br], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'mid) slot
8     -> 'p * ('r1*'r2) -> ('mid, 'post, 'v) session -> ('pre, 'post, 'v) session

```

---

**Listing 7** Travel agency

---

```

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

```

---

## 4 Applications

### 4.1 Travel agency

We demonstrate programming in `session-ocaml` using the Travel agency scenario from [9], which consists of typical patterns found in business and financial protocols. The scenario is played by three participants: customer, agency and service (Listing 7). customer and service initially do not know each other, and agency mediates a deal between them by session delegation.

customer begins an order session with agency and binds it to their own slot 0 (each process has a separate slot sequence). Then customer requests and receives the price for the desired journey after sending the `quote` label. In our scenario, customer requests "London to Paris" and agency replies with a fixed price `80.0`.

Then customer might send the `agree` label to proceed the transaction with the current price. Or if customer does not agree with the price, customer can cancel the transaction by sending the `reject` label. Or, customer can send `quote` again and this will be repeated an arbitrary number of times for different journeys (we omit this branch from the code). 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 slot 1. Then it delegates to `service`, through slot 1, the interactions with `customer` remaining for slot 0. `customer` then sends the billing address (unaware that he/she is now talking to `service`), and `service` replies with the dispatch date (`now()`) for the purchased tickets. The transaction is complete.

The protocol type 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 polarity and communication directions (§ 3.1), hence the `service` can handle both polarities. It reflects the part after `agree` in the protocol above where `'r1` is `req` and `'r2` is `resp`. Thus delegation with the polarised session types and slots effectively gives a way to coordinate *higher order* communication incurred by link mobility.

Static checking of delegation makes it easier to find errors otherwise hard to analyse due to the indirect nature of delegation. Consider a case that `service` changes its behaviour to receive `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 ]
```

Whereas that of `agency` remains same as before, it results in a type error at the moment when a service channel is passed. Without static typing, the run-time error would be deferred until the beginning of actual `client-service` communication.

## 4.2 An SMTP protocol

This section shows an SMTP client implementation by `session-ocaml`. Listing 8 and Listing 9 show the protocol type of SMTP and message types representing SMTP commands and replies; and Listing 10 shows the client implementation. Line 2 in Listing 10 generates a service channel for connecting to the SMTP server. Here `smtp_adapter` is an *adapter* that converts a sequence of session messages to a TCP stream. Its definition is shown in Listing 11 and built using the combinators shown in Listing 12. The functions `req` and `resp` accept a function to convert between a message of type `'v` and a command string and construct an adapter. `bra` and `sel` are branching and selection respectively, and `cls` is the end of the session. The function with the same name as the message type is a function for converting to a string (or vice versa) and is responsible for actual stream processing. In OCaml, `f @@ g` means function composition `fun x -> f (g x)`, and `begin e end` means (*e*). Since OCaml evaluates eagerly, each function is  $\eta$ -expanded with a parameter `ch` so that it does not recurse infinitely.

---

**Listing 8** The protocol type of SMTP

---

```

1 type smtp =
2   [ `msg of resp * r200 * [ `msg of req * ehlo * [ `msg of resp * r200 * mail_loop]]
3 and mail_loop =
4   [ `branch of req *
5     [ `left of [ `msg of req * mail * [ `msg of resp * r200 * rcpt_loop]]
6     | `right of [ `msg of req * quit * [ `close]]]]]
7 and rcpt_loop =
8   [ `branch of req *
9     [ `left of [ `msg of req * rcpt *
10      [ `branch of resp * [ `left of [ `msg of resp * r200 * rcpt_loop]
11      | `right of [ `msg of resp * r500 * [ `msg of req * quit * [ `close]]]]]]]
12     | `right of body]]]
13 and body =
14   [ `msg of req * data * [ `msg of resp * r354 * [ `msg of req * string list *
15     [ `msg of resp * r200 * mail_loop]]]]]

```

---



---

**Listing 9** Types for SMTP commands and replies

---

```

1 (* EHL0 example.com *)           (* MAIL FROM: alice@example.com *)
2 type ehlo = EHL0 of string       type mail = MAIL of string
3 (* RCPT TO: bob@example.com *)   (* DATA *)
4 type rcpt = RCPT of string       type data = DATA
5 (* QUIT *)                       (* Success e.g. 250 0k *)
6 type quit = QUIT                 type r200 = R200 of string list
7 (* Error e.g. 554 Relay denied *) (* 354 Start mail input *)
8 type r500 = R500 of string list   type r354 = R354 of string list

```

---

The adapter for branch `bra` is asymmetric in its parameters [18]. `bra` has a parser of type `string -> 'v` option on the left side since the adapter determines a continuation in a branch according to the parsed result of a received string. The adapter chooses `left` if the parser succeeds (returns `Some(x)`), and `right` if it fails (`None`). By nesting `bra`, any nesting of branch can be constructed.

Comparing to the existing Haskell implementation in [11], an advantage is that our OCaml version enjoys equi-recursive session types, so we avoid the manual annotation of repeated `unwind` operations needed to unfold iso-recursive types in Haskell. A shortcoming of the OCaml version is the explicit nature of adapter. However, since the adapter and the protocol type have the same structure, it can be generated semi-automatically from the type declaration in Listing 8 when OCaml gains ad hoc polymorphism such as type classes. We expect this to be possible with modular-implicits [31], which will be introduced in a future version of OCaml. On the other hand, it is also possible to omit the protocol type declaration in Listing 8 by inferring the type of the adapter.

## 5 Related work

We discuss related work focusing on the functional programming languages. For other related work, see § 1.

**Listing 10** An implementation of SMTP client

---

```

1 open Session0
2 let ch = TcpSession.new_channel smtp_adapter "smtp.example.com:25"
3 let smtp_client () = connect_ ch begin fun () -> let%R200 s = recv () in
4   send (EHLO("me.example.com")) >> let%R200 _ = recv () in
5   select_left () >> (* enter into the main loop *)
6   send (MAIL("alice@example.com")) >> let%R200 _ = recv () in
7   select_left () >> (* enter into recipient loop *)
8   send (RCPT("bob@example.com")) >>
9   branch2 (fun () -> let%R200 _ = recv () in (* recipient Ok *)
10    select_right () >> (* proceed to sending the mail body *)
11    send DATA >> let%R354 _ = recv () in
12    send (escape mailbody) >> let%R200 _ = recv () in
13    select_right () >> send QUIT >> close ())
14   (fun () -> let%R500 msg = recv () in (* a recipient is rejected *)
15    (List.iter print_endline msg; send QUIT) >> close ()) end ()

```

---

**Listing 11** The TCP adapter for a SMTP client

---

```

1 let rec smtp_adapter ch = (resp r200 @@ req ehlo @@ resp r200 @@ ml_p) ch
2 and ml_p ch = sel ~left:(req mail @@ resp r200 @@ rp_p) ~right:(req quit @@ cls) ch
3 and rp_p ch = sel ~left:(req rcpt @@ bra ~left:(r200, rp_p)
4   ~right:(resp r500 @@ req quit @@ cls))
5   ~right:bd_p ch
6 and bd_p ch = (req data @@ resp r354 @@ req string_list @@ resp r200 @@ ml_p) ch

```

---

**Implementations in Haskell** The first work done by Neubauer and Thiemann [18] implements the first-order single-channel session types with recursions. Using parameterised monads, Pucella and Tov [26] provide multiple sessions, but manual reordering of symbol tables is required. Imai et al. [11] extend [26] with delegation, handling multiple sessions in a user-friendly manner by using type-level functions. Orchard and Yoshida [20] use an embedding of effect systems in Haskell via graded monads based on a formal encoding of session-typed  $\pi$ -calculus into PCF with an effect system. Lindley and Morris [16] provide an embedding of the GV session-typed functional calculus [30] into Haskell, building on a linear  $\lambda$ -calculus embedding by Polakow [25]. Duality inference is mostly represented by a multi-parameter type class with functional dependencies [14]; For instance, `class Dual t t' | t -> t', t' -> t` declares that `t` can be inferred from its dual `t'` and vice versa. However, all of the above works depend on type-level features in Haskell, hence they are not directly applicable to other programming languages including OCaml. See [21] for a detailed survey. `session-ocaml` generalises the authors' previous work in Haskell [11] by replacing type-level functions with lenses, leading to wider applicability to other programming languages.

**Implementations in OCaml** Padovani [22] introduces FuSe, which implements multiple sessions with dynamic linearity checking and its single-session version with static checking in OCaml. Our `session-ocaml` achieves static typing for multiple sessions with delegation by introducing session manipulations based on



**Listing 12** Combinators for TCP adapters

---

```

1 type 'p net = raw_chan -> (('p, serv) sess * all_empty, all_empty, unit) monad
2 val req : ('v -> string) -> 'p net -> ['msg of req * 'v * 'p] net
3 val resp : (string -> 'v parse_result) -> 'p net -> ['msg of resp * 'v * 'p] net
4 val sel : left:'p1 net -> right:'p2 net ->
5     ['branch of req * ['left of 'p1|'right of 'p2]] net
6 val bra : left:((string -> 'v1 parse_result) * 'p1 net) -> right:'p2 net ->
7     ['branch of resp * ['left of ['msg of resp * 'v1 * 'p1] |'right of 'p2]] net
8 val cls : ['close] net

```

---

lenses; and provides an idiomatic way to declare branching with arbitrary labels; while FuSe combines static and dynamic approach to achieve them.

The following example shows that `session-ocaml` can avoid linearity violation, while FuSe dynamically checks it 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 by using the wildcard `_` at the end of the `loop`. In FuSe 0.7, `loop` is well-typed and terminates in error `InvalidEndpoint` at runtime. In `session-ocaml`, this error inherently does not occur since each endpoint is implicit inside the monad and indirectly accessed by lenses.

[22] gives a micro-benchmark which measures run-time performance between the static and dynamic versions of FuSe. Based on the benchmark, it is shown that the overhead incurred by dynamic checking is negligible when implemented with a fast communication library such as `Core` [12], and concludes that the static version of FuSe performs well enough in spite of numerous closure creations in a monad. The FuSe implementation has been recently extended to *context free session types* [29] by adding an *endpoint* attribute to session types [23].

On duality inference, a simple approach in OCaml is firstly introduced by Pucella and Tov [26]. The idea in [26] 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 [5]. FuSe [22] reduces its size by almost half using the encoding technique in [3] by modelling binary session types as a chain of linear channel types as follows. A session type in FuSe  $(\text{'a}, \text{'b}) \text{ t}$  prescribes input (`'a`) and output (`'b`) capabilities. A transmission and a reception of a value `'v` followed by a session  $(\text{'a}, \text{'b}) \text{ t}$  are represented as  $(\text{\_0}, \text{'v} * (\text{'a}, \text{'b}) \text{ t}) \text{ t}$  and  $(\text{'v} * (\text{'a}, \text{'b}) \text{ t}, \text{\_0}) \text{ t}$  respectively, where `\_0` means “no message”; then the dual of a session type is obtained by swapping the top pair of the type. A drawback of these FuSe notations is it becomes less readable when multiple nestings are present. For example, in a simplified variant of the logic operation server in Listing 2 with no recursion nor branch, 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’s 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. To recover the readability, FuSe supplies the translator *Rosetta* which compiles FuSe types into session type notation with the prefixing style and vice versa. Our polarised session types are directly represented in a *prefixing* manner with the slight restriction shown in § 3.3.

## 6 Conclusion

We have shown `session-ocaml`, a library for session-typed communication which supports multiple simultaneous sessions with delegation in OCaml. The contributions of this paper are summarised as follows. (1) Based on lenses and the slot monad, we achieved a fully static checking of session types by the OCaml type system without adding any substantial extension to the language. Previously, a few implementations were known for a single session [22, 26], but the one that allows statically-checked multiple sessions is new and shown to be useful. To the authors’ knowledge, this is the first implementation which combines lenses and a parameterised monad. (2) On top of (1), we proposed macros for arbitrarily labelled branches. The macros “patch up” only the branching and selection parts where linear variables are inevitably exposed due to limitation on polymorphic variants. (3) We proposed a session type inference framework solely based on the OCaml built-in type unification. Communication safety is guaranteed by checking equivalence of protocol types inferred at both ends with different polarities.

Type inference plays a key role in using lenses without the burden of writing any type annotations. Functional programming languages such as Standard ML, F# and Haskell have a nearly complete type inference, hence it is relatively easy to apply the method presented in this paper. On the other hand, languages such as Scala, Java and C# have a limited type inference system. However, by a recent extension with Lambda expressions in Java 8, lenses became available without type annotations in many cases (see a proof-of-concept at <https://github.com/keigo/slotjava>). The main difficulty for implementing session types is selection primitives since they require type annotations for non-selected branches. Development of such techniques is future work.

Our approach which uses slots for simultaneous multiple sessions resembles parameterised session types [2, 19], and it is smoothly extendable to the multiparty session type framework [6]. We plan to investigate code generations from Scribble [28] (a protocol description language for the multiparty session types) along the line of [7, 8] integrating with parameterised features [2, 19].

**Acknowledgments** We thank Raymond Hu and Dominic Orchard for their comments on an early version of the paper. 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. Atkey, R.: Parameterized Notions of Computation. *Journal of Functional Programming* 13(3-4), 355–376 (2009)
2. Charalambides, M., Dinges, P., Agha, G.A.: Parameterized, Concurrent Session Types for Asynchronous Multi-Actor Interactions. *Science of Computer Programming* 115-116, 100–126 (2016)
3. Dardha, O., Giachino, E., Sangiorgi, D.: Session Types Revisited. In: *PPDP '12: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*. pp. 139–150. ACM, New York, NY, USA (2012)
4. Garrigue, J.: A mailing-list post (2006), available at <https://groups.google.com/d/msg/fa.caml/GWWtHOP35dI/IsrOze-qVLwJ>
5. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: *ESOP '98: Proceedings of the 7th European Symposium on Programming*. *Lecture Notes in Computer Science*, vol. 1381, pp. 122–138. Springer (1998)
6. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: *POPL*. pp. 273–284. ACM (2008), a full version, *JACM*, Vol 63(1), No. 9, 67 pages, 2016
7. Hu, R., Yoshida, N.: Hybrid Session Verification through Endpoint API Generation. In: *FASE. LNCS*, vol. 9633. Springer (2016)
8. Hu, R., Yoshida, N.: Explicit Connection Actions in Multiparty Session Types. In: *FASE. LNCS*, Springer (2017)
9. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. pp. 516–541 (2008)
10. Imai, K., Yoshida, N., Yuen, S.: Session-ocaml: a session-based library with polarities and lenses. Tech. rep., Imperial College London (2017), to appear.
11. Imai, K., Yuen, S., Agusa, K.: Session Type Inference in Haskell. In: *Postproceedings of Thrid Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2010)*. vol. 69, pp. 74–91 (March 2010)
12. Jane Street Developers: Core library documentation (2016), available at <https://ocaml.janestreet.com/ocaml-core/latest/doc/core/>
13. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session Types for Rust. In: *WGP 2015: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. pp. 13–22. ACM (2015)
14. Jones, M.P.: Type Classes with Functional Dependencies. In: *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*. pp. 230–244. Springer (2000)
15. Kmett, E.: Lenses, Folds and Traversals (2012), available at <http://lens.github.io/>
16. Lindley, S., Morris, J.G.: Embedding Session Types in Haskell. In: *Haskell 2016: Proceedings of the 9th International Symposium on Haskell*. pp. 133–145. ACM (2016)
17. Milner, R.: *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press (1999)
18. Neubauer, M., Thiemann, P.: An Implementation of Session Types. In: *PADL'04 : Practical Aspects of Declarative Languages. Lecture Notes in Computer Science*, vol. 3057, pp. 56–70. Springer (2004)
19. Ng, N., Coutinho, J.G., Yoshida, N.: Protocols by Default: Safe MPI Code Generation based on Session Types. In: *CC'15*. pp. 212–232. LNCS, Springer (2015)

20. Orchard, D., Yoshida, N.: Effects as sessions, sessions as effects. In: POPL 2016: 43th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 568–581. ACM (2016)
21. Orchard, D., Yoshida, N.: Sessions types with linearity in Haskell. In: Gay, S.J., Ravara, A. (eds.) Behavioural Types: from Theory to Tools. River Publishers (2017)
22. Padovani, L.: A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming* 27, e4 (2016)
23. Padovani, L.: Context-Free Session Type Inference. In: ESOP 2017: 26th European Symposium on Programming. *Lecture Notes in Computer Science* (2017), to appear. Preliminary version available at <https://hal.archives-ouvertes.fr/hal-01385258/>
24. Pierce, B.C.: Recursive Types. In: *Types and Programming Languages*, chap. 20, pp. 267–280. MIT Press (2002)
25. Polakow, J.: Embedding a Full Linear Lambda Calculus in Haskell. In: Haskell '15: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. pp. 177–188. ACM (2015)
26. Pucella, R., Tov, J.A.: Haskell Session Types with (Almost) No Class. In: Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 25–36. ACM (2008)
27. Scalas, A., Yoshida, N.: Lightweight Session Programming in Scala. In: ECOOP 2016: 30th European Conference on Object-Oriented Programming. *LIPIcs*, vol. 56, pp. 21:1–21:28. Dagstuhl (2016)
28. Scribble Project homepage, [www.scribble.org](http://www.scribble.org)
29. Thiemann, P., Vasconcelos, V.T.: Context-Free Session Types. In: ICFP '16: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 462–475 (2016)
30. Wadler, P.: Propositions as sessions. In: ICFP '12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 273–286. ACM (2012)
31. White, L., Bour, F., Yallop, J.: Modular implicits. In: ML'14: ACM SIGPLAN ML Family Workshop 2014. *Electronic Proceedings in Theoretical Computer Science*, vol. 198, pp. 22–63 (2015)
32. Yoshida, N., Vasconcelos, V.T.: Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science* 171(4), 73–93 (2007)