11

An OCaml Implementation of Binary Sessions

Hernán Melgratti[1,2] **and Luca Padovani**[3]

[1]Departamento de Computación, Universidad de Buenos Aires, Argentina
[2]CONICET-Universidad de Buenos Aires, Instituto de Investigación en Ciencias de la Computación (ICC), Buenos Aires, Argentina
[3]Dipartimento di Informatica, Università di Torino, Italy

Abstract

In this chapter we describe `FuSe`, a simple `OCaml` module that implements binary sessions and enables a hybrid form of session type checking without resorting to external tools or extensions of the programming language. The approach combines static and dynamic checks: the former ones are performed at compile time and concern the structure of communication protocols; the latter ones are performed as the program executes and concern the linear usage of session endpoints. We recall the minimum amount of theoretical background for understanding the essential aspects of the approach (Section 11.1) and then describe the API of the `OCaml` module throughout a series of simple examples (Section 11.2). In the second half of the chapter we detail the implementation of the module (Section 11.3) and discuss a more complex and comprehensive example, also arguing about the effectiveness of the hybrid approach with respect to the early detection of protocol violations (Section 11.4). We conclude with a survey of closely related work (Section 11.5).

The source code of `FuSe`, which is partially described in this chapter and can be used to compile and run all the examples given therein, can be downloaded from the second author's home page.

11.1 An API for Sessions

We consider the following grammar of types and session types

$$
\begin{array}{rcl}
t, s & ::= & \texttt{bool} \mid \texttt{int} \mid \alpha \mid T \mid [\texttt{l}_i : t_i]_{i\in I} \mid \cdots \\
T, S & ::= & \texttt{end} \mid \texttt{!}t.T \mid \texttt{?}t.T \mid \&[\texttt{l}_i : T_i]_{i\in I} \mid \oplus[\texttt{l}_i : T_i]_{i\in I} \mid A \mid \overline{A}
\end{array}
$$

where types, ranged over by t and s, include basic types, type variables, session types, disjoint sums, and possibly other (unspecified) types. Session types, ranged over by T and S, comprise the usual constructs for denoting depleted session endpoints, input/output operations, branches and choices, as well as possibly dualized session type variables A, B, etc.

The *dual* of a session type T, written $\overline{T}$, is obtained as usual by swapping input and output operations and is defined by the following equations:

$$
\begin{array}{rclcrclcrcl}
\overline{\overline{A}} & = & A & \quad & \overline{(\texttt{?}t.T)} & = & \texttt{!}t.\overline{T} & \quad & \overline{\&[\texttt{l}_i : T_i]_{i\in I}} & = & \oplus[\texttt{l}_i : \overline{T_i}]_{i\in I} \\
\overline{\texttt{end}} & = & \texttt{end} & & \overline{(\texttt{!}t.T)} & = & \texttt{?}t.\overline{T} & & \overline{\oplus[\texttt{l}_i : T_i]_{i\in I}} & = & \&[\texttt{l}_i : \overline{T_i}]_{i\in I}
\end{array}
$$

Following Gay and Vasconcelos [4], our aim is to incorporate binary sessions into a (concurrent) functional language by implementing the API shown in Table 11.1. The `create` function creates a new session and returns a pair with its two peer endpoints with dual session types. The `close` function is used to signal the fact that a session is completed and no more communications are supposed to occur in it. The `send` and `receive` functions are used for sending and receiving a message, respectively: `send` sends a message of type α over an endpoint of type $\texttt{!}\alpha.A$ and returns the same endpoint with its type changed to A to reflect that the communication has occurred; `receive` waits for a message of type α from an endpoint of type $\texttt{?}\alpha.A$ and returns a pair with the message and the same endpoint with its type changed to A. The `branch` and `select` functions deal with sessions that may continue along different paths of interaction, each path being associated with a *label* $\texttt{l}_i$. Intuitively, `select` takes a label $\texttt{l}_k$ and an endpoint of type $\oplus[\texttt{l}_i : A_i]_{i\in I}$

Table 11.1 Application programming interface for binary sessions

`val create`	:	$\texttt{unit} \rightarrow A \times \overline{A}$
`val close`	:	$\texttt{end} \rightarrow \texttt{unit}$
`val send`	:	$\alpha \rightarrow \texttt{!}\alpha.A \rightarrow A$
`val receive`	:	$\texttt{?}\alpha.A \rightarrow \alpha \times A$
`val select`	:	$(\overline{A_k} \rightarrow [\texttt{l}_i : \overline{A_i}]_{i\in I}) \rightarrow \oplus[\texttt{l}_i : A_i]_{i\in I} \rightarrow A_k$
`val branch`	:	$\&[\texttt{l}_i : A_i]_{i\in I} \rightarrow [\texttt{l}_i : A_i]_{i\in I}$

where $k \in I$, sends the label over the endpoint and returns the endpoint with its type changed to A_k, which is the continuation corresponding to the selected label. The most convenient OCaml representation for labels is as functions that *inject* an endpoint (say, of type $\overline{A_k}$) into a disjoint sum $[\mathtt{l}_i : \overline{A_i}]_{i \in I}$ where $k \in I$. This explains the type of `select`'s first argument. Dually, `receive` waits for a label from an endpoint of type $\&[\mathtt{l}_i : A_i]_{i \in I}$ and returns the continuation endpoint injected into a disjoint union.

We note a few more differences between the API we implement in this chapter and the one described by Gay and Vasconcelos [4]. First of all, we use parametric polymorphism to give session primitives their most general type. Second, we have a single function `create` to initiate a new session instead of a pair of `accept`/`request` functions to synchronize a service and a client. Our choice is purely a matter of simplicity, the alternative API being realizable on top of the one we present (the API implemented in the FuSe distribution already provides for the `accept`/`request` functions, which we will see at work in Section 11.4). Finally, our communication primitives are *synchronous* in that output operations block until the corresponding receive is performed. Again, this choice allows us to provide the simplest implementation of these primitives solely using functions from the standard OCaml library. Asynchronous communication can be implemented by choosing a suitable communication framework.

11.2 A Few Simple Examples

Before looking at the implementation of the communication primitives, we illustrate the API at work on a series of simple examples. In doing so, we assume that the API is defined in a module named `Session`. The following code implements a client of an ‘‘echo’’ service, a service that waits for a message and bounces it back to the client.

```
let echo_client ep x =
  let ep = Session.send x ep in
  let res, ep = Session.receive ep in
  Session.close ep;
  res
```

The parameter `ep` has type $\mathtt{!}\alpha\mathtt{.?}\beta\mathtt{.end}$ and `x` has type α. The function `echo_client` starts by sending the message `x` over the endpoint `ep`. The construction `let` rebinds the name `ep` to the endpoint returned by the primitive `send`, which now has type $\mathtt{?}\beta\mathtt{.end}$. The endpoint is then used for receiving a message of type β from the service. Finally, echo_client closes the session and returns the received message.

The service is implemented by the echo_service function below, which uses the parameter ep of type ?α.!α.end to receive a message x and then to sent it back to the client before closing the session.

```
let echo_service ep =
  let x, ep = Session.receive ep in
  let ep = Session.send x ep in
  Session.close ep
```

There is an interesting asymmetry between (the types of) client and service in that the message x sent by the service is the very same message it receives, whereas the message res received by the client does not necessarily have the same type as the message x it sends. Indeed, there is nothing in echo_client suggesting that x and res are somewhat related. This explains the reason why the session type of the endpoint used by the client (!α.?β.end) is more general than that used by the service (?α.!α.end) aside from the fact that the two session types describe protocols with complementary actions. In particular, !α.?β.end is *not* dual of ?α.!α.end according to the definition of duality given earlier: in order to connect client and service, β must be *unified* with α. The code that connects echo_client and echo_service through a session is shown below:

```
let _ =
  let a, b = Session.create () in
  let _ = Thread.create echo_service a in
  print_endline (echo_client b "Hello, world!")
```

The code creates a new session, whose endpoints are bound to the names a and b. Then, it activates a new thread that applies echo_service to the endpoint a. Finally, it applies echo_client to the remaining endpoint b.

We now wish to generalize the echo service so that a client may decide whether to use the service or to stop the interaction without using it. A service that offers these two choices is illustrated below:

```
let opt_echo_service ep =
  match Session.branch ep with
  | `Msg ep → echo_service ep
  | `End ep → Session.close ep
```

In this case the service uses the branch primitive to wait for a label selected by the client. We use OCaml's polymorphic variant tags (`Msg and

`` `End `` in this case) as labels because they do not have to be declared explicitly, unlike data constructors of plain algebraic data types. The initial type of ep is now $\&[\texttt{End} : \texttt{end}, \texttt{Msg} : ?\alpha.!\alpha.\texttt{end}]$ and the value returned by branch has type $[\texttt{End} : \texttt{end}, \texttt{Msg} : ?\alpha.!\alpha.\texttt{end}]$. In the Msg branch the service behaves as before. In the End branch the service closes the session without performing any further communication.

The following function realizes a possible client for opt_echo_service:

```
let opt_echo_client ep opt x =
  if opt then
    let ep = Session.select (fun x → `Msg x) ep
    in echo_client ep x
  else
    let ep = Session.select (fun x → `End x) ep
    in Session.close ep; x
```

This function has type $\oplus[\texttt{End} : \texttt{end}, \texttt{Msg} : !\alpha.?\alpha.\texttt{end}] \to \texttt{bool} \to \alpha \to \alpha$ and its behavior depends on the boolean parameter opt: when opt is true, the client selects the label Msg and then follows the same protocol as echo_client; when opt is false, the client selects the label End and then closes the session. Note that we have to η-expand the polymorphic variant tags `` `Msg `` and `` `End `` so that their type matches that expected by select. When the same label is used several times in the same program, it is convenient to define the η-expansion once, for example as

```
let _Msg x = `Msg x
let _End x = `End x
```

Note also that the messages sent and received now have the same type in the initial type of ep. This is because of the structure of opt_echo_client, which returns either x or the message returned by the service.

A further elaboration of the echo service allows the client to send an arbitrary number of messages before closing the session. In order to describe this protocol we must extend the syntax of session types presented earlier to permit recursive types. In practice, the representation of session types we will choose in Section 11.3 allows us to describe recursive protocols by piggybacking on OCaml's support for equi-recursive types, which is enabled by passing the -rectypes option to the compiler. The implementation of the elaborated echo service is therefore a straightforward recursive function:

```
let rec rec_echo_service ep =
  match Session.branch ep with
  | `Msg ep → let x, ep = Session.receive ep in
              let ep = Session.send x ep in
              rec_echo_service ep
  | `End ep → Session.close ep
```

Note the recursive call `rec_echo_service ep` in the `Msg` branch, which allows the server to accept again a choice from the client after replying back to a request. The `rec_echo_service` function now expects an endpoint `ep` of type $\texttt{rec}\,A\&[\texttt{End}:\texttt{end},\texttt{Msg}:\,?\alpha.!\alpha.A]$ where $\texttt{rec}\,A\,T$ denotes the (equi-recursive) session type T in which occurrences of A stand for the session type itself.

The following client

```
let rec rec_echo_client ep =
  function
  | [] → let ep = Session.select _End ep in
         Session.close ep; []
  | x :: xs → let ep = Session.select _Msg ep in
              let ep = Session.send x ep in
              let y, ep = Session.receive ep in
              y :: rec_echo_client ep xs
```

has type $\texttt{rec}\,A\oplus[\texttt{End}:\texttt{end},\texttt{Msg}:\,!\alpha.?\beta.A] \to \texttt{list}\;\alpha \to \beta\;\texttt{list}$ and repeatedly invokes the recursive echo service on each element of a list.

11.3 API Implementation

In order to implement the API presented and used in the previous sections we have to make some choices regarding the `OCaml` representation of session types and of session endpoints. In doing so we have to take into account the fact that `OCaml`'s type system is not substructural and therefore is unable to statically check that session endpoints are used linearly. In the rest of this section we address these concerns and then detail the implementation of the API in Table 11.1.

Representation of session types. `FuSe` relies on the encoding of session types proposed by Dardha *et al.* [1] and further refined by Padovani [13]. The basic idea is that a sequence of communications on a session endpoint can be compiled as a sequence of one-shot communications on linear channels

(channels used exactly once) where each exchanged message carries the actual *payload* along with a *continuation*, namely a (fresh) channel on which the subsequent communication takes place.

The image of the encoding thus relies on two types:

- a type $\mathbb{0}$ which is not inhabited, and
- a type $\langle\rho, \sigma\rangle$ which describes channels for receiving messages of type ρ and sending messages of type σ. Both ρ and σ can be instantiated with $\mathbb{0}$ to indicate that no message is respectively received and/or sent.

The correspondence between session types T and types of the form $\langle t, s\rangle$ is given by the map $[\![\cdot]\!]$ defined below

Encoding of session types

$$\begin{aligned}
[\![\texttt{end}]\!] &= \langle\mathbb{0}, \mathbb{0}\rangle\\
[\![?t.T]\!] &= \langle[\![t]\!] \times [\![T]\!], \mathbb{0}\rangle\\
[\![!t.T]\!] &= \langle\mathbb{0}, [\![t]\!] \times [\![\overline{T}]\!]\rangle\\
[\![\&[\texttt{l}_i : T_i]_{i\in I}]\!] &= \langle[\texttt{l}_i : [\![T_i]\!]]_{i\in I}, \mathbb{0}\rangle\\
[\![\oplus[\texttt{l}_i : T_i]_{i\in I}]\!] &= \langle\mathbb{0}, [\texttt{l}_i : [\![\overline{T_i}]\!]]_{i\in I}\rangle\\
[\![A]\!] &= \langle\rho_A, \sigma_A\rangle\\
[\![\overline{A}]\!] &= \langle\sigma_A, \rho_A\rangle
\end{aligned}$$

and extended homomorphically to all types. We assume that for each session type variable A there exist two distinct type variables ρ_A and σ_A that are also different from any other type variable α.

For example, the session type $?\alpha.A$ is encoded as $\langle\alpha\times\langle\rho_A, \sigma_A\rangle, \mathbb{0}\rangle$, which describes a channel for receiving a message of type $\alpha\times\langle\rho_A, \sigma_A\rangle$ consisting of a component of type α (that is the actual payload of the communication) and a component of type $\langle\rho_A, \sigma_A\rangle$ (that is the continuation channel on which the rest of the communication takes place). There is a twist in the encoding of outputs for the session type of the continuation is dualized. The reason for this is that the type associated with the continuation channel in the encoding describes the behavior of the *receiver* of the continuation rather than that of the *sender*. As we will see, this twist provides us with a simple way of expressing duality relations between session types, even when they are (partially) unknown. The encodings of $\oplus[\texttt{l}_i : T_i]_{i\in I}$ and $\&[\texttt{l}_i : T_i]_{i\in I}$ follow the same lines and make use of polymorphic variant types to represent the selected or received choice. As an example, the encoding of $T = \oplus[\texttt{End} : \texttt{end}, \texttt{Msg} : !\alpha.?\beta.\texttt{end}]$ is computed as follows

$$\begin{aligned}
[\![T]\!] &= \langle \mathbb{0}, \texttt{[End} : [\![\texttt{end}]\!], \texttt{Msg} : [\![\texttt{?}\alpha\texttt{.!}\beta\texttt{.end}]\!]\texttt{]}\rangle \\
&= \langle \mathbb{0}, \texttt{[End} : \langle \mathbb{0}, \mathbb{0}\rangle, \texttt{Msg} : \langle \alpha \times [\![\texttt{!}\beta\texttt{.end}]\!], \mathbb{0}\rangle\texttt{]}\rangle \\
&= \langle \mathbb{0}, \texttt{[End} : \langle \mathbb{0}, \mathbb{0}\rangle, \texttt{Msg} : \langle \alpha \times \langle \mathbb{0}, \beta \times [\![\texttt{end}]\!]\rangle, \mathbb{0}\rangle\texttt{]}\rangle \\
&= \langle \mathbb{0}, \texttt{[End} : \langle \mathbb{0}, \mathbb{0}\rangle, \texttt{Msg} : \langle \alpha \times \langle \mathbb{0}, \beta \times \langle \mathbb{0}, \mathbb{0}\rangle\rangle, \mathbb{0}\rangle\texttt{]}\rangle
\end{aligned}$$

If instead we consider the session type $\overline{T} = \texttt{\&[End} : \texttt{end}, \texttt{Msg} : \texttt{?}\alpha\texttt{.!}\beta\texttt{.end]}$, then we derive:

$$\begin{aligned}
[\![\overline{T}]\!] &= \langle \texttt{[End} : [\![\texttt{end}]\!], \texttt{Msg} : [\![\texttt{?}\alpha\texttt{.!}\beta\texttt{.end}]\!]\texttt{]}, \mathbb{0}\rangle \\
&= \langle \texttt{[End} : \langle \mathbb{0}, \mathbb{0}\rangle, \texttt{Msg} : \langle \alpha \times [\![\texttt{!}\beta\texttt{.end}]\!], \mathbb{0}\rangle\texttt{]}, \mathbb{0}\rangle \\
&= \langle \texttt{[End} : \langle \mathbb{0}, \mathbb{0}\rangle, \texttt{Msg} : \langle \alpha \times \langle \mathbb{0}, \beta \times [\![\texttt{end}]\!]\rangle, \mathbb{0}\rangle\texttt{]}, \mathbb{0}\rangle \\
&= \langle \texttt{[End} : \langle \mathbb{0}, \mathbb{0}\rangle, \texttt{Msg} : \langle \alpha \times \langle \mathbb{0}, \beta \times \langle \mathbb{0}, \mathbb{0}\rangle\rangle, \mathbb{0}\rangle\texttt{]}, \mathbb{0}\rangle
\end{aligned}$$

Remarkably we observe that the encoding of $\overline{T}$ can be obtained from that of T by swapping the two components of the resulting channel types. This is a general property:

Theorem 1 *If* $[\![T]\!] = \langle t, s\rangle$, *then* $[\![\overline{T}]\!] = \langle s, t\rangle$.

An equivalent way of expressing this result is the following: if $[\![T]\!] = \langle t_1, t_2\rangle$ and $[\![S]\!] = \langle s_1, s_2\rangle$, then

$$T = \overline{S} \iff [\![T]\!] = [\![\overline{S}]\!] \iff t_1 = s_2 \wedge t_2 = s_1$$

meaning that the chosen encoding allows us to reduce session type duality to type equality. This property holds also for unknown or partially known session types. In particular, $[\![A]\!] = \langle \rho_A, \sigma_A\rangle$ and $[\![\overline{A}]\!] = \langle \sigma_A, \rho_A\rangle$.

We end the discussion of session type representation with two remarks. First, although the representation of session types chosen in `FuSe` is based on the continuation-passing encoding of sessions into the linear π-calculus [1], we will implement the communication primitives in `FuSe` so that only the payload (or the labels) are actually exchanged. Therefore, the semantics of `FuSe` communication primitives is consistent with that given in [4] and the components corresponding to continuations in the above types are solely used to relate the types of session endpoints as these are passed to, and returned from, `FuSe` communication primitives. Second, the `OCaml` type system is not substructural and there is no way to qualify types of the form $\langle t, s\rangle$ as linear, which is a fundamental requirement for the type safety of the API. We will overcome this limitation by means of a mechanism that detects linearity violations at runtime. Similar mechanisms have been proposed by Tov and Pucella [20] and by Hu and Yoshida [5].

Table 11.2 OCaml interface of the API for binary sessions

```
module Session : sig
  type 𝟘
  type (ρ,σ) st (* OCaml syntax for ⟨ρ,σ⟩ *)
  val create  : unit → (ρ,σ) st × (σ,ρ) st
  val close   : (𝟘,𝟘) st → unit
  val send    : α → (𝟘,(α × (σ,ρ) st)) st → (ρ,σ) st
  val receive : ((α × (ρ,σ) st),𝟘) st → α × (ρ,σ) st
  val select  : ((σ,ρ) st → α) → (𝟘,[>] as α) st → (ρ,σ) st
  val branch  : ([>] as α,𝟘) st → α
end
```

Having chosen the representation of session types, we can see in Table 11.2 the OCaml interface of the module that implements the binary session API. In OCaml syntax, the type $\langle t, s\rangle$ is written `(t,s) st`. There is a direct correspondence between the signatures of the functions in Table 11.2 and those shown in Table 11.1 so we only make a couple of remarks. First, we extensively use Theorem 1 whenever we need to refer to a session type and its dual. This can be seen in the signatures of `create`, `send` and `select` where both `(ρ,σ) st` and `(σ,ρ) st` occur. Second, in the types of `select` and `branch` the syntax `[>] as α` means that α can only be instantiated with a polymorphic variant type. Without this constraint the signatures of `select` and `branch` would be too general and the API unsafe: it would be possible to `receive` a label sent with `select`, or to `branch` over a message sent with `send`. Note that the constraint imposed by `[>] as α` extends to every occurrence of α in the same signature.

By comparing Tables 11.1 and 11.2 it is clear that the encoding makes session types difficult to read. This problem becomes more severe as the protocols become more involved. The distribution of FuSe includes an auxiliary tool, called `rosetta`, that implements the inverse of the encoding to pretty print encoded session types into their familiar notation. The tool can be useful not only for documentation purposes but also to decipher the likely obscure type error messages issued by OCaml. Hereafter, when presenting session types inferred by OCaml, we will often show them as pretty printed by `rosetta` for better clarity.

Representation of session endpoints. Session primitives can be easily implemented on top of any framework providing channel-based communications. FuSe is based on the `Event` module of OCaml's standard library, which provides communication primitives in the style of Concurrent ML [16] and the abstract type *t* Event.channel for representing channels carrying messages of type *t*. It is convenient to wrap the Event module so as to implement *unsafe communication channels*, thus:

```
module UnsafeChannel : sig
  type t
  val create     : unit → t
  val send       : α → t → unit
  val receive    : t → α
end = struct
  type t         = unit Event.channel
  let create     = Event.new_channel
  let send x u   = Event.sync
                     (Event.send u (Obj.magic x))
  let receive u = Obj.magic
                     (Event.sync (Event.receive u))
end
```

We just need three operations on unsafe channels, create, send and receive. The first one creates a new unsafe channel, which is simply an Event channel for exchanging messages of type unit. The choice of unit over any other OCaml type is immaterial: the messages exchanged over a session can be of different types, hence the type parameter we choose here is meaningless because we will perform unsafe cast at each communication. These casts cannot interfere with the internals of the Event module because *t* Event.channel is parametric on the type *t* of messages and therefore the operations in Event cannot make any assumption on their content. The implementation of send and receive on unsafe channels is a straightforward adaptation of the corresponding primitives of the Event module. Observe that, consistently with the communication API of Concurrent ML, Event.send and Event.receive do not perform communications themselves. Rather, they create *communication events* which occur only when they are synchronized through the primitive Event.sync. The Obj.magic function from the standard OCaml library has type $\alpha \to \beta$ and performs the necessary unsafe casts.

We now have all the ingredients for giving the concrete representation of (encoded) session types. This representation is kept private to the FuSe module so that the user can only manipulate session endpoint through the provided API:

```
type (α,β) st = { chan : UnsafeChannel.t;
                  mutable valid : bool }
```

A session type is represented as a record with two fields: the chan field is a reference to the unsafe channel on which messages are exchanged; the mutable valid field is a boolean flag that indicates whether the endpoint can be safely used or not. Every operation that uses the endpoint first checks whether the endpoint is valid. If this is the case, the valid flag of the endpoint is reset to false so that any subsequent attempt to reuse the same endpoint can be detected. Otherwise, an InvalidEndpoint exception is raised. It is convenient to encapsulate this functionality in an auxiliary function use, which is private to the module and whose implementation is shown below:

```
let use u = if u.valid then u.valid ← false
            else raise InvalidEndpoint
```

In principle, checking that the valid field is true and resetting it to false should be performed atomically, to account for the possibility that several threads are attempting to use the same endpoint simultaneously. In practice, since OCaml's scheduler is not preemptive and use allocates no memory, the execution of use is guaranteed to be performed atomically in OCaml's runtime environment. Different programming languages might require a more robust handling of the validity flag [13].

Whenever an operation on a session endpoint completes and the session endpoint is returned, its valid flag should be set to true again. Doing so on the existing record, though, would be unsafe. Instead, a new record referring to the very same unsafe channel must be created. Again it is convenient to provide this functionality as a private, auxiliary function fresh:

```
let fresh u = { u with valid = true }
```

Implementation of communication primitives. A new session is initiatied by creating a new unsafe channel ch and returning the two peer endpoints of the session, which both refer to the same channel. The valid flag of each peer is set to true, indicating that it can be safely used:

```
let create () = let ch = UnsafeChannel.create ()
                in { chan = ch; valid = true },
                   { chan = ch; valid = true }
```

The implementation of `close` simply invalidates the endpoint. `OCaml`'s garbage collector takes care of any further finalization that may be necessary to clean up the corresponding unsafe channel:

```
let close = use
```

The `send` operation starts by checking that the endpoint is valid and, in this case, invalidates it. Then, the message `x` is transmitted over the underlying unsafe channel and a refreshed version of the endpoint is returned. The `receive` operation is analogous, except that it returns a pair containing the message received from the underlying unsafe channel and the refreshed endpoint:

```
let send x u =
  use u; UnsafeChannel.send x u.chan; fresh u
let receive u =
  use u; (UnsafeChannel.receive u.chan, fresh u)
```

The `select` operation is behaviorally equivalent to `send`, since its purpose is to transmit the selected label (which is its first argument) over the channel. On the other hand the `branch` operation injects the refreshed session endpoint with the function received from the channel:

```
let select = send
let branch u =
  use u; UnsafeChannel.receive u.chan (fresh u)
```

We conclude this section showing the type inferred by `OCaml` for the `rec_echo_client` defined in Section 11.1:

```
val rec_echo_client :
  (𝟘,[> `End of (𝟘,𝟘) st
     |  `Msg of (β × (𝟘,γ × (𝟘,α) st) st,𝟘) st]
     as α) st → β list → γ list
```

As expected, the type is rather difficult to understand. Part of this difficulty is a consequence of the fact that the type expression *t* `as` α, which is used in `OCaml` also to denote a recursive type, is placed in a position such that *t* does not correspond to the encoding of a session type. It is only by unfolding this recursive type that one recovers an image of the encoding function. The same signature pretty printed by `rosetta` becomes

```
val rec_echo_client :
  rec X.⊕[ End: end | Msg: !α.?β.X ] →
  α list → β list
```

whose interpretation is straightforward.

11.4 Extended Example: The Bookshop

In this section we develop a `FuSe` version of a known example from the literature [4], where mother and child order books from an online bookshop. The purpose of the programming exercise is threefold. First, we see a usage instance of the `accept` and `request` primitives provided by `FuSe` for establishing sessions over service channels. Second, we discuss a non-trivial example in which the session types automatically inferred by `OCaml` are at the same time more general and more precise than those given by Gay and Vasconcelos [4]. This is made possible thanks to the support for parametric polymorphism and subtyping in session types that `FuSe` inherits for free from `OCaml`'s type system. Finally, we use the example to argue about the effectiveness of the `FuSe` implementation of binary sessions in detecting protocol violations, considering that `FuSe` combines both static and dynamic checks.

Service channels in `FuSe` are provided by the module `Service`, whose signature is shown below.

```
module Service : sig
  type α t
  val create  : unit → α t
  val accept  : (ρ,σ) st t → (ρ,σ) st
  val request : (ρ,σ) st t → (σ,ρ) st
  val spawn   : ((ρ,σ) st → unit) → (ρ,σ) st t
end
```

The type A `Service.t` describe a service channel that allows initiation of sessions of type $\overline{A}$. A session is created when two threads invoke `accept` and `request` over the same service channel. In this case, `accept` returns a session endpoint of type A and `request` returns its peer of type $\overline{A}$.

The bookshop is modeled as a function that waits for session initiations on the service channel `showAccess` and invokes `shopLoop` at each connection:

```
  let shop shopAccess =
    shopLoop (Service.accept shopAccess) []
```

A session initiated with the bookshop is handled by the function `shopLoop`, which operates over the established session endpoint `s` and the current list `order` of books in the shopping cart. The `shopLoop` function is recursive and repeatedly offers the possibility of adding a new book by selecting the `Add` label. When `Checkout` is selected instead, the bookshop waits for a credit card number and an address and sends back an estimated delivery date computed by an unspecified deliveryOn function.

```
let rec shopLoop s order =
  match Session.branch s with
  | `Add s →
      let book, s = Session.receive s in
      shopLoop s (book :: order)
  | `CheckOut s →
      let card, s = Session.receive s in
      let address, s = Session.receive s in
      let s = Session.send (deliveryOn order) s in
      Session.close s
```

The type inferred by OCaml for shopLoop is

```
val shopLoop :
  rec X.&[ Add: ?α.X | CheckOut: ?β.?γ.!day.end ]
  → α list → unit
```

which is structurally the same given by Gay and Vasconcelos [4], except for the type variables α, β and γ. Indeed, the body of shopLoop does not use the received values book, card and address and therefore their type remains generic.

We now model a mother process placing an order for two books, one chosen by her and another selected by her son. In principle, the mother could let the son select his own book by delegating the session with the bookshop to him. However, the mother wants to be sure that her son will buy just one book that is suitable for his age. To enforce these constraints, the mother sends her son a voucher, that is a function providing a controlled interface with the bookshop. Overall, the mother is modeled thus:

```
let mother card addr shopAccess sonAccess book =
  let c = Service.request shopAccess in
  let c = Session.select _Add c in
  let c = Session.send book c in
  let s = Service.request sonAccess in
  let s = Session.send (voucher card addr c) s in
  Session.close s
```

where the parameters card, addr and book stand for information about payment, delivery address and mother's book. In addition, shopAccess and sonAccess are the service channels for connecting with the bookshop and the son, respectively. The mother establishes a session c with the bookshop and adds book to the shopping cart. Afterwards, she initiates another session s for sending the voucher to her son. The voucher is modeled by the function:

```
let voucher card address c book =
  let c =
    if isChildrensBook book then
      let c = Session.select _Add c in
      Session.send book c
    else c
  in
  let c = Session.select _CheckOut c in
  let c = Session.send card c in
  let c = Session.send address c in
  let day, c = Session.receive c in
  Session.close c
```

where book is chosen by the son. If book is appropriate – something that is checked by the unspecified function isChildrensBook – the book is added to the shopping cart. Then, the order is completed and the connection with the bookshop closed.

For voucher and mother OCaml infers the following types:

```
val voucher : α → β →
  rec X.⊕[ Add: !γ.X | CheckOut: !α.!β.?δ.end ]
  → γ → unit
val mother : α → β →
  &[ Add: ?γ.rec X.&[ Add: ?δ.X
                    | CheckOut: ?α.?β.!ϵ.end ]
   ] Service.t → ?(δ → unit).end Service.t → γ
  → unit
```

In contrast to the type of mother given by Gay and Vasconcelos [4], the type inferred by OCaml makes it clear that mother always adds *at least one book* to the shopping cart. The connection between mother and shopLoop is still possible because the protocol followed by mother is more deterministic than – or a supertype of [3] – the one she is supposed to follow.

To finish the exercise we model the son as the following function:

```
let son sonAccess book =
  let s = Service.accept sonAccess in
```

```
    let f, s = Session.receive s in
    f book;
    Session.close s
```

where `sonAccess` is the service channel used for accepting requests from his mother and `book` is the book he wishes to purchase. Note that the mother sends a function (obtained as the partial application of `voucher`) which is saturated by the son who provides the chosen `book`.

Overall, the code for connecting the three peers is shown below:

```
let _ =
  let mCard = "0123 4567 7654 3210" in
  let mAddr = "17 session type rd" in
  let mBook = "Life of Ada Lovelace" in
  let sBook = "1984" in
  let shopAccess = Service.create () in
  let sonAccess = Service.create () in
  let _ = Thread.create shop shopAccess in
  let _ = Thread.create (son sonAccess) sBook in
  mother mCard mAddr shopAccess sonAccess mBook
```

It is not possible to qualify session endpoints as linear resources in `OCaml`. This means that there are well-typed programs that, by using session endpoints non-linearly, cause communication errors and/or protocol violations. In the rest of this section we use the example developed so far to do some considerations concerning the effectiveness of the library in detecting programming errors involving session endpoints. In particular we argue that, despite the lack of linear qualification of session endpoints, `OCaml`'s type system is still capable of detecting a fair number of linearity violations. In the worst case, those violations that escape `OCaml`'s type checker are at least detected at runtime with the mechanism we have put in place in Section 11.3.

We can simulate a linearity violation by replacing the session endpoint bound by a `let` with `_`. For example, we can replace line 10 in the body of `voucher` with

```
10      let _ = Session.send address c in
```

so that the very same endpoint `c` is used both for this `send` and also for the subsequent `receive`. This linearity violation is detected by `OCaml` because the type of a session endpoint used for an output is incompatible (*i.e.*, not unifiable) with that of an endpoint used for an input. Now suppose that we replace line 8 in the same function with

```
8       let _ = Session.select _CheckOut c in
```

so that the same endpoint `c` is used for both a `select` and the subsequent `send`. Even if `select` and `send` are both output operations, the type of messages resulting from the encoding of a plain message output has a topmost $\times$ constructor which is incompatible with the polymorphic variant type resulting from the encoding of a label selection. Therefore, also this linearity violation is detected by `OCaml`'s type checker. In general, any linearity violation arising from the use of different communication primitives is detected by `OCaml`. Consider then line 9, and suppose that we replace it with

```
9       let _ = Session.send card c in
```

so that the same endpoint `c` is used for sending both `card` and `address`. In this case the session endpoint is used for performing two plain outputs and the sent messages have compabile (*i.e.*, unifiable) types. Therefore, taken in isolation, the `voucher` function would be well typed. In the context of the whole program, however, `OCaml` detects a type error also in this case. The point is that the faulty version of `voucher` now implements a different protocol than before. In particular, it appears as if `voucher` sends just one message after selecting `CheckOut` and before receiving the estimated delivery date. On the contrary, the protocol of the bookshop as implemented by `shopLoop` still expects to receive two messages before the delivery date is sent back to the client. Therefore, the protocols of the bookshop and the one inferred by the combination of `mother` and `voucher` are no longer dual to each other and the session request performed by `mother` to the bookshop is ill typed. For this problem to go undetected, there must be *another* linearity violation in the body of `shopLoop`, in the place that corresponds exactly to the point where the same violation occurs in `voucher`.

A simpler example of linearity violation that goes undetected by `OCaml`'s type checker can be obtained by duplicating the `f book` application in the body of the `son` function. This modification might correspond either to a genuine programming error or to a malicious attempt of `son` to purchase more than one book. The reason why this duplication results into a linearity violation is that the closure corresponding to `f` contains the session endpoint `c` from `mother`, so applying `f` twice results in two uses of the same `c`. This error is detected by the type system of Gay and Vasconcelos [4] where the function `f` has a linear arrow type. In `FuSe`, the program compiles correctly, but the second application of `f` triggers the runtime mechanism that detects linearity violations causing the `InvalidEndpoint` exception to be raised.

11.5 Related Work

Several libraries of binary sessions have been proposed for different functional programming languages. Most libraries for `Haskell` [6, 12, 15, 17] use a monad that encapsulates the endpoints of open sessions. Besides being a necessity dictated by the lazyness of the language, the monad prevents programmers from accessing session endpoints directly thus guaranteeing that endpoint linearity is not violated. The monad also tracks the evolution of the type of session endpoints automatically, not requiring the programmer to rebind explicitly the same endpoint over and over again. However, the monad has a cost in terms of either expressiveness, usability, or portability: the monad defined by Neubauer and Thiemann [12] supports communication on a single channel only and is therefore incapable of expressing session interleaving or delegation. Pucella and Tov [15] propose a monad that stores a stack of endpoints (or, better, of their capabilities) allowing for session interleaving and delegation to some extent. The price for this generality is that the programmer has to write explicit monadic actions to reach the channel/capability to be used within the monad; also for this reason delegation is severely limited. Imai *et al.* [6] show how to avoid writing such explicit actions relying on a form of type-level computations. Lindley and Morris [9] describe another Haskell embedding of session types that provides first-class channels. Linearity is enforced statically using higher-order abstract syntax.

A different approach is taken in `Alms` [19, 21], a general-purpose programming language whose type system supports parametric polymorphism, abstract and algebraic data types, and built-in affine types as well. Tov [19] illustrates how to build a library of binary sessions on top of these features. Because `Alms`' type system is substructural, affine usage of session endpoints is guaranteed statically by the fact that session types are qualified as affine. Further embeddings of session types in other experimental and domain-specific languages with substructural type systems have been described by Mazurak and Zdancewic [10], Lindley and Morris [8], and Morris [11].

Scalas and Yoshida [18] propose a library of binary session for `Scala` that is very related to our approach. As in `FuSe`, Scalas and Yoshida use a runtime mechanism to compensate for the lack of affine/linear types in `Scala` and work with the encoded representation of session types given by Dardha *et al.* [1]. A notable difference is that `Scala` type system is nominal, so that encoded session types are represented by `Scala` (case) classes which must be either provided by the programmer or generated from the protocol. This means that the protocol cannot be inferred automatically from the code and that the subtyping relation between session types is constrained by the (fixed) subclassing relation between the classes that represent them.

The main source of inspiration for the representation of session types in `FuSe` originates from the continuation-passing encoding of binary sessions [1] and partially studied also in some earlier works [2, 7]. Our representation of encoded session types allows session type duality to be expressed solely in terms of type equality, whereas the representation chosen by Dardha *et al.* [1] requires a residual albeit simple notion of duality for the topmost channel type capability. Another difference is that we consider the encoding at the type level only, not requiring the explicit exchange of continuation channels for the implementation of communication primitives. For these reasons, the soundness of the encoding [1] cannot be used directly to argument about the soundness of `FuSe`'s typing discipline. Padovani [13] formalizes `FuSe`'s approach to binary sessions along with the necessary conditions under which the program does not raise exceptions. The same paper also illustrates a simple monadic API built on top of the primitives in Table 11.1 and investigates the overhead of the various approaches to linearity.

In addition to the features described in this chapter, `FuSe` supports sequential composition of session types. This feature makes it possible to describe with greater precision protocols whose set of (finite) traces is *context-free* as opposed to regular [14,22]. As discussed by Thiemann and Vasconcelos [22], these protocols arise naturally in the serialization of structured data types. Currently, `FuSe` provides the first and only implementation of context-free session type checking and inference.

Acknowledgments We thank the anonymous reviewers for their careful reading of the chapter and suggestions of improvements and related work.

References

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

[2] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *Proceedings of CONCUR'11*, LNCS 6901, pages 280–296. Springer, 2011.

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

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

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

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

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

[8] Sam Lindley and J. Garrett Morris. Lightweight Functional Session Types. Unpublished manuscript available at `http://homepages.inf.ed.ac.uk/slindley/papers/fst-draft-february2015.pdf`, 2015.

[9] Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In *Proceedings of Haskell'16*, Haskell 2016, pages 133–145, New York, NY, USA, 2016. ACM.

[10] Karl Mazurak and Steve Zdancewic. Lolliproc: to concurrency from classical linear logic via curry-howard and control. In *Proceeding of ICFP'10*, pages 39–50. ACM, 2010.

[11] J. Garrett Morris. The best of both worlds: linear functional programming without compromise. In *Proceedings of ICFP'16*, pages 448–461. ACM, 2016.

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

[13] Luca Padovani. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming*, 27, 2017.

[14] Luca Padovani. Context-Free Session Type Inference. In *Proceedings of the 26th European Symposium on Programming (ESOP'17)*, LNCS. Springer, 2017.

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

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

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

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

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

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

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

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