FuSe: an OCaml implementation of binary sessions

Luca Padovani – Dipartimento di Informatica, Università di Torino – Italy

Contents

1 Overview

1.1 Introduction to binary sessions

A *binary session* is a communication channel used to exchange messages between two threads according to an agreed protocol. The session is accessed by means of two *peer endpoints*, each owned by one thread: a message sent over one endpoint is received from its peer, and vice versa. Each endpoint has a *session type* that specifies the input/output operations allowed on it, including the type of messages that can be sent and received, and in which order. To ensure communication safety, peer endpoints must have *dual* session types, such that an input operation on one endpoint is matched by a corresponding output on the peer, and vice versa.

The literature on binary sessions has started with [Honda, 1993, Honda et al., 1998] and its theoretical and practical aspects have been investigated in a large number of subsequent works. In particular, Gay and Vasconcelos [2010] have studied an integration of binary sessions into an `ML`-like functional language equipped with a set of session-based communication primitives. `FuSe` is a lightweight implementation of such primitives as an `OCaml` module that turns any out-of-the-box installation of `OCaml` into a session type checker. Below is a summary of the features supported by `FuSe` and illustrated in greater detail in the rest of this tutorial:

Input/Output: `FuSe` sessions enable the exchange of any `OCaml` data type over sessions, including session endpoints themselves (this latter feature is often referred to as *delegation* in the technical literature) (Section 2.2).

Internal and external choices: `FuSe` sessions support branching whereby one thread *offers* a finite range of supported continuation protocols, each identified by a unique *label*, and the other thread *selects* one branch by sending the corresponding label. `FuSe` allows programmers to use arbitrary tag sets in the form of `OCaml`'s polymorphic variant tags (Section 2.3).

Parametric polymorphism: session types may contain (session) type variables allowing the specification of parametric protocols (such as `!`α`.?`α`.end`) and partially known protocols (such as `?int.`A). These features are used extensively in the typing of `FuSe` primitives.

Duality checking: `OCaml`'s type checker verifies whether the two endpoints of a session are used according to complementary protocols. It is possible to specify duality constraints also for session types containing session type variables.

Session type inference: programmers are not required to write protocol specifications explicitly, since the type inference engine of `OCaml` is capable of inferring (recursive, polymorphic) session types from the usage of communication primitives.

Runtime linearity checks: Given that `OCaml`'s type system is not substructural, only some – but not all – endpoint linearity violations are statically detected. `FuSe` provides a fallback mechanism that detects and signals the remaining violations at runtime (Section 2.4).

Unbounded protocols: `FuSe` supports equi-recursive session types by leveraging on `OCaml` equi-recursive types (Section 3.1).

Context-free protocols: `FuSe` provides higher-order combinators for dealing with sequential compositions in session types (Section 3.2). Using these combinators, `FuSe` gives support for context-free session types as defined by Thiemann and Vasconcelos [2016].

Service channels: `FuSe` provides support for shareable service channels to establish sessions using the familiar pair of `accept`/`request` connection primitives (Section 4.1).

Session subtyping: `FuSe` supports the subtyping relation for session types defined by Gay and Hole [2005] through explicit coercions. This allows endpoints to be used in accordance with the safe substitutability principle (Section 4.2).

1.2 Technical references

FuSe is based on an encoding of session types that has been studied by Kobayashi [2002], Demangeon and Honda [2011], and Dardha et al. [2012]. Padovani [2015] shows how the encoding can be used to embed session type checking into a conventional ML-like type system with support for parametric polymorphism. The representation of session types in the current version of FuSe is a further elaboration of that described by Padovani [2015] that is both simpler and more expressive. In particular, it fully supports the notion of session subtyping defined by Gay and Hole [2005] that considers recursive session types and extends it to parametric session types.

Runtime mechanisms for the detection of linearity and/or affinity violations similar to those adopted in FuSe have been described by Tov and Pucella [2010] and Hu and Yoshida [2016].

1.3 Types and notation

In FuSe, a session type is represented as an abstract OCaml type

```
('a, 'b) Session.st
```

which can be thought of as the type of session endpoints for receiving messages of type 'a and sending messages of type 'b (in reality, FuSe primitives instantiate 'a and 'b with types that specify other information in addition to the payload of communications). The type parameters 'a and 'b can be instantiated with the abstract type

```
Session._0
```

which is not inhabited and means "no message". For example, the type

```
(Session._0, 'a) Session.st
```

denotes a session endpoint from which no message can be received and on which a message of type 'a can be sent.

Every session type has a *dual* obtained by swapping its two type parameters and which represent the complementary protocol. For example, the session type `(Session._0, 'a) Session.st` is dual to `('a, Session._0) Session.st` since the complementary protocol of "sending a message of type 'a" is "receiving a message of type 'a". In general, if one session endpoint is used according to the session type `(t, s) Session.st` for arbitrary t and s, then we expect the peer endpoint to be used according to the dual session type `(s, t) Session.st`. This ensures communication safety within the session.

FuSe defines a series of type aliases to simplify writing complex session types and to make them (slightly) easier to read:

```
type 'a Session.ot = (Session._0, 'a) Session.st          (* output *)
type 'a Session.it = ('a, Session._0) Session.st          (* input  *)
type    Session.et = (Session._0, Session._0) Session.st  (* end    *)
```

In particular, 'a Session.ot denotes session endpoints for sending messages of type 'a, 'a Session.it denotes session endpoints for receiving messages of type 'a, and Session.et denotes endpoints that cannot be used for sending or receiving messages. They represent the end of a session or part thereof.

In addition to these types, FuSe also defines the abstract type

```
('a, 'b) Session.seq
```

that, combined with Session.st and the aforementioned aliases, can be used to represent *context-free session types* (Section 3.2).

For the sake of readability, in the rest of this tutorial we write types using a more compact notation, whose meaning and correspondence with OCaml syntax are shown in Table 1. Obviously, OCaml prints types using their concrete syntax rather than this notation, which may result in rather obscure error messages. However, the distribution of FuSe comprises a "decoder", an

Notation	Meaning	OCaml syntax
t, s	types	any OCaml type expression
$\alpha, \beta, \ldots$	type variables	'a, 'b, ...
T, S, ...	session types	(t,s) st
A, B, ...	session type variables	('a,'b) st
T dual	dual session type	(s,t) st if T is (t,s) st
A dual	dual session type variable	('b,'a) st if A is ('a,'b) st
end	terminated session type	et
?s.T	receive message	(s * T) it
!s.T	send message	(s * T dual) ot
$\&[l_i : T_i]_{i \in I}$	accept choice	([< 'l$_i$ of T$_i$]$_{i \in I}$) it
$\oplus[l_i : T_i]_{i \in I}$	select choice	([> 'l$_i$ of T$_i$ dual]$_{i \in I}$) ot
{A}.B	sequence	((A,B) seq, (A dual,B dual) seq) st

Table 1: Meaning and OCaml syntax of the notation used in this tutorial.

external tool that parses OCaml types (and a subset of OCaml module signatures) and pretty prints them using such notation. This pretty printing function is described by Padovani [2015].

The FuSe library comprises two different APIs for session communication: the "bare" version and the "monadic" version. This tutorial covers only the bare version of the library and all the examples discussed henceforth are supposed to occur in a file that starts with the declaration

```
module Session = Session.Bare
```

2 Basic Input/Output

2.1 Session initiation and termination

Before communication in a session can take place, the session must be created and its two endpoints given to two parallel threads. Session creation is achieved with the primitive

```
Session.create : unit → A * A dual
```

that, applied to (), returns the two endpoints of the session, with dual session types.

When communication is supposed to terminate within a session, each endpoint of the session should be closed with the primitive

```
Session.close : end → unit
```

In the current version of FuSe, Session.close is just a no-op. However, it is good practice to always close a session when it is no longer necessary for this helps OCaml to detect type errors. Also, the runtime system might detect the fact that an endpoint is left unused and signal this by raising an exception (Section 2.4).

2.2 Message passing

Basic message passing is supported by the primitives Session.send and Session.receive. The primitive

```
Session.send : α → !α.A → A
```

applied to a message of type t and an endpoint of type !t.T sends the message over the endpoint and returns the same endpoint with its type changed to T. Communication in FuSe is *synchronous*, therefore Session.send blocks the caller thread until a different thread performs the corresponding Session.receive on the peer endpoint. The primitive

```
Session.receive : ?α.A → α * A
```

applied to an endpoint of type `?t.T` waits for a message from the endpoint and returns a pair with the message, of type `t`, and the endpoint with its type changed to `T`.

A typical usage pattern of these primitives is shown in the code fragment below, which defines a function `op_client` that forwards its two arguments `x` and `y` over a session endpoint `ep`, then retrieves a message `result` from the same endpoint (presumably, some elaboration of `x` and `y`), closes `ep` and returns the result. In this and the following examples, we assume that the source file includes a line `module Session = Session.Bare` at the beginning, which selects the "bare" module for session communications supported by `FuSe`.

Tadder.ml:20-25
```
let op_client ep x y =
  let ep = Session.send x ep in
  let ep = Session.send y ep in
  let result, ep = Session.receive ep in
  Session.close ep;
  result
```

Note the explicit threading of the session endpoint `ep`: each communication primitive is applied to `ep`, it performs an input/output operation on it, and returns, possibly among other data, the same endpoint with a possibly different session type that describes the rest of the protocol. `OCaml` infers the following type for `op_client`

```
val op_client : !α.!β.?γ.end → α → β → γ
```

Below is a service that can communicate safely with the above client:

Tadder.ml:27-31
```
let add_service ep =
  let x, ep = Session.receive ep in
  let y, ep = Session.receive ep in
  let ep = Session.send (x + y) ep in
  Session.close ep
```

The service receives two messages `x` and `y` from `ep` and sends back their sum, before closing `ep`. The type inferred by `OCaml` for `add_server` is:

```
val add_service : ?int.?int.!int.end → unit
```

Note that the type of the messages received and sent by `add_service` is `int`, because the body of `add_service` uses `+`, which in `OCaml` denotes the sum of integer numbers. Conversely, the type of `op_client` is parametric in that of the exchanged messages, for `op_client` makes no assumption on their content.

The following code spawns a thread that executes `add_service` and connects `op_client` with `add_service` through a new session:

Tadder.ml:33-36
```
let _ =
  let a, b = Session.create () in   (* create session with endpoints a and b *)
  let _ = Thread.create add_service a in (* spawns service in its own thread *)
  print_int (op_client b 1 2)
```

Session type checking is useful to detect communication errors. The following variation of the code above shows an instance of communication error due to the fact that `op_client` attempts to send the wrong type of messages to `add_service`:

Fadder.ml:33-36
```
let _ =
  let a, b = Session.create () in
  let _ = Thread.create add_service a in
  print_int (op_client b 1.0 2.0)
```

`OCaml` refuses to compile this program yielding the following type error

```
Error: This expression has type
  float
but an expression was expected of type
  int
```

referring to the constant `1.0`. In words, `OCaml` has detected that this application of `op_client` would send a message of type `float` to `add_service`, whereas `add_service` expects to receive a message of type `int`. To understand how `OCaml` detects this problem, one should observe that `a` and `b` are peer endpoints of the same session created with `Session.create`. Therefore, from the typing of `Session.create`, we know that `a` and `b` must have dual session types. Then we see that `a` must have type `?int.?int.!int.end`, for this is the type of `add_service`'s argument, and `b` must have the dual type `!int.!int.?int.end`, meaning that the type variable α in the type of `op_client` is instantiated with `int`. This type is incompatible with that of `1.0`, which is the second argument of `op_client`.

Consider now the following variation of `add_service`

Finc.ml:27-30
```
let dec_service ep =
  let x, ep = Session.receive ep in
  let ep = Session.send (x - 1) ep in
  Session.close ep
```

which computes the predecessor of the received message and has type

```
val dec_service : ?int.!int.end → unit
```

If we try to connect `op_client` above with `succ_service` instead of `add_service` we obtain the type error

```
Error: This expression has type
  !int.?int.end
but an expression was expected of type
  !int.!α.?β.end
```

referred to the occurrence of `b` in `(op_client b 1 2)`. This time, the type of the message sent by `op_client` to `succ_service` has the correct type `int`, but `op_client` also tries to send a second message to `succ_service` at a point of the protocol when `succ_service` is instead trying to answer `op_client`.

2.3 Branching protocols

A protocol may have branching points representing alternative paths that can be selected by means of a tag. As for plain input/output, there is one *active* thread that selects a particular branch and a *passive* thread that accepts the selection. The handling of choices in `FuSe` is somewhat obscure compared to that of plain messages. This is due to the fact that tags should not explicitly occur in the types of the primitives that handle choices, to ensure their generality.

The primitive to select a branch has the following type

```
Session.select : (A dual → α) → ([>] as α) Session.ot → A
```

and can be applied to an endpoint whose type permits an output operation. The choice is selected by applying `Session.select` to a function that injects the peer of `ep` into a polymorphic variant type α. The typical usage of `Session.select` is:

```
let ep = Session.select (fun x → `Tag x) ep in ...
```

Since `OCaml` polymorphic variant tags are not curried, it is necessary to η-expand the tag appropriately. In principle, `Session.select` can be applied to any function with type A `dual` $\to \alpha$. In practice, it is recommended not to use functions more complicated that η-expansions such as the one above. The current implementation of `FuSe` ensures that the function passed to

`Session.select` is applied only when the selection has been accepted by the receiving thread, but this behavior could change in future versions of `FuSe`.

The passive thread is supposed to accept a selection with the following primitive

```
Session.branch : ([>] as α) Session.it → α
```

which returns the endpoint to which it is applied injected through the function passed by the active thread.

The code fragment below defines a function `math_service` that combines the functionalities of `add_service` and `dec_service` in Section 2.2:

Tchoices.ml:20-28

```
let math_service ep =
  match Session.branch ep with
  | `Add ep → let x, ep = Session.receive ep in
              let y, ep = Session.receive ep in
              let ep = Session.send (x + y) ep in
              Session.close ep
  | `Dec ep → let x, ep = Session.receive ep in
              let ep = Session.send (x - 1) ep in
              Session.close ep
```

To use this service, a client must first *select* a desired operation (this version of `math_service` supports `` `Add `` and `` `Dec ``), and then follow the corresponding protocol, which involves sending the right number of arguments to the service and retrieving the result:

Tchoices.ml:30-36

```
let math_client ep x y =
  let ep = Session.select (fun x → `Add x) ep in (* select `Add operation *)
  let ep = Session.send x ep in
  let ep = Session.send y ep in
  let result, ep = Session.receive ep in
  Session.close ep;
  result
```

`OCaml` infers the following types for `math_service` and `math_client`:

```
val math_service : &[Add : ?int.?int.!int.end, Dec : ?int.!int.end ]
val math_client  : ⊕[Add : !α.!β.?γ.end] → α → β → γ
```

Once again, `math_client` is parametric in the type of the messages exchanged on the endpoint. Apart from that, there is another major difference with `math_server`: the client only selects `Add`, whereas the service accepts both `Add` and `Dec`. The communication between client and service is safe anyway, since the client never tries to select an operation not supported by the service. In particular, the code fragment

Tchoices.ml:38-41

```
let _ =
  let a, b = Session.create () in
  let _ = Thread.create math_service a in
  print_int (math_client b 1 2)
```

that connects client and service is well typed. However, if we change the client so that it selects an unsupported operation, as in

Fchoices.ml:30-36

```
let math_client ep x y =
  let ep = Session.select (fun x → `Mul x) ep in (* select `Mul operation *)
  let ep = Session.send x ep in
  let ep = Session.send y ep in
  let result, ep = Session.receive ep in
  Session.close ep;
  result
```

then OCaml reports a type error that hints at the problem

```
This expression has type
  ⊕[ Add: !int.!int.?int.end | Inc: !int.?int.end ]
but an expression was expected of type
  ⊕[ Mul: !α.!β.?γ.end ]
The first variant type does not allow tag(s)
  ‘Mul
```

2.4 Endpoint linearity and overlaps

Virtually all session type systems are substructural, since communication safety and protocol fidelity are guaranteed provided that session endpoints are used linearly. OCaml's type system is not substructural, hence it is unable in general to understand when a session endpoint is used non linearly, namely when it is used disrespecting the threading of endpoints induced by the communication primitives or when it is prematurely discarded. For example, the function

Toverlap.ml:20-23
```
let client_with_overlap ep =
  let _ = Session.send 1 ep in
  let ep = Session.send 2 ep in (* overlapping use of ep *)
  Session.close ep
```

sends two numbers on the endpoint ep before closing it and yet its type is

```
val client_with_overlap : !int.end → unit
```

suggesting that client_with_overlap uses ep for sending *one* number only. This discrepancy between the obervable behavior of client_with_overlap and its type is due to the fact that client_with_overlap violates the linearity of ep: it is the very same ep that is used in the two send operations. In this case, we say that the second send overlaps with the first one, because it is performed on the very same endpoint of the first send and not on the endpoint returned by the first send.

FuSe has a runtime mechanism that detects overlaps and raises an InvalidEndpoint exception in such cases. The mechanism distinguishes between *valid* and *invalid* endpoints. A valid endpoint can be used for input/output operations or closed. Using the endpoint *invalidates* it, and any subsequent attempt to use the same endpoint will raise an InvalidEndpoint exception. Whenever an endpoint is used by a communication primitive and returned by the primitive, as for example in the case of send and receive, the resulting endpoint is "refreshed" to a valid state and can be used for subsequent operations.

It should be noted that the typing of communication primitives makes it possible to statically detect a number of overlaps. For example,

Foverlap1.ml:20-22
```
let overlap ep =
  let _ = Session.send 1 ep in
  Session.close ep (* statically detected overlap *)
```

yields the following type error

```
Error: This expression has type
  !int.A
but an expression was expected of type
  end
```

referred to the occurrence of ep on the last line, because the session type of endpoints that can be used for an output operation is incompatible with that of endpoints that can be closed.

Overlaps may be statically detected even if the overlapping operations are of the same kind (*e.g.*, two outputs or two inputs), provided that they involve messages with incompatible types.

A particular example is that of `send`s and `select`s, whose overlapping is detected because polymorphic variant types (occurring in the type of `Session.select`) are incompatible with products (occurring in the type of `Session.send`). For example,

Foverlap2.ml:20-23
```
let overlap ep =
  let _ = Session.select (fun x → `Tag x) ep in
  let _ = Session.send 1 ep in (* statically detected overlap *)
  Session.close ep
```

yields the type error

```
  Error: This expression has type
    ⊕[Tag : A]
  but an expression was expected of type
    !α.B
```

referred to the occurrence of `ep` in `Session.send 1 ep`. It is possible to query at runtime whether an endpoint is valid or not. This can be achieved with the function

```
  Session.is_valid : A → bool
```

It is also possible to *acquire* an endpoint, making sure that the obtained reference is the *only* valid reference to that endpoint in the entire program:

```
  Session.acquire : A → A
  Session.try_acquire : A → A option
```

The former function applied to an endpoint `ep` attempts to acquire `ep`. If `ep` is valid, the function invalidates every other occurrence of `ep` in the program and returns `ep` itself. If `ep` is invalid, it raises `InvalidEndpoint`. The second function returns either `Some ep` if `ep` is valid (and invalidates every other occurrence of `ep` in the program) or `None` if `ep` is invalid.

A different kind of linearity violation occurs when every reference to a valid endpoint is lost, so that the endpoint can no longer be used for input/output operations nor closed. This situation may lead to deadlocks, for there can be threads waiting for an interaction on the peer endpoint. `FuSe` has a runtime mechanism that detects these linearity violations and raises an `UnusedValidEndpoint` exception when they occur. It should be noted that the time at which this exception is raised is unpredictable in general, since the detection of unused valid endpoints is tied to the internal workings of `OCaml`'s garbage collector.

3 Unbounded and Sequential Protocols

3.1 Recursive session types

It is possibile to define protocols that involve arbitrarily long sequences of communications. `FuSe` takes full advantage of `OCaml`'s support for equi-recursive types and lifts them at the level of session types.

Below is an enhancement of the `math_service` discussed in Section 2.3 that is capable of handling an arbitrary number of `Add`, `Dec`, and `IsZero` operations before closing the session:

Tcounter.ml:20-34
```
let rec_math_service =
  let rec aux ep =
    match Session.branch ep with
    | `Add ep →    let x, ep = Session.receive ep in
                   let y, ep = Session.receive ep in
                   let ep = Session.send (x + y) ep in
                   aux ep
    | `Dec ep →    let x, ep = Session.receive ep in
                   let ep = Session.send (x - 1) ep in
```

```
                    aux ep
    | `IsZero ep → let x, ep = Session.receive ep in
                    let ep = Session.send (x == 0) ep in
                    aux ep
    | `Quit ep →    Session.close ep
  in aux
```

Correspondingly, the type of `rec_math_service` inferred by `OCaml` is

```
val rec_math_service :
  (&[ Add: ?int.?int.!int.α
    | Dec: ?int.!int.α
    | IsZero: ?int.!bool.α
    | Quit: end] as α) → unit
```

where the `OCaml` type expression `(t as α)` denotes the same type as `t` in which occurrences of α stand for the type itself.

As an example, the client below uses `rec_math_service` to compute the sum of the first `n` naturals

Tcounter.ml:36–54

```
let rec_math_client ep =
  let rec aux acc ep n =
    let ep = Session.select (fun x → `IsZero x) ep in
    let ep = Session.send n ep in
    let is_zero, ep = Session.receive ep in
    if is_zero then
      let ep = Session.select (fun x → `Quit x) ep in
      Session.close ep;
      acc
    else
      let ep = Session.select (fun x → `Add x) ep in
      let ep = Session.send acc ep in
      let ep = Session.send n ep in
      let acc, ep = Session.receive ep in
      let ep = Session.select (fun x → `Dec x) ep in
      let ep = Session.send n ep in
      let n, ep = Session.receive ep in
      aux acc ep n
  in aux 0 ep
```

and its inferred type is

```
val rec_math_client :
  (⊕[IsZero: !β.?bool.⊕[Add: !int.!β.?int.⊕[Dec: !β.?β.α],
                        Quit: end]] as α) → β → int
```

Note that `rec_math_client` is parametric in the type of its second argument, which represents the natural `n`. This is because there is nothing, in the body of `rec_math_client`, suggesting that it should have type `int`. Conversely, the `acc` parameter of `aux` is initialized to `0`, hence it is clear that its type should be `int`. Also, the session type of the `ep` argument of `rec_math_client` reflects the internal behavior of the function, rather than the availability of the operations in the service. In particular, the syntactic placement of recursions in the types of `rec_math_client` and `rec_math_service` differ considerably. `OCaml` is able to infer that `rec_math_client` and `rec_math_service` can interact correctly because an equi-recursive type is equivalent to its own unfolding, hence the syntactic placement of recursions is irrelevant.

3.2 Sequencing in Session Types

The `Session.(@>)` operator can be used for carrying out the prefix of a sequentially composed protocol and has the following signature:

```
Session.(@>) : (A → end) → {A}.B → B
```

The expression `f @> ep` where `ep` has type `{A}.B`, evaluates `f ep` where `f` is supposed to perform the protocol specified by `A`. Once this part of the protocol is completed and the type of `ep` has reduced to `end`, `@>` returns `ep` with its type changed to `B`, so that the subsequent part of the protocol can be performed. For the soundness of `@>` it is fundamental that the endpoint returned by `f` is exactly the same endpoint `ep` that was supplied to `f`. In the current version of `FuSe`, this requirement is verified by means of a runtime check. In case `f` returns an endpoint other than `ep`, an exception is raised.

To see a practical use of `@>`, we consider an example given by Thiemann and Vasconcelos [2016]. Let

Ttree.ml:20-20
```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

by the definition of an algebraic data type for representing binary trees. Suppose now that we want to define a function that serializes a binary tree over a session. We could try to use a function such as the following one:

Ttree.ml:57-63
```
let rec send_tree t ep =
  match t with
  | Leaf → Session.select (fun x → `Leaf x) ep
  | Node (v, l, r) → let ep = Session.select (fun x → `Node x) ep in
                     let ep = Session.send v ep in
                     let ep = send_tree l ep in
                     let ep = send_tree r ep in ep
```

Although the definition looks reasonable, the session type of the endpoint `ep` inferred by `OCaml` is not precise enough:

```
send_tree : α tree → ⊕[ Leaf: β | Node: !α.β ] as β → β
```

Note that the inferred protocol never terminates and allows for infinite sequences such as `Leaf, Leaf, Leaf, ...` not corresponding to any (serialized) binary tree. The problem is that conventional session type can only describe protocols whose set of (finite) traces is a regular language, whereas the protocol corresponding to serialized binary trees is context-free. We can use `@>` to specify a place in the code of `send_tree` that is meant to correspond to a sequential composition in the protocol, thus:

Ttree.ml:67-73
```
let rec send_tree t ep =
  match t with
  | Leaf → Session.select (fun x → `Leaf x) ep
  | Node (v, l, r) → let ep = Session.select (fun x → `Node x) ep in
                     let ep = Session.send v ep in
                     let ep = send_tree l @> ep in
                     let ep = send_tree r ep in ep
```

This revised version of `send_tree` is now typed correctly:

```
send_tree : α tree → ⊕[ Leaf: end | Node: !α.{β}.β ] as β → end
```

Note that using `@>` it is possible to *resume* the interaction on an endpoint with type `end`. It may be useful to collect the results of the functions to which `@>` is applied. To this aim, the following generalization of `@>` is provided:

```
Session.(@=) : (A → α * end) → {A}.B → α * B
```

As an example, the function `receive_tree` reads a serialized tree back into its algebraic form

Ttree.ml:75-81

```
let rec receive_tree ep =
  match Session.branch ep with
  | `Leaf ep → Leaf, ep
  | `Node ep → let v, ep = Session.receive ep in
               let l, ep = receive_tree @= ep in
               let r, ep = receive_tree ep in
               Node (v, l, r), ep
```

and has type:

```
receive_tree : &[ Leaf: end | Node: ?α.{β}.β ] as β → α tree * end
```

It should be noted that `@>` and `@=` do not perform any communication or synchronization in addition to those entailed by the session types A and B. Also, the type constructor used to denote sequential composition is abstract and does not interact with any other session type. In particular, `!α.!β.end` and `{!α.end}.!β.end` are different session types that cannot be interchanged, even though they result in exactly the same sequence of observable operations.

4 Other features

4.1 Service channels

In all the examples discussed so far we have used the `Session.create` primitive to establish a new session. This primitive returns both endpoints of the session, which must be distributed to a corresponding pair of threads in order for communication to take place. `FuSe` provides an alternative API for establishing new sessions based on *service channels*. Unlike session endpoints, these channels can be distributed and aliased without restrictions, as their only purpose is that of establishing new sessions and not to realize structured communications.

A service channel has type

```
T Service.t
```

and represents a service that accepts incoming connections and behaves according to T on the corresponding sessions. A new service channel is created using the primitive

```
Service.create : unit → A Service.t
```

and availability to accept connections must be explicitly signalled by means of the blocking primitive

```
Service.accept : A Service.t → A
```

which returns one endpoint of the session established with a peer *requesting* the connection through the indicated service channel. Requests are signalled by clients using the primitive

```
Service.request : A Service.t → A dual
```

A frequent programming pattern involves a persistent service that waits for incoming connections on a service channel and spawns a copy of itself each time a connection is established. `FuSe` provides a convenient primitive to realize this pattern:

```
Service.spawn : (A → unit) → A Service.t
```

The primitive must be applied to a function representing the service body, which is spawned in its own thread at each connection and is applied to the corresponding session endpoint. `Service.spawn` returns a service channel that can be used by clients to request connections to such service. As an example, the code that connects `client` and `add_server` in Section 2.2 can be simply written as

Tadder.ml:38-39

```
let _ =
  print_int (op_client (Service.request (Service.spawn add_service)) 1 2)
```

4.2 Subtyping

FuSe supports the notion of subtyping for session types defined by Gay and Hole [2005]. This makes it possible to use an endpoint of type T wherever an endpoint of type S is expected if T is a subtype of S. In this tutorial, we will only introduce subtyping by examples. The interested reader may refer to [Gay and Hole, 2005] for its formal definition.

Technically, the properties of subtyping derive from the definition of the Session.st abstract type, which is reported below:

```
type (+α, -β) st
```

The variance annotations + and - specify that the Session.st type is *covariant* in the type of input messages and *contravariant* in the type of output messages. For example, given the following type definitions

```
type in_A   = &[A : end]
type in_AB  = &[A : end, B : end]
type out_A  = ⊕[A : end]
type out_AB = ⊕[A : end, B : end]
```

we have that in_A is a subtype of in_AB and out_AB is a subtype of out_A. The intuition follows the standard principle of *safe substitutability*: a well-typed code fragment using an endpoint ep of type in_AB must be able to cope with both A and B tags. By replacing ep with another endpoint ep' of type in_A, the same code fragment will keep working properly since only A tags can be received from ep'. The dual scenario holds for outputs. A well-typed code fragment using an endpoint ep of type out_A will only select A tags on ep. If ep is replaced with ep' of type out_AB, which allows both A and B tags to be selected, the same code fragment will continue to work properly.

One caveat of using subtyping in OCaml is that the endpoint must be explicitly coerced by the programmer. For example, if ep is of type in_A, then

```
(ep :> in_AB)
```

coerces ep to type in_AB, which is a supertype of in_A.

Subtyping also acts for the types of exchanged messages, following the usual covariance rule for inputs and contravariance rule for outputs. Therefore, given the type definitions

```
type in_in_A   = ?in_A.end
type in_in_AB  = ?in_AB.end
type out_in_A  = !in_A.end
type out_in_AB = !in_AB.end
```

we have that in_in_A is a subtype of in_in_AB and dually out_in_AB is a subtype out_in_A. As noted by Gay and Hole [2005], however, subtyping is *always covariant* with respect to the type of continuations. Therefore, given the type definitions

```
type in_int_in_A = ?int.in_A
type in_int_in_AB = ?int.in_AB
type out_int_out_A = !int.out_A
type out_int_out_AB = !int.out_AB
```

we have that in_int_in_A is a subtype of in_int_in_AB and out_int_out_AB is a subtype of out_int_out_A.

References

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

Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *Proceedings of CONCUR'11*, LNCS 6901, pages 280–296. Springer, 2011. doi: 10.1007/978-3-642-23217-6_19.

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

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

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

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.

Raymond Hu and Nobuko Yoshida. Hybrid Session Verification through Endpoint API Generation. In *Proceedings of FASE'16*, LNCS. Springer, 2016. To appear.

Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, LNCS 2757, pages 439–453. Springer, 2002. doi: http://dx.doi.org/10.1007/978-3-540-40007-3_26. Extended version availbale at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

Luca Padovani. A simple library implementation of binary sessions. Technical report, Università di Torino, 2015. Available at `https://hal.archives-ouvertes.fr/hal-01216310`.

Peter Thiemann and Vasco T. Vasconcelos. Context-Free Session Types. In *Proceedings of ICFP'16*, pages 462–475. ACM, 2016.

Jesse A. Tov and Riccardo Pucella. Stateful Contracts for Affine Types. In *Proceedings of ESOP'10*, LNCS 6012, pages 550–569. Springer, 2010. ISBN 978-3-642-11957-6. doi: 10.1007/978-3-642-11957-6_29.