Session Types without Sophistry
System Description

Oleg Kiselyov and Keigo Imai

[1] Tohoku University, Japan
[2] Gifu University, Japan

Abstract. Whereas ordinary types approximate the results, session types approximate communication among computations. As a form of type-state, they describe not only what is communicated now but also what is to be communicated next. Writing session-typed programs in an ordinary programming language such an OCaml requires inordinary cleverness to simulate type-level computations and linear typing – meaning the implementation and the error messages are very hard to understand. One is constantly reminded of template metaprogramming in C++.
We present a system exploring a very different approach to session typing: lowering type-level sophistry to ordinary programming, while maintaining the static assurances. Error messages are detailed and customizable, and one can use an ordinary debugger to investigate session-type problems. Our system is a binary-session–typed DSL for service-oriented programming in OCaml, supporting multiple communication channels, internal and external choices, recursion, and also channel delegation.
The key idea is staging: ordinary run-time checks in the generator play the role of "type-checks" from the point of view of the generated program. What is a fancy type to the latter is ordinary data to the generator.

1 Introduction

Whereas ordinary types approximate the results, session types approximate communication among computations. Session types [14, 47] are appealing because they can be inferred and statically checked, and because well–session-typed programs "do not go wrong": no two parties attempt to both read from or both write to their communication channel; no computation sends the data its party is not prepared to handle; no program tries to use closed or delegated away channels.[3] Therefore, there have been developed many session-typed communication libraries [21–23, 32, 35, 37, 39, 41–43]. They are, in essence, DSLs for process orchestration embedded in an extant mature programming language: data processing parts are programmed as usual; data communication, written via DSL operations, is guaranteed to obey the protocol.

On the other hand, session type systems needed for realistic service-oriented programs are substructural and rather complicated [1, 2, 6, 51], with type-level

[3] Binary session type systems like [14] and its successors, used in many libraries including ours, do not in general prevent deadlocks (see § 5).

computations to express duality, with resource-sensitivity, with extensible (type-level) record types and (equi-)recursive types. They are a poor match for the type system of the typical host language such as OCaml, Scala or Haskell, and hence have to be emulated, often with extraordinary sophistication, exploiting the (mis)features of the host type system to the full (see examples in §5). Although the emulation is possible – after all, the host type systems are Turing-complete – it often feels like programming an actual Turing Machine. Abstraction, error reporting and debugging are lacking. Linear types are a particular challenge [21, 39, 43]. The emulation invariably also affects end users: as complicated inferred types that quickly become unreadable [39]; as referring to channels by De Bruijn indices rather than by names [21, 24, 41, 42]; and especially as bewildering error messages should something go wrong [22, 41].

Having developed session-type libraries ourselves and become familiar with intricacies and frustrations of type-level programming, we cannot help but envy the ordinary term-level programming, which is actually designed for programming. We would like to:

- add a session-typed communication layer to an existing programming language, reusing all its libraries, tools and support;
- take a non-toy, off-the-shelf session-type system such as [51] essentially as it is;
- use the host language itself (rather than its type system) to implement the session-type checking and inference;
- statically guarantee that a well-sessioned program "does not go wrong";
- make error messages customizable and use the host language debugging facilities to debug session types problems.

We have built an embedded DSL satisfying all these desiderata, relying on staging, a form of metaprogramming. The key idea is type checking as a staged computation. Our contributions are as follows:

1. The DSL, called `<session>`, for service-oriented programming embedded in OCaml. It supports bidirectional communication on arbitrary many channels, internal and external choices, channel delegation, and recursion – over named FIFO pipes. Other back-ends such as UDP or HTTP can be easily added.
2. The showcase of using staging for embedding DSLs with sophisticated type systems, maintaining the static guarantees.
3. The showcase of implementing extensible DSLs. In fact, `<session>` is built by progressively extending the base DSL with choices and then delegation and recursion. Type-checking operations, in particular, unification, are likewise extensible. Extensible records with record subtyping is one of the extensions.
4. The example of using canonical structures (first proposed for Coq, see §4.2), e.g., to support communication for arbitrary many, user-defined types.

The next section presents `<session>` on a progression of examples, at the same time reminding of session type systems. §2.3 deals with errors and error

messages. More interesting details of `<session>` are shown in §3. We then expound two implementation techniques characteristic of `<session>`: staging in §4.1 and canonical structures in §4.2. (The lack of space precludes the description of extensible mutually recursive functions and unification: we refer to the source code and the comments therein.) §5 discusses the related work.

The complete code is available at the following URL:

`http://okmij.org/ftp/Computation/types.html#sessions`.

2 Session Types by Example

This section recalls the binary session types (specifically, Yoshida and Vasconcelos' liberal system [51, §3]), by example, using `<session>`. The section hence also serves as an introduction to `<session>`. Figure 1 presents the DSL in full (as an OCaml signature), which we will explain step by step.

Ordinary type systems such as the Hindley-Milner system and its variations deal with (potentially open) expressions, such as `x+1>y`. Assuming the free variables `x` and `y` have the type `int`, the type system judges the expression well-typed and infers its type as `bool`. The type is an approximation of the expression's result – computed statically, that is, before evaluating it. In fact, we cannot evaluate the sample expression by itself since it is not a complete program: it is open. In a sound type system, the type correctly approximates an expression's result (if it ever comes), from which follows that a well-typed program "does not go wrong". For example, we may use our sample expression in a conditional **if** `x+1>y` **then** ... **else** ..., without worrying what to do should `x+1>y` happen to return, say, a string.

2.1 Basic Communication

Session type systems deal not with expressions to evaluate but with communicating processes to run, such as the process

$$y_1?[z_1]\,\mathsf{in}\;\; y_2?[z_2]\,\mathsf{in}\;\; y_2![z_1{>}z_2];\mathsf{inact} \tag{1}$$

in the conventional process calculus notation, employed in [51]. This process has two communication channels, or, to be precise, *endpoints*,[4] y_1 and y_2, which are represented by free variables. (Our process is hence a mere process fragment; we complete it soon.) It is to receive a value on the endpoint y_1, bind it to the variable z_1, receive another value on y_2 binding it to z_2, and send on y_2 the result of the comparison of z_1 and z_2. After that, the process is finished.

Our `<session>` is a (Meta)OCaml library to write processes and orchestrate them. It represents a process – to be precise, a perhaps infinite *sequence* of computations and communications – as an OCaml value of the abstract type `th`

[4] What we call an endpoint, Yoshida and Vasconcelos [51, §3] call a "polarized channel", following Gay and Hole [12].

(named for "thread"). An endpoint is represented as a value of the type ep. The sample process (1) is written as[5]

```
let p1 y1 y2 =  recv y1 Int @@ fun z1 →
                recv y2 Int @@ fun z2 →
                send y2 Bool .< .~z1 > .~z2 >. @@
                finish
⇝ val p1 : ep → ep → th =  <fun>
```

(the last line shows the type inferred by OCaml for p1). We use OCaml's let-statement to assign the process a name for easy reference, and make explicit its free endpoint variables y1 and y2. Comparing the sample process in the two notations, (1) and ours, shows them quite similar. Our notation however clearly distinguishes the binding occurrences of z1 and z2 (and we write finish instead of "inact" for the ended process). Also explicit in the p1 code are Int and Bool, which may be regarded as type annotations on the communicated values. That these annotations are mandatory is a drawback of the embedding (although not that big), which we discuss in §4.2.

The p1 code also betrays staging. Staging is what MetaOCaml [25, 26] adds to OCaml: the facility to generate code to compile and execute later. To be precise, MetaOCaml adds the type α code for values representing the generated code, and two facilities to produce such values. One, akin to quote in Lisp, is enclosing an expression in so-called "brackets", for example: .<1 > 2>.. The bracketed expression is not evaluated; rather, it becomes (a fragment of) the generated code. The other facility, called "escape", is like Lisp unquote. It can be understood as poking a hole in a bracketed expression, turning it into a code template. In p1 code, .< .~z1 > .~z2 >. is such a template, with two holes to be filled by the code values bound to the variables z1 and z2 – producing the code of the comparison expression. Although bracketed expressions are not evaluated, they are type checked. For example, in order for .< .~z1 > .~z2 >. to be well-typed, with the type bool code, the variables z1 and z2 should be of the type int code – or a type error is raised. Thus MetaOCaml statically guarantees that the generated code is well-typed – and also free from scoping errors (like unbound or unintentionally bound identifiers): unlike Lisp quotations, MetaOCaml is hygienic. Staging is crucial in our approach to session typing, as detailed in §4.1. Staging also lets <session> distinguish process computations (which are put in brackets) from process communications (described by the combinators such as recv and send). Thus <session> is a DSL for orchestration.

Type-checking a <session> expression in OCaml gives its OCaml type that says nothing about communication (see, for example, the type of p1). Evaluating the expression gives its session type (as well as the code to run, to be discussed in §2.2); the error case is detailed in §2.3. The expression p1 however is open (represents an incomplete process fragment) and cannot be evaluated. We can still get its session type, by evaluating infer Fun(EP,Fun(EP,TH)) p1, which supplies

[5] The right-associative infix operator @@ of low precedence is application: f @@ x + 1 is the same as f (x + 1) but avoids the parentheses. The operator is the analogue of $ in Haskell.

(a) Types

type proc	top-level process
type th	communication thread
type ep	session endpoint
type shared	shared name, e.g., host:name

(b) Basics

```
val proc   : th → proc
val (||)   : proc → proc → proc

val new_shared : string → shared

val request : shared → (ep → th) → th
val accept  : shared → (ep → th) → th
val send    : ep → α trep → α code → th → th
val recv    : ep → α trep → (α code → th) → th

val othr    : unit code → th → th
val let_    : α code → (α code → th) → th
val finish  : th
```

(c) Debugging, logging, etc

```
val describe_ep : ep → string code
val describe_sh : shared → string code
val debuglog    : string → th → th
```

(d) Inference, execution, deployment

```
val infer       : α trep → α → string
val proc_run    : proc → unit
val proc_deploy : proc → unit code list
```

(e) Internal and external choices

```
type label =  string
val branch : ep → (label * th) list → th
val select : ep → label → th → th
val ifte   : bool code → then_:th → else_:th → th
```

(f) Delegation

```
val deleg_to   : ep → ep → th → th
val deleg_from : ep → (ep → th) → th
```

(g) Iteration

```
val toploop : (th → th) → th
val loop : ep list → (th → th) → th
```

Fig. 1. The syntax of `<session>`, as OCaml module signatures

(a) Environments

```
type envd   (in text, Δ)    Linear environment: finite map from ep to sess
type envg   (in text, Γ)    Non-linear environment: finite map from shared to sess
```

(b) Type formulas, as an extensible data type sess

```
type sess =  ..
type sess +=  Var of var ref | End
```

(c) Basic communication extension

```
type sess +=  Send : α trep * sess → sess | Recv : α trep * sess → sess
```

(d) External and internal choices, based on row types [33]

```
type sess +=  Bra : rows → sess | Sel : rows → sess
and rows =
  | Row : (label * sess) * rows → rows
  | RowVar : rowvar ref → rows
  | RowClosed
```

(e) Delegation

```
type sess +=  DSend : sess * sess → sess | DRecv : sess * sess → sess
```

(f) Recursion

```
type sess +=  Mu : id * sess → sess | RecVar : id * bool(*dual*) → sess
```

Fig. 2. Session types (see the explanations text; trep will be explained in §4.2)

the two "assumed" endpoints, obtaining:[6]

```
ep_hyp-12/13        : Recv(int,End)                    (2)
ep_hyp-14/15        : Recv(int,Send(bool,End))         (3)
```

Unlike the ordinary type (which is a single formula), a session type is like an environment: a finite map from names to formulas, see Fig. 2. To be precise, a session type is a pair of environments: the linear Δ (which [51] calls "typing") and the non-linear Γ (called sorting in [51]). They are so named because in the system of [51], endpoints are to be used linearly, but shared points, discussed later, do not have to be. Shown above is the linear environment inferred for p1 (Γ is empty). The environment specifies the communication pattern for the two endpoints of p1 in order for it to be well-sessioned (the concrete names for those hypothetical endpoints, correspond to the free variables y1 and y2, are made up by infer).

Session types (environments and type formulas) are the ordinary data types in <session>. The type formulas are an extensible data type, because we keep extending the syntax of formulas as we add more features to <session>. The

[6] It should also be possible to supply a session type and check an expression against it, to verify its communication obeys the protocol stated in the type. After all, if we can infer a session type, we can check against it. However, we have not yet offered this facility in the public library interface.

type formulas describe the communication protocol: the approximation, or pattern, of the actual communication over a channel (endpoint). End is the end of interactions; Send(t,s) means sending a value of the type t (represented as "type representation" data type trep, see §4.2) with further interactions being described by s. Recv(t,s) is the protocol of receiving a value of the type t and then continuing as s. Thus, the process fragment p1, according to its inferred session type, communicates on two endpoints. From one endpoint, (2), it reads an integer and closes it; for the other, (3), it reads an integer, then sends a boolean and closes.

2.2 Sessions

A session, whose type we have just discussed, is a series of interactions between two parties over a channel. (This paper deals only with binary sessions.) A session begins when two parties rendez-vous at a "common point" and establish a fresh channel; it concludes when the communications over the channel end (as we will see, `<session>` detects the end as part of the session type inference, and automatically arranges for closing the channel and freeing its resources.) The rendez-vous point is called shared in `<session>`, created on the base of a name, such as a host name, known to all parties. The exact representation of shared depends on the underlying low-level communication library: for a TCP/IP back-end, shared may be a socket_addr; for the FIFO pipe backend, shared is represented by two (unidirectional) pipes, whose names are derived from the supplied known name. There may be many rendez-vous at the same shared – all of which, however, establish channels with the same protocol. This is the basic assumption of structured communication behind session type systems. Therefore, shared itself may be assigned a session type, describing the common protocol of these channels.

A rendez-vous is performed when one process executes accept and the other request, see Fig.1, on the same shared. (In TCP/IP terms, when one process "connects" and the other "accepts" the connection.) As the result, a fresh private communication channel is created; each of the two processes receive the respective endpoint of it and can start communication.

To complete our running example p1 we create two channels, in two consecutive rendez-vous on two different shared:

```
let a =  new_shared "sha" and b =  new_shared "shb"
let pc =  request b @@ fun y2 → accept a @@ fun y1 → p1 y1 y2
```

Why one shared or one channel would not suffice is discussed in §2.3; on the other hand, which operation to use, request or accept, is arbitrary at this point of developing the example. The party communicating with pc is the process q:

```
let q =
  accept b  @@ fun x2 →
  request a @@ fun x1 →
  send x1 Int .<1>. @@
  send x2 Int .<2>. @@
  recv x2 Bool @@ fun z →
```

```
  othr .<Printf.printf "got␣%b\n" .~z>. @@
  finish
```

Since q is meant to communicate with pc, the choice of accept and request is no longer arbitrary. The operation othr lets us perform computations other than communication, specified as an arbitrary OCaml code enclosed in brackets. In case of q, this computation is printing, of the received value.

Both pc and q have no free endpoints and can be regarded as "top-level processes": cast as proc. Top-level processes can be combined to run in parallel:

```
let r = proc pc || proc q
```

The inferred session type of proc pc is

```
sh>sha-49          : Recv(int,End)                    (4)
sh>shb-50          : Send(int,Recv(bool,End))         (5)
```

which is the non-linear environment Γ for proc pc; as top-level processes have no free endpoints, the linear environment Δ is always empty. The environment Γ associates shared with session types. Process proc pc rendez-vous on two shared, which hence show in the printed Γ. Here, `sh>sha-49` is the internal identifier for the shared point with the name "sha" created earlier, and similar with for "shb". Comparing (4) with the earlier (2) illustrates what we have explained already: the session type of a shared is the session type of channels created at its rendez-vous. However, (5) and (3) are not the same: they look "symmetric", or dual. Indeed, when two processes communicate over a channel, one sends and the other receives. Thus the session types of two endpoints of the same channel have to be dual. The session type of a channel is taken to be the session type of the ep of the accept-ing process – or the dual to the session type of the ep of the request-or.

The inferred session type, or Γ, for proc q is the same as for proc pc ((4) and (5)), which means the parallel composition r is well-sessioned. When pc is sending an integer, q will be waiting to receive it. Evaluating r does more than just the session type inference and checking. We also get the code for the processes to run in parallel. The top-level r is the parallel composition of two complete th, and hence two pieces of code are produced. Here is the first one, corresponding to proc pc:[7]

```
1  .<let lv_78 = {sh_arname = "/tmp/SHshb−50.fifo"; sh_name = "shb−50"} in
2   let lv_77 = {sh_arname = "/tmp/SHsha−49.fifo"; sh_name = "sha−49"} in
3   let rawep_79 = sh_request lv_78 in
4   let rawep_80 = sh_accept lv_77 in
5   let x_81 = int_of_string (ep_read rawep_80) in
6   ep_close rawep_80;
7   (let x_82 = int_of_string (ep_read rawep_79) in
8    ep_write rawep_79 (if x_81 > x_82 then "T" else "F");
9    ep_close rawep_79;
10  ())>.
```

[7] To improve readability, we adjusted indentation and removed module references, while the rest is left as-is. Variables lv_77 and lv_78 are generated via let-insertion.

Clearly seen are the calls to the low-level communication library, as well as the serialization/deserialization code such as int_of_string, converting sent and received values to/from strings, that is, the sequence of bytes to exchange over the channel. The serialization/deserialization code is generated by <session>.

Lines 5 through 10 are the code generated for the process fragment p1, with the variable rawep_80 standing for y1 and rawep_79 to y2. Noticeable are the ep_close calls to close and deallocate the endpoints, which were not present in p1. A call to close an endpoint is inserted as part of session type inference, as soon as it is determined that the endpoint's communication is complete. For example, when the inferred session type of p' in recv y1 Int @@ p' does not mention y1, this endpoint can be closed right after recv completes. Just as the automatic memory management, the automatic endpoint management eliminates the class of subtle bugs, as well as relieving the programmer of a chore.

The generated code for the processes can be extracted by proc_deploy (see Fig.1), stored into a file, compiled and then deployed on communication nodes. Alternatively, <session> provides proc_run to run the generated code as separate (fork-ed) processes, for testing. One may do make tests to test-execute r (and all other examples that come with <session>.)

2.3 What If One Makes a Mistake: OCaml types v. Session types

There are many opportunities for mistakes. This section shows what happens if we make some of them. After all, detection and reporting mistakes is the main reason to use a type system in the first place.

Some mistakes are caught already by the OCaml type checker, for example:

```
let p1' y1 y2 =  recv y1 T.Int @@ fun z1 → recv y2 T.Int @@ fun z2 →
  send y2 T.Bool .<~z1 > ~z2>.
  ^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: This expression has type th → th but an expression was expected of type th
```

```
let p1'' y1 y2 =  recv y1 T.Int @@ fun z1 →recv y2 T.Int @@ fun z2 →
  send y2 T.Bool .<~z1 + ~z2>. @@ finish
                  ^^^^^^^^^
Error: This expression has type int but an expression was expected of type bool
```

with the detailed error message. The type errors mean that p1' and p1'' are not well-formed processes.

In §2.2 we have created two shared for pc, saying that one would not suffice. Let us see what happens if we do use only one shared (e.g., by mistake):

```
let a =  new_shared "sha"
let pc1 =  request a @@ fun y2 → accept a @@ fun y1 → p1 y1 y2
```

This code type-checks in OCaml, meaning it is a well-formed <session> expression. It is legitimate, after a rendez-vous on a shared do another rendez-vous on

the same shared, with the same or different process[8] – but not in the case of p1. All channels created on the same shared should be used with the same communication protocol. However, a glance at (2) and (3) tells the two endpoints of p1 are used rather differently. Therefore, pc_1, albeit a well-formed <session> expression, is not well-sessioned. Indeed, its evaluation ends in an exception that the types Send(int,Recv(bool,End) and Recv(int,End), inferred separately for shared a, are not unifiable. Session-typing problems are reported as exceptions, carrying the problem descriptor (e.g., unification failure) and the details (the non-unifiable types themselves) – so that one may print a custom error messages upon catching it, along with the backtrace. Standard tools like Emacs or other IDEs understand such backtraces, thus allowing to investigate the problem.

When the session-type inference succeeds, it returns the session type plus the code generated for the process (which can be extracted, for the complete process, with proc_deploy). The evaluation of pc_1 ended in an exception, and hence no code has been generated. Contrapositively, if the code is successfully generated, it represents a well-sessioned process. This is a *static* guarantee, from the process point-of-view – we know the process shall obey the protocol before running its generated code.

When defining the fragment p1 in §2.1 we meant it to communicate on two endpoints, denoted by the variables y1 and y2. Nothing stops the caller of p1, however, from supplying the same endpoint value for both variables (i.e., make y1 and y2 alias the same endpoint):

```
let pc2 =  request b @@ fun y2 → p1 y2 y2
```

This time evaluating pc_2 produces no errors: after all, it expresses a legitimate communication behavior – only not the intended one and not corresponding to the process q. Therefore, evaluating proc pc_2 || proc q raises an exception that the two processes make non-unifiable assumptions about the protocol associated with "shb", namely, Send(int,Recv(bool,End)) vs. Recv(bool,End). One can then look closely into the inferred session types for pc_2 and its fragments, possibly using the OCaml debugger, identifying the source of the problem.

The process q defined in §2.2 accepted on shared point b and requested on a; therefore, its party pc should first request on b and accept on a. It is very easy to confuse the two operations and write

```
let pc3 =  accept b @@ fun y2 → request a @@ fun y1 → p1 y1 y2
```

Although this code defines a legitimate process, it cannot be a party to q. Therefore, proc pc_3 || proc q raises an exception that the two processes make different assumptions about the protocol of the shared b, viz., Recv(int,Send(bool,End)) and Send(int,Recv(bool,End)). The two inferred types look dual, which is a hint at a request/accept confusion.

Thus, when a <session> expression passes the OCaml type check, the computations within the corresponding process are well-typed and "won't go wrong",

[8] The code does not say that pc_1 rendez-vous with itself, which is impossible. Communications on a shared are synchronous (i.e. request and response blocks until its counterpart becomes available) while those on session endpoints are asynchronous.

```
1  let srv ep =
2    loop_with_val (.< 0 >., [ep]) @@
3      fun continue acc →
4    branch ep
5    ["add",
6       recv ep T.Int @@ fun x →
7       let_ .<.~x + .~acc>. @@ fun acc →
8       send ep T.Int acc @@
9       continue acc
10    "quit",
11      send ep T.Int acc  @@
12      finish
13   ]

14 let cli ep =
15   select ep "add" @@
16   send ep T.Int .<1234>. @@
17   recv ep T.Int @@ fun acc0 →
18   select ep "add" @@
19   send ep T.Int .<5678>. @@
20   recv ep T.Int @@ fun acc1 →
21   select ep "quit" @@
22   recv ep T.Int @@ fun ans →
23   othr .<printf "sum:␣%d\n" .~ans>. @@
24   finish

25 let p = let sh = new_shared "sh" in proc (accept sh srv) || proc (request sh cli)
```

Fig. 3. Example: An arithmetic server

and the process itself is well-formed ("syntactically correct"). Further, when the `<session>` expression successfully evaluates, it produces the code for the process, whose computations and, in addition, communications are statically assured to do no wrong.

3 Elaborate Examples: Choice, Recursion and Delegation

Arithmetic Server. Figure 3 shows a more interesting example with external and internal choices and recursions. This is the standard example of the so-called "arithmetic server", which is common in literature. Function srv is a server which takes an endpoint ep and iterates over a loop via construct loop_with_val. The loop construct is supplied a pair of the initial value .< 0 >. of an accumulator and the endpoint [ep] which are used in the following body of iteration. It binds itself to variable continue and the accumulator to acc. Note that the construct itself does *not* iterate but just produce the code for iteration. The loop body offers two labels "add" and "quit" via external choice construct branch. Here, labels represented by two strings then become part of the type for external choice, which is an ordinary runtime value in (Meta)OCaml. In the "add" branch, an integer is received, bound to x and added to the accumulator. The result is rebound to acc and sent back to the client. The server then recurs (with the updated acc) to handle further requests. On "quit", the server reports the accumulator to the client and terminates. Client's function cli should be understood similarly.

Compatibility of srv and cli is checked at Line 25. Thanks to equi-recursive nature of session types, this program actually typechecks. The example exhibits a form of *session subtyping* in `<session>` implemented via *row types*, following the Links language [5]. By evaluating infer Fun(EP,TH) srv, we get the following type which describes the server's protocol:

```
Mu(18,Bra(quit:Send(int,End)+add:Recv(int,Send(int,RecVar18))+RClosed))   (6)
```

Session Mu(id, t) denotes a (equi-)recursive type, with RecVarid bound to the whole Mu(id, t) expression. Bra(l_1:t_1+...+l_n:t_n+RClosed) shows an external choice among labels $l_1, \ldots, l_n$ where t_i describes communication after l_i is chosen. RClosed in the end says that the choice is *closed*, disallowing other labels. In total, the above session type (correctly) specifies the recursive behavior of the server with two operations quit and add. Similarly, infer Fun(EP,TH) cli yields the client's type which is dual to the type above:

```
Sel(add:Send(int,Recv(int,
  Sel(add:Send(int,Recv(int,                                    (7)
    Sel(quit:Recv(int,End)+RMeta21)))+RMeta22)))+RMeta23)
```

Note that the type (7) does not show any recursive structure as well. Session Sel(l_1:t_1+...+l_n:t_n+RMetaid) is an internal choice, where RMetaid is a *row variable* which can contain more alternatives, enabling session subtyping. The type unification invoked by (||) at Line 25 confirms that the session (6) and (7) are *dual* to each other; thus the programmer can conclude cli and srv have no deadlock, in an *earlier* stage. Moreover, such type features come without annotations like enter in [41] (see § 5), thanks to the flexibility of metaprogramming.

```
1  let agency agc_ch svc_ch =
2    accept agc_ch @@ fun cus_ep →
3    loop [cus_ep] @@ fun continue →
4      branch cus_ep
5      ["quote", begin
6         recv cus_ep String @@ fun dest →
7         send cus_ep Int .< 350 >. @@
8         continue
9       end;
10      "accept", begin
11         request svc_ch @@ fun svc_ep →
12         deleg_to svc_ep cus_ep @@
13         finish
14      end]
15
16 let service svc_ch =
17   accept svc_ch @@ fun svc_ep →
18   deleg_from svc_ep @@ fun cus_ep →
19   recv cus_ep String @@ fun address →
20   send cus_ep
21      String .< "2020−04−01" >. @@
22   finish
23
24 let customer ch =
25   request ch @@ fun ep →
26   loop [ep] @@ fun continue →
27     select ep "quote" @@
28     send ep String
29        .< "Tokyo_to_Akita" >. @@
30     recv ep Int @@ fun cost →
31     ifte .< .~cost < 400 >. ~then_: begin
32       select ep "accept" @@
33       send ep String
34          .< "Tokyo,_JP" >. @@
35       recv ep String @@ fun date →
36       finish
37     end
38       ~else_: continue
```

Fig. 4. Example: Travel Agency

Example with Delegation. Delegation allows one to pass a session-typed channel to another peer, enabling dynamic change of the communication topology in a system[9]. Figure 4 is the Travel Agency example from [21] (originally in [18]). The scenario is played by three participants: customer, agency and service. Process

[9] Ours deleg_from and deleg_to are called throw and catch in [51] (we changed the names to avoid association with exceptions)

customer knows agency while customer and service initially do not know each other, and agency mediates a deal between customer and service by delegation. We use accumulator-less loop combinator in this example. Upon quote request from customer, agency replies a rate (fixed to 350 for simplicity) and re-starts from the beginning, and if customer agrees on the price (label "accept"), agency delegates the rest of the session to service in Line 12 using deleg_to. Process service accepts the delegation in Line 18 using deleg_from, and consumes the rest of session by receiving the delivery address (of type string) and then sending the delivery date ("2020−04−01"). Note that the original OCaml implementation in [21] uses *lenses* to convey delegated (linear) variables in types, while we use an ordinary, term-level variables, resulting in less complication in (OCaml) types.

Note also that there is a subtle difference in *ownership* control of session type systems [14, 51] from usual notion of *linearity*. That is, some primitives assume implicit *presence* of End types in continuation — for example, send ep 350 finish has Send(Int,End) in ep — while the delegation requires *absence* of a session in continuation, as in the end of agency above. To avoid such ambiguity, implementations (e.g. [23, 32, 39]) usually require the channel to be explicitly closed in the end of a session, while `<session>` does not demand such annotation.

4 Notable Implementation Techniques

4.1 Staging

We now describe and justify staged embedded DSLs as an implementation technique of supporting session- and other advanced type systems in an existing (staged) language.

Session types with no safety or usability compromises call for a language system designed for them, e.g., Links [5], which offers session types natively. Achieving this golden standard, and implementing and supporting a programming language requires time, effort and investment beyond the reach of many. Not only one has to implement a type checker, but also the whole compiler – as well as libraries, tools, build systems. One has to maintain them, write documentation, advertise and build community.

DSLs embedded in a mature, well supported host language are an attractive alternative. The host language provides the compilation, infrastructure, community – letting the DSL author concentrate on expressing domain-specific constructs and types as terms and types of the host language. The first problem comes when the DSL type system significantly differs from that of the host language – which is the case of session types, the form of *type-state* [46]. It requires advanced, modal or substructural type systems [7, 19, 49], rarely offered by a host language. One has to resort to emulation, whose problems we detailed in the Introduction. The main problem, for the implementor, is that type systems are rarely designed for writing code. Using a host type system as a programming language in which to emulate an advanced DSL type system is excruciating.

Staging helps, by letting the DSL implementor map DSL constructs *and* DSL types to host language terms. Whatever the DSL type checking and inference is

needed, can be programmed in the host language itself (rather than in its type system). That seems inadequate as DSL type errors will be reported too late: not when compiling a DSL program but when running it. One has to remember, however, that with staging, there are two (potentially more) run-times: the run-time of the code generator and the run-time of the generated code. It is the latter that corresponds to the traditional run-time, and which "should do no wrong". The run-time of the generator, from the point of view of the generated program, is a sort of "compile-time". Run-time errors in the generator are akin to the traditional type-error and compiler diagnostics: an indication that a compiler gave up on the source program and produced no object code. On the other hand, when the code is generated, one has the confidence it has passed the checks of both the host and the DSL type systems.

We now show a concrete illustration of the approach. Since `<session>` is rather advanced, we use a similar but simpler example, also featuring type-state: a DSL with operations to open, write to and close an arbitrary number of communication channels – and the type system that prevents using a channel after it has been closed. The manual closing of channels allows for more accurate and timely management of scarce resources than achievable with, say, region discipline. This is the example described in [29, §6]. Although seemingly simple, embedding this DSL in Haskell required heavy and unwieldy type-level programming, with the predictable result of large inferred types, fragile inference and confusing error messages [29, §6.2].

Let us see if we can do better with staging. Fig. 5 presents the interface, sample code, and most of the implementation (see the accompanying source code for full details and more examples.)[10] Most operations should be self-explanatory. The left-associative // is the "semicolon", to compose DSL expressions. The main assumption is the factoring of the DSL into communication (channel operations) and computations. The latter are represented as string code whereas the former are as values of the type comm. Such a factoring is common: monadic IO in Haskell, Lwt and Async libraries in OCaml are just a few other examples.

The sample DSL expression p, when evaluated, produces the expected code of opening, writing to, and closing output channels. If instead of ch1 we close ch2, the evaluation of p ends with a UsedAfterClose exception mentioning the offending channel – and produces no code.

The key is the realization of comm as an "annotated code": a record carrying the code generated for the DSL expression. The field c_chan of the record is the annotation: the DSL type associated with the code. As in `<session>`, it is a finite map (implemented with OCaml's Stdlib.Map) of channel ids ch_id and their statuses Closed or Active. The close operation generates code to close the channel – and the annotation that the channel, which should be active before, becomes Closed. Likewise, the write operation annotates the channel writing code with the fact that the channel was and to remain Active. The composition c1 // c2 merges not only the code but also the annotations, thus inferring the

[10] The language is quite like the STATE language in [27, §7]: the imperative part of Reynolds' Idealized Algol, as pointed out by Bob Atkey. Instead of var we write ch.

Interface

```
type comm   type ch

val (//) : comm → comm → comm
val skip  : comm

val close : ch → comm
val write : ch → string code → comm
val open_ :
    string → (ch → comm) → comm
val if_   : bool code → then_:comm → else_:comm → comm
```

Sample code

```
let p =
  open_ "/tmp/a1" @@ fun ch1 →
  open_ "/tmp/a2" @@ fun ch2 →
  write ch1 .<"s1">.  //
  close ch1 //
  write ch2 .<string_of_int 5>. //
  close ch2
```

Implementation: types

```
type ch_id =  string
type ch_status =  Closed | Active
type ch =
  {chch: out_channel code; chid: ch_id}
module M =
  Map.Make(struct type t =  ch_id let compare =  compare end)
type styp =  ch_status M.t
type comm =  {c_code: unit code; c_chan: styp}
```

"Type" errors

```
exception NotClosed of ch_id
exception UsedAfterClosed of ch_id
exception ClosedOnlyInOneBranch of
            bool * ch_id
```

```
let skip =
  {c_code =  .<()>.;
   c_chan =  M.empty}

let close =  fun {chch;chid} →
  {c_code =  .<close_out .~chch>.;
   c_chan =  M.singleton chid Closed}

let write =  fun {chch;chid} str →
  {c_code =  .<output_string .~chch .~str>.;
   c_chan =  M.singleton chid Active}
```

```
let (//) =  fun c1 c2 →
  let c_code =
      .< .~(c1.c_code); .~(c2.c_code) >. in
  let merger chid c1t c2t =
    match (c1t,c2t) with
    | (None,ct) | (ct,None) → ct
    | (Some Active, ct)     → ct
    | (Some Closed, Some _) →
        raise (UsedAfterClosed chid) in
  let c_chan =
    M.merge merger c1.c_chan c2.c_chan
  in {c_code;c_chan}
```

Fig. 5. The writeDSL: interface, sample code, implementation

DSL type (channel statuses) for the composed expression. The merging is done by Stdlib.Map.merge operation, with merger determining which associations from the input maps get to the output map, and how to deal with merge conflicts. If a channel remains active after c1, its status in c1 // c2 is determined by its status in c2. On the other hand, if the channel is Closed in c1 and yet appears in c2's annotation, it is the "use after close" error, and reported by throwing an exception.

4.2 Canonical Structures

To put it simply, Canonical Structures is a facility to obtain a value of a given type – for example, a value of the type int→string, that is, the function to

"show" an integer. Since there are many such functions, the user has to register the "canonical" value of this type. In the simplest case, searching for a canonical instance is a mere look up in the database of registered values. Instead of a canonical value itself, however, the database may provide a rule how to make it, from some other registered values (e.g., how to "show" a pair if we can show its components). Querying this database of facts and rules is quite like the evaluation of a Prolog/Datalog query.

From the point of view of the Curry-Howard correspondence, finding a term of a given type is finding a proof of a proposition. This is how this facility was developed in Coq, as a programmable unification technique for proof search, as expounded in [34].[11] Our implementation, inspired by that remarkable paper, is an attempt to explain it in plain OCaml, experiment with and use beyond Coq.

The rudiment of canonical structures is already present in OCaml, in the form of the registry of printers for user-defined types. It is available only at the top-level, however, and deeply intertwined with it. We have implemented this facility for all programs, as a plain, small, self-contained library, with no compiler or other magic. It can be used independently from `<session>`. Unlike the OCaml top-level–printer or Haskell type-class resolution, searching for a canonical instance is fully user-programmable. One may allow "overlapping instances", or prohibit them, insisting on uniqueness. One may allow for backtracking, fully or in part.

In `<session>` the canonical structures are used to look up the code for serializers and deserializers, to print types, and to implement infer to infer session types of process fragments with an arbitrary number of free endpoint variables.

Our implementation of Canonical Structures is user-level. Therefore, the look up of canonical values happens at run-time – rather than at compile time, as in type-class resolution. The look-up failures are also reported at run-time. It should be stressed, however, that in `<session>`, Canonical Structures are used only during code generation. The run-time errors at that point are run-time errors in the generator. From the point of view of the generated code, these are "compile-time" errors. Therefore, Canonical Structures in metaprograms roughly correspond to type classes in ordinary programs.

Since our Canonical Structures are implemented completely outside the compiler, the types of values to look up have to be explicitly specified as values of the α trep data type, which represents types at the value level. For example, a value Fun(Int,Bool) represents the type int→bool (and itself has the type (int→bool) trep). The data type can be easily extended with representations of user-defined data types (the `<session>` code shows a few examples). The trep values may be regarded as type annotations; in particular, as with other type annotations, if the user sets them wrong, the type error is imminent. Therefore, they are not an additional source of mistakes, but still cumbersome. If a compiler could somehow "reflect" an inferred type of an expression and synthesize a trep value, these annotations could be eliminated. We are contemplating how

[11] That tutorial paper also compares canonical structures to related approaches, in particular, implicits and type classes.

such reflection facility could be supported by OCaml, taking inspiration from the run-time-type proposal [13] and type-level implicits proposals [10, 50].

5 Related Work

The session type system employed in `<session>` is essentially the same as the liberal system [51, §3]. However, we distinguish threads `th` and endpoint-closed top-level processes. Only the latter may be parallel-composed. The reason is not of principle but practicality: web application and other such services do not spawn processes at will but rely on a worker pool, for better control of resources.

Links [5, 33] has session types on top of linear types and row polymorphism. Its core calculus GV [31, 48] has stronger properties like global progress, determinism, and termination, while [51] can lead to a deadlock with two or more sessions. We chose [51] as it has more liberal form of parallel composition. Adopting our approach to GV (and extending to exception handling [9]) is future work.

Several implementations have been done in Haskell [22, 32, 35, 37, 41, 42] and compared in [38] in detail. They are also established in Rust [23] (using its substructural types) and Scala [43] (based on dynamic linearity checking).

Implementation of session types in OCaml, firstly done by Padovani [39] and then Imai et al. [20, 21], seems a touchstone to spread into wider range of programming languages since it does not have substructural types nor any *fancy* features like type classes or implicits. The key issues are (1) static checking of *linearity*, (2) inference of *dual* session types and (3) encoding of *branching labels*. For (1), static checking of linearity in [39] is based on a parameterised monad of [41]. Imai et al. [21] provides a handy way to operate on multiple sessions using type-level indexes encoded by polymorphic *lenses* [8, 40], based on the idea by Garrigue [11, 20]. However, it requires much elaboration on types; for example, the type signature of the `send` primitive involves *six* type variables because of index-based manipulation for linearity and partially due to polarity encoding, which we will explain in the following (2).

For duality (2), there is a subtle tension between type inference, readability of types, and type compatibility. Pucella and Tov [41] showed a manual construction of duality witness in various languages including OCaml, while it can be automatically generated by type classes (and type functions [28]) in Haskell. On the other hand, Padovani adopts an encoding into i/o types by Dardha et al [6], achieving duality inference by OCaml's typechecker, which is also applied by Scalas et al. in Scala [43]. Dardha et al's encoding, however, is quite verbose, to the point that the resulting session types are hard to understand for humans (for details, see [21, § 6.2]). To mitigate it, the implementation of [39] provides the *type decoder*. Imai et al. resolved it by having *polarities* in types, however, it introduces complication on types, as we mentioned above. Furthermore, the polarity-based encoding has a type compatibility issue in delegations [21, in the end of § 3.3]. Summarizing the above, duality encoding in types has problems of (a) manual construction, as in [20, 41], (b) type decoder [39] or (c) compatibility problem [21], while our `<session>` does not have such problems at all.

Furthermore, duality is not just a swapping of output and input when a recursion variable occurs in a carried type, as pointed out independently by Bernardi et al. [2] and Bono et al. [3] which is usually overlooked (see [38, § 10.3.1]). Instead, we use $\overline{\mu\alpha.T} = \mu\alpha.\overline{T[\overline{\alpha}/\alpha]}$ in the Links language [33, § 12.4.1].

Type-level branching labels (3) are another obstacle for having session types in languages like Rust and Scala (e.g. [23, 43]) from which our approach does not suffer, as we have labels at the ordinary, term-level.

Hu et al. [18] showed a binary session extension to Java, *SessionJava*, including syntax extensions for protocols and session-based control structures. By contrast, `<session>` implements binary session types as a library on top of MetaOCaml, using only standard staging features like brackets and escape. Their work also includes *session delegation protocol* over distributed environment, which is orthogonal to the syntax and can possibly be added to `<session>`.

Scribble [44] is an implementation of *multiparty* session types [15] in various programming languages via code generation, including Java [16, 17], Go [4], and F# [36]. Multiparty session types take a top-down approach to generate session types from a global description of protocol called *global type*. On the other hand, Lange et al. [30] directly verifies session types via model checking. Extending `<session>` to the multiparty setting is future work.

6 Conclusions

We have presented the session-typed DSL `<session>` for service-oriented programming embedded in MetaOCaml. It was an experiment to see how the "type checking as staging" idea really works in practice, for a non-trivial, type-state–based type system and a non-trivial DSL. Overall, we are satisfied with our implementation experience: we have provided the same or even stronger guarantees than the other, mainstream implementations; we emit helpful error diagnostics; and we enjoyed programming in a mature implementation language rather than in a bare Post system. There is room for improvement (such as the `trep` annotations discussed in §4.2), and we are considering proposals to OCaml developers.

We have not yet implemented session-type annotations – that is, define the protocol as a session type, and then check that a process satisfies it. However, this is easy to add. We also want to extend our approach to group communication and multiparty session types.

The topic of this paper has been implementing session-type DSLs rather than developing session type systems themselves. Nevertheless, `<session>` turns out a good tool to prototype variations and extensions of session types. In the future work we plan to investigate one such extension: cancellation and failure modes.

Acknowledgments We thank anonymous reviewers for many, helpful comments and suggestions. This work was partially supported by JSPS KAKENHI Grant Number 18H03218 and 17K12662.

Bibliography

[1] Bernardi, G., Dardha, O., Gay, S.J., Kouzapas, D.: On duality relations for session types. In: Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers. pp. 51–66 (2014), https://doi.org/10.1007/978-3-662-45917-1_4

[2] Bernardi, G., Hennessy, M.: Using higher-order contracts to model session types. Logical Methods in Computer Science 12(2) (2016), https://doi.org/10.2168/LMCS-12(2:10)2016

[3] Bono, V., Messa, C., Padovani, L.: Typing copyless message passing. In: Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. pp. 57–76 (2011), https://doi.org/10.1007/978-3-642-19718-5_4

[4] Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed Programming Using Role Parametric Session Types in Go. In: 46th ACM SIGPLAN Symposium on Principles of Programming Languages. vol. 3, pp. 29:1–29:30. ACM (2019), https://doi.org/10.1145/3290342

[5] Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures. pp. 266–296 (2006)

[6] 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)

[7] Fluet, M., Morrisett, G., Ahmed, A.J.: Linear regions are all you need. In: Sestoft, P. (ed.) Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3924, pp. 7–21. Springer (2006)

[8] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3), 17 (2007), https://doi.org/10.1145/1232420.1232424

[9] Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. PACMPL 3(POPL), 28:1–28:29 (2019)

[10] Furuse, J.: Typeful PPX and Value Implicits. In: OCaml 2015: The OCaml Users and Developers Workshop (2015), implementation available at https://bitbucket.org/camlspotter/ppx_implicits

[11] Garrigue, J.: Safeio (a mailing-list post) (2006), available at https://github.com/garrigue/safeio
[12] Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005), https://doi.org/10.1007/s00236-005-0177-z
[13] Henry, G., Garrigue, J.: Runtime types in OCaml. In: OCaml 2013: The OCaml Users and Developers Workshop (2013), available at https://ocaml.org/meetings/ocaml/2013/proposals/runtime-types.pdf
[14] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: ESOP'98. Lecture Notes in Computer Science, vol. 1381, pp. 22–138. Springer (1998), https://doi.org/10.1007/BFb0053567
[15] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM 63(1), 9:1–9:67 (2016), http://doi.acm.org/10.1145/2827695
[16] Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. LNCS, vol. 9633, pp. 401–418. Springer (2016), http://dx.doi.org/10.1007/978-3-662-49665-7_24
[17] Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. LNCS, vol. 10202, pp. 116–133 (2017), https://doi.org/10.1007/978-3-662-54494-5_7
[18] Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: ECOOP'08. LNCS, vol. 5142, pp. 516–541. Springer (2008)
[19] Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. 27(2), 264–313 (2005)
[20] Imai, K., Garrigue, J.: Lightweight linearly-typed programming with lenses and monads. Journal of Information Processing 27, 431–444 (2019), https://doi.org/10.2197/ipsjjip.27.431
[21] Imai, K., Yoshida, N., Yuen, S.: Session-ocaml: a Session-based Library with Polarities and Lenses. Sci. Comput. Program. 172, 135–159 (2018), https://doi.org/10.1016/j.scico.2018.08.005
[22] Imai, K., Yuen, S., Agusa, K.: Session Type Inference in Haskell. In: Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. pp. 74–91 (2010), https://doi.org/10.4204/EPTCS.69.6
[23] 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), https://doi.org/10.1145/2808098.2808100
[24] Kiselyov, O.: Typed tagless final interpreters. In: Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures. LNCS, vol. 7470, pp. 130–174. Springer (2010), https://doi.org/10.1007/978-3-642-32202-0_3
[25] Kiselyov, O.: The design and implementation of BER MetaOCaml - system description. In: FLOPS. pp. 86–102. No. 8475 in Lecture Notes in Computer Science, Springer (2014)

[26] Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml approach. Foundations and Trends in Programming Languages, Now Publishers (2018)

[27] Kiselyov, O.: Effects without monads: Non-determinism – back to the Meta Language. Electronic Proceedings in Theor. Comp. Sci. 294, 15–40 (2019), https://arxiv.org/abs/1905.06544

[28] Kiselyov, O., Peyton Jones, S., Shan, C.: Fun with Type Functions. In: Roscoe, A.W., Jones, C.B., Wood, K. (eds.) Reflections on the Work of C. A. R. Hoare, pp. 301–331. Springer (2010)

[29] Kiselyov, O., Shan, C.c.: Lightweight monadic regions. In: Gill, A. (ed.) Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 1–12. ACM Press, New York (25 Sep 2008)

[30] Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. pp. 97–117 (2019)

[31] Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 560–584 (2015)

[32] 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), https://doi.org/10.1145/2976002.2976018

[33] Lindley, S., Morris, J.G.: Lightweight Functional Session Types (2017), In [45, § 12].

[34] Mahboubi, A., Tassi, E.: Canonical structures for the working coq user. In: Interactive Theorem Proving - 4th International Conference. Lecture Notes in Computer Science, vol. 7998, pp. 19–34. Springer (2013), https://hal.inria.fr/hal-00816703

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

[36] Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138. ACM (2018), https://doi.org/10.1145/3178372.3179495

[37] 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), https://doi.org/10.1145/2837614.2837634

[38] Orchard, D., Yoshida, N.: Session Types with Linearity in Haskell (2017), In [45, § 10].

[39] Padovani, L.: A Simple Library Implementation of Binary Sessions. Journal of Functional Programming 27, e4 (2016)
[40] Pickering, M., Gibbons, J., Wu, N.: Profunctor Optics: Modular Data Accessors. The Art, Science, and Engineering of Programming 1(2), Article 7 (2017), https://doi.org/10.22152/programming-journal.org/2017/1/7
[41] Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Gill, A. (ed.) Proceedings of the 1st ACM SIGPLAN Symposium on Haskell. pp. 25–36. ACM Press, New York (25 Sep 2008)
[42] Sackman, M., Eisenbach, S.: Session Types in Haskell: Updating Message Passing for the 21st Century. Tech. rep., Imperial College London (June 2008), http://pubs.doc.ic.ac.uk/session-types-in-haskell/
[43] 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), https://10.4230/LIPIcs.ECOOP.2016.21
[44] Scribble: Scribble home page (2019), http://www.scribble.org
[45] Simon Gay, A.R. (ed.): Behavioural Types: from Theory to Tools. River Publisher (2017), https://www.riverpublishers.com/research_details.php?book_id=439
[46] Strom, R.E., Yellin, D.M.: Extending typestate checking using conditional liveness analysis. IEEE Transactions on Software Engineering 19(5), 478–485 (May 1993)
[47] Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE'94 - Parallel Architectures and Languages Europe. Lecture Notes in Computer Science, vol. 817, pp. 398–413. Springer (1994), https://doi.org/10.1007/3-540-58184-7_118
[48] Wadler, P.: Propositions as sessions. J. Funct. Program. 24(2-3), 384–418 (2014)
[49] Walker, D., Crary, K., Morrisett, J.G.: Typed memory management via static capabilities. ACM Trans. Program. Lang. Syst. 22(4), 701–771 (2000)
[50] 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), https://doi.org/10.4204/EPTCS.198.2
[51] Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electr. Notes Theor. Comput. Sci 171(4), 73–93 (2007)