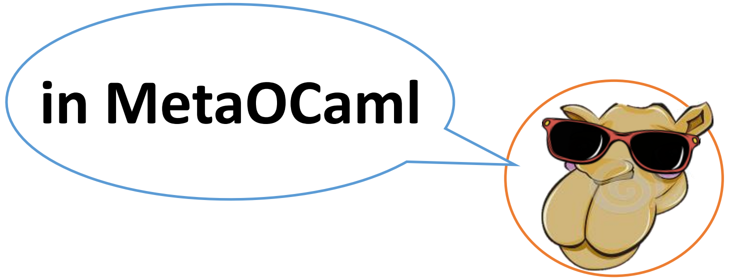
Session Types without Sophistry

System Description



Oleg Kiselyov

Tohoku University (JP)

Keigo Imai

Gifu University (JP)

15th International Symposium on Functional and Logic Programming

15th Sept. 2020

Introduction: <session>

A DSL for service orchestration embedded in OCaml

- Multiple bidirectional communication channels
- Internal and external choices
- Recursion
- Delegation

Static assuarances: well-session-typed programs "do not go wrong"

- Don't attempt to both read from or both write to a channel
- Obey communication protocol
- Don't use a closed or delegated away channel

(Deadlock-freedom is not guaranteed, for binary session types)

How do we differ?

State of the art

- Links language
- Embedded DSLs
  - Types are rather convoluted (still fun, but…)
  - Error messages are hard to analyse
  - Much like C++ template metaprogramming / Turing Machine programming

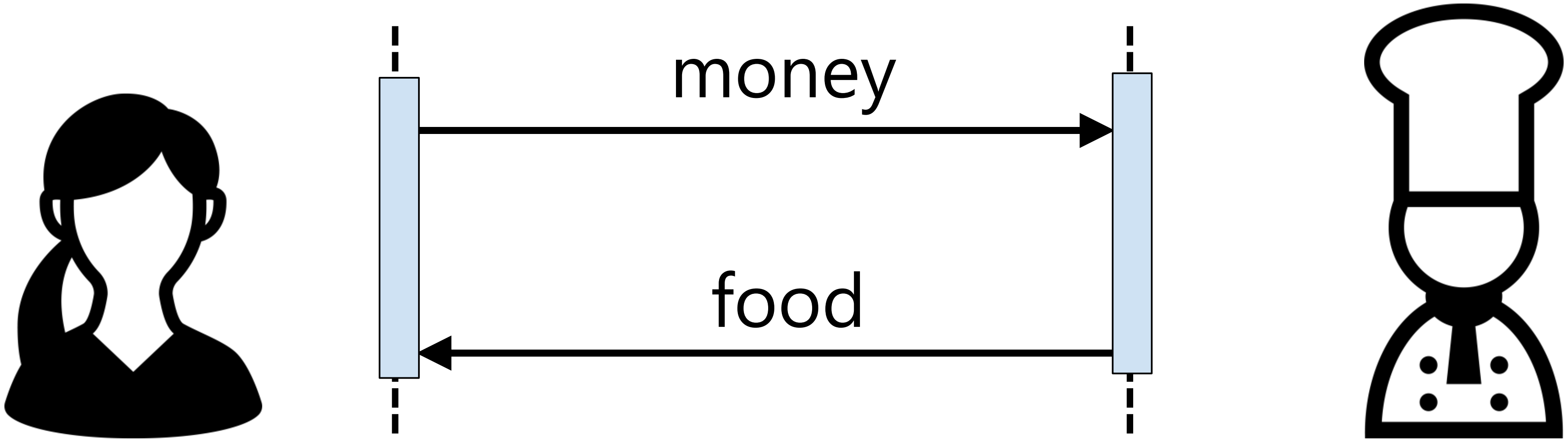
A new method for embedding DSL with a sophisticated type system

- No type-level programming
- Maintaining static guarantees
- Detailed, understandable and customisable error messages

Session Types in 3 minutes

! money; **?**food .... **Dual** .... **?**money; **!** food

(peers doing reciprocal actions)



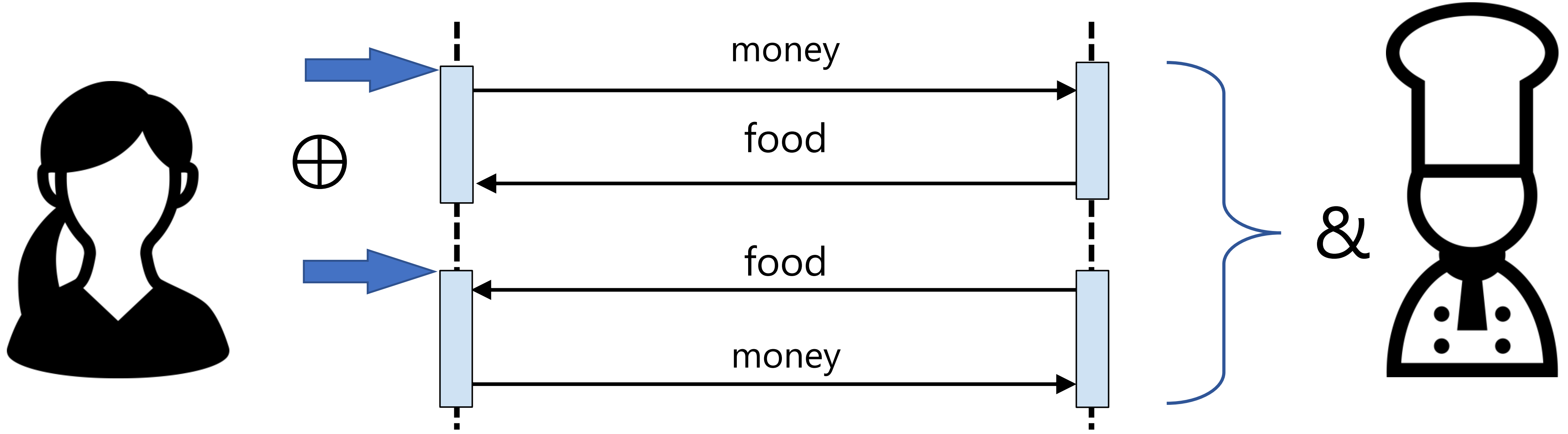
Session Types in 3 minutes

$\oplus$\{take_away: !money; ?food,
eat_in: ?food; !money\}

$\&$\{take_away: ?money; !food,
eat_in: !food; ?money\}

$\oplus$ … internal choice

$\&$ … external choice



(**Proactively** choose branch)

(**Passively** wait for a choice)

Workflow of <sessions>

Communicating Program in **MetaOCaml**

Earlier stage (meta-level)

Session-type Checking & Code generation

Later stage (object-level)

Session-type safe!

Generated Code in **OCaml**

Type-directed programming with <session>!

An integer comparator service of type ?int. ?int. !bool

```
let sh = new_unix_pipe "cmp"
let compare_server =
  accept sh (fun fd ->
    recv fd Int (fun x ->
      recv fd Int (fun y ->
        send fd Bool .< .~x > .~y >.
          finish)))
```

1. Establish a connection
2. Write communication using combinators (**`recv/send`**)

Type-directed programming with <sessions>!

An integer comparator server of type ?int. ?int. !bool

```
let sh = new_unix_pipe "cmp"
let compare_server =
  accept sh @@ fun fd ->
  recv fd Int @@ fun x ->
  recv fd Int @@ fun y ->
  send fd Bool .< .~x > .~y >. @@
  finish
```

1. Establish a connection
2. Write communication using combinators (**recv/send**)

(Session-)Type inference made simple

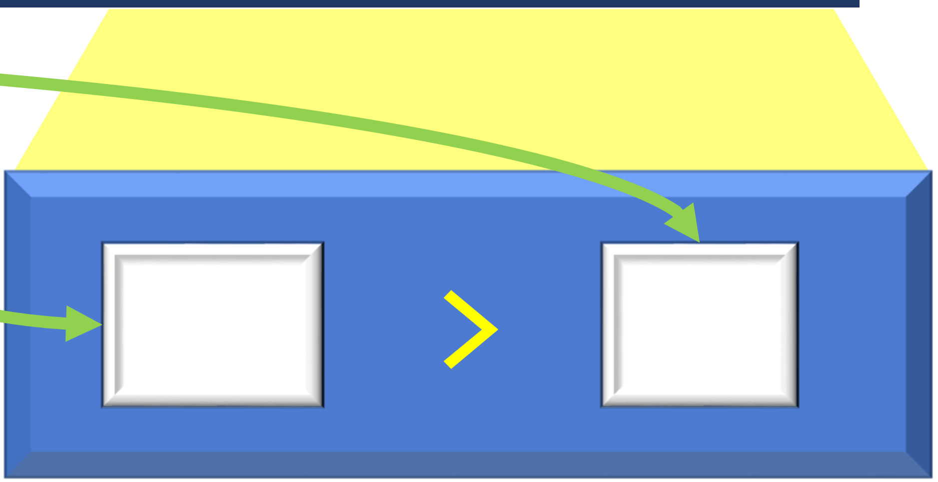
```
metaocaml> infer_thread cmp_server;;
- : string = "[{sh>cmp-3: ?(int).?(int).!(bool).end}][]"
```

- Session types are in term-level, thus just printed as a string
- User-friendly session type syntax

MetaOCaml feature: Use of quotation

```
…
recv fd Int (fun x ->
recv fd Int (fun y ->
…
```

```
send fd Bool .< .~x > .~y >.
```



Earlier stage

Later stage

100 > 200
===> false

Workflow of <sessions> (again)

Communicating Program in **MetaOCaml**

Earlier stage

Session-type Checking & Code generation

Later stage

- No type intricacies
- Better error reporting

Session-type safe!

Generated Code in **OCaml**

Catch Session Type-errors via a Stack Backtrace

...in an 'early' stage!

Comm_basic.th

```
let cmp_client =
  request sh (fun ep ->
    send ep Int .< 100 >. @@
    send ep Int .< 200 >. @@
    send ep Int .< 300 >. @@ (* oops *)
    recv ep Bool @@ fun ans ->
    othr .< Printf.printf "%b" .~ans >. @@
    finish)
```

A session-type error:

Reported in an **earlier** stage

i.e. (sort-of) compile-time,

or "preprocess"-time

```
Fatal error: exception Session Unification Error: unify_gamma: shared cmp-0:
Error: Session types are not unifiable, caused by:

./ex_FLOPS2020.ml:28 (2-13):
        Session: !(bool).end, and

./ex_FLOPS2020.ml:37 (4-25) (dualised):
        Session: !(int).?(bool).end

Called from file "map.ml", line 408, characters 45-57
Called from file "comm_basic.ml", line 157, characters 10-37
Called from file "ex_FLOPS2020.ml", line 42, characters 12-46
```

Trace spots the exact location

Session type errors in [Imai et al., '17]

- Reports two whole interaction-trees between peers…
  - Errors are captured at top-level
  - Type Debugger [Tsushima & Olaf, FLOPS'18] might help

In <session>, term-level debugging is sufficient

- Debugger tools are also useful (e.g. ocamldebug)

```
connect_ xor_ch (fun () ->
  send s (false,true) >>
  recv s >>= fun b ->
  recv s >>= fun b ->   ← actual error is just here (recv duplicated)
  print_bool b;
```

example_journal1.ml 1 of 1 problem

```
This expression has type
  (([ `msg of
        req * (bool * bool) *
        [ `msg of resp * bool * [ `msg of resp * bool * [ `close ] ] ] ],
    req * resp)
   sess * 'a, empty * 'a, unit)
  session
but an expression was expected of type
  (([ `msg of req * (bool * bool) * [ `msg of resp * bool * [ `close ] ] ],
    cli)
   sess * all_empty, all_empty, 'b)
  session
These two variant types have no intersection ocamllsp
```

<sessions>: API type is simple enough

<sessions> [Kiselyov & Imai, 2020]:

```
val send: fd -> 'a code -> th -> th
val recv: fd -> ('a code -> th) -> th
```

<sessions>: API type is simple enough

<sessions> [Kiselyov & Imai, 2020]:

```
val send: fd -> 'a typ -> 'a code -> th -> th
val recv: fd -> 'a typ -> ('a code -> th) -> th
```

Note: 'a typ is for serialisation: not necessary for inter-thread communication

(e.g. for OCaml multicore!)

<sessions>: API type is simple enough

<sessions> [Kiselyov & Imai, 2020]:

```
val send: fd -> 'a typ -> 'a code -> th -> th
val recv: fd -> 'a typ -> ('a code -> th) -> th
```

Note: 'a typ is for serialisation: not necessary for inter-thread communication (e.g. for OCaml multicore!)

Session-OCaml [Imai et al., 2017]:

```
val send : (([`msg of 'r1 * 'v * 'p], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'post) lens -> 'v ->
          ('pre, 'post, unit) monad
```

6 type variables to handle duality & linearity in a static way

Related Work: Convoluted Type Encodings, Toward Static Linearity

Full-sessions (in Haskell) [Imai et al., 2010]:

```
send :: (Pickup ss n (Send v a), Update ss n a ss', IsEnded ss F) => Channel t n -> v -> Session t ss ss' ()
```

6 type variables and 3 type class constraints in context

GVinHs (in Haskell) [Lindley & Morris, 2016]:

```
send :: DualSession s => repr tf i h t -> repr tf h o (st (t <!> s)) -> repr tf i o (st s)
```

8 type variables and a type class, (based on Wadler's GV & Polakow's linearity monad)

Session-OCaml [Imai et al., 2017]:

```
val send : (([`msg of 'r1 * 'v * 'p], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'post) lens -> 'v ->
         ('pre, 'post, unit) monad
val recv : (([`msg of 'r2 * 'v * 'p], 'r1*'r2) sess, ('p, 'r1*'r2) sess, 'pre, 'post) lens ->
         ('pre, 'post, 'v) monad
```

No type classes at all (portable!), but with 6 type variables

Code Generation without hassle, via MetaOCaml

Generated from compare_server

```
val compare_server : th =
{code = .<
  let tmp_1 = {sh_arname = "/tmp/SHsh-0.fifo"; sh_name = "sh-0"} in
  let fd_2 = sock_accept tmp_1 in
  let x_3 = int_of_string (fd_read fd_2) in
  let x_4 = int_of_string (fd_read fd_2) in
  fd_write fd_2 string_of_bool (x_3 > x_4);
  fd_close fd_2;
  ()>. ;
 penv = (<abstr>, <abstr>)}
```

MetaOCaml Code

```
let sh = new_unix_pipe "cmp"
let compare_server =
  accept sh @@ fun fd ->
  recv fd Int @@ fun x ->
  recv fd Int @@ fun y ->
  send fd Bool .< .~x > .~y >. @@
  finish
```

A more elaborated example

[Lindley&Morris,'17]

Branchings and loops, and session-type unification via row types

```
let bakery fd =
  branch fd
  ["take_away", begin
    recv fd Money @@ fun money ->
    send fd Food .< hamberger >. @@
    finish
  end;
  "eat_in", begin
    send fd Food .< hamberger >. @@
    recv fd Money @@ fun money ->
    finish
  end]
```

```
let bakery_customer fd =
  select fd "take_away" @@
  send fd Money .< Yen 100 >. @@
  recv fd Food @@ fun food ->
  finish
```

```
⊕ { take_away: !(money).?(food).end; 'rMeta15 }
```

Row variable

Unifiable via dualisation

```
& { eat_in: !(food).?(money).end;
    take_away: ?(money).!(food).end; <> }
```

Conclusions

A new method for embedding DSL with a sophisticated type system

- No type level programming
- No dependent of fancy types
- Maintaining static guarantees in meta-level, then generating code
- Detailed, understandable and customisable error messages

Future Work

- Multiparty Session Types, OCaml multicore, …

Thank you!