Session Types without Sophistry

System Description

in MetaOCaml

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 assurances: well-session-typed programs "do not go wrong"

• Don't attempt to both read from or both write to a channel
• Obey 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
  o Types are rather convoluted (still fun, but...)
  o Error messages are hard to analyse
  o 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

\[ ! \text{money}; ? \text{food} \quad \text{Dual} \quad ? \text{money}; ! \text{food} \]

(peers doing reciprocal actions)
Session Types in 3 minutes

\(\oplus\{\text{take\_away}: !\text{money}, ?\text{food},\)
\(\text{eat\_in}: ?\text{food}, !\text{money}\}\)

\(\&\{\text{take\_away}: ?\text{money}, !\text{food},\)
\(\text{eat\_in}: !\text{food}, ?\text{money}\}\)

\(\oplus\) ... internal choice

\(\&\) ... external choice

(Proactively choose branch)

(Passively wait for a choice)
Workflow of <sessions>

Communicating Program in MetaOCaml

Session-type safe!

Generated Code in OCaml

Session-type Checking & Code generation

Earlier stage

Later stage
Type-directed programming with <sessions>!

An integer comparator server of type \(\texttt{?int. ?int. !bool}\)

```ocaml
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

```ocaml
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

Earlier stage

Later stage

... recv fd Int (fun x -> ... recv fd Int (fun y -> ... send fd Bool < (~x) > (~y) >.

100 > 200 ====> false
Workflow of `<sessions>` (again)

- Communicating Program in **MetaOCaml**
- **Session-type Checking & Code generation**
  - Session-type safe!
  - Generated Code in **OCaml**
  - Earlier stage
  - Later stage
  - No type intricacies
  - Better error reporting
Catch Session Type-errors via a Stack Backtrace

...in an 'early' stage!

A session-type error:

Reported in an earlier stage

i.e. (sort-of) compile-time, or "preprocess"-time

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 <sessions>, term-level debugging is sufficient

- Debugger tools are also useful (e.g. ocamldebug)
<sessions>: API type is simple enough

<sessions> [Kiselyov & Imai, 2020]:

```ocaml
val send: fd -> 'a code -> th -> th
val recv: fd -> ('a code -> th) -> th
```
<sessions>: API type is simple enough

<sessions> [Kiselyov & Imai, 2020]:

```ocaml
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]:

```ocaml
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]:

```ocaml
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]:

```haskell
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]:

```haskell
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]:

```ocaml
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>)}
A more elaborated example

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 .< humberger >. @@
    finish
  end;

"eat_in", begin
  send fd Food .< humberger >. @@
  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 }
& { eat_in: !(food).?(money).end; take-away: ?(money).!(food).end; <> }
```

Row variable

Unifiable via dualisation
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

Thank you!