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 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
  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

⊕\{\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}\}

⊕ ... internal choice

& ... external choice

\text{(Proactively choose branch)}

\text{(Passively wait for a choice)}
Workflow of <sessions>

Communicating Program in MetaOCaml

Session-type Checking & Code generation

Session-type safe!

Generated Code in OCaml

Earlier stage (meta-level)

Later stage (object-level)
Type-directed programming with <session>!

An integer comparator service of type \( ?\text{int.} \ ?\text{int.} \ !\text{bool} \)

```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`

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

Later stage

Session-type Checking & Code generation

• 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!

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 <session>, term-level debugging is sufficient

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

<sessions> [Kiselyov & Imai, 2020]:

```plaintext
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
define type { 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
define (val send : fd -> 'a typ -> 'a code -> th -> th) and (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
define (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 :: DualSession s => repr tf i h t -> repr tf h o (st (t <!> s)) -> repr tf i o (st s)
```

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 = "\(\text{/tmp/SHsh-0.fifo}\); sh_name = "\(\text{sh-0}\)"} in
  let fd_2 = \text{sock_accept} \text{tmp}_1 \text{in}
  let x_3 = \text{int_of_string} (\text{fd_read} \text{fd}_2) \text{in}
  let x_4 = \text{int_of_string} (\text{fd_read} \text{fd}_2) \text{in}
  \text{fd_write} \text{fd}_2 \text{string_of_bool} \text{(x}_3 > \text{x}_4);
  \text{fd_close} \text{fd}_2;
  ()>. ;
  penv = (\text{<abstr>}, \text{<abstr>})}

MetaOCaml Code

let sh = \text{new_unix_pipe} \text{"cmp"}
let compare_server =
  \text{accept} sh \text{@@ fun} \text{fd} ->
  \text{recv} \text{fd} \text{Int} \text{@@ fun} x ->
  \text{recv} \text{fd} \text{Int} \text{@@ fun} y ->
  \text{send} \text{fd} \text{Bool} .< .\text{x} > .\text{y} >. @@
  \text{finish}
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 .< 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!