Polymorphic Multiparty Session Handlers in OCaml

Keigo Imai
Gifu University, Japan

1 Summary

Concurrent programming is widespread today; however, typing disciplines for concurrency are hardly accessible for programmers, showing a stark contrast to their sequential counterparts, such as typing for objects or functions. *Session types* [3] have been repeatedly integrated into various languages in the last decade. However, lack of *linearity* in the host type system usually compromises their safety, hampering possible adoption in real-world use. For instance, see the following common (mis)use of session types written in OCaml [11] on the left, which tries to output an infinite stream of `'*'` on channel `s`:

```
let rec bad () = send "*" s; bad ()
```

```
let rec loop s = loop (send "*" s)
```

Unfortunately, a *linearity violation error* is raised in the second iteration of `bad()`, prohibiting the unrestricted use of `s`. Instead, to track sessions via the host type system, channels must be used *linearly*, i.e., used *exactly once*, as in `(loop s)` using `loop` in the right, where the *fresh* channel returned by `send` is then consumed in the next iteration. OCaml typechecker does not ensure such property, as in other languages.

Event handlers and callbacks. Event handlers have been utilised for concurrent (e.g., GUI) and distributed (e.g., TCP/IP) programming. Along with the known benefit of event handlers for runtime performance (see, e.g., [10, § 2.1]), it features *callbacks* as the notable programming style, enabling the programmer to focus on *reactions* of the system without explicitly handling resources in concern.

Polymorphic event handlers for sessions. Our proposal, *polymorphic session handlers*, alleviates the need for linear types in session programming by exploiting the event handler style, thus achieving *fully static* typechecking of session types in the vanilla OCaml. The benefit is to express sessions as an event handler using *structural polymorphism* prevalent in OCaml, offering (1) concise, (2) easy-to-reuse, and (3) safe-by-construction concurrent programming environment. The following example of an arithmetic server on the left shows all the essential ingredients of our proposal:

```
let rec server acc = object
  method add x =
    `res(x+acc, server (x+acc))
  method div x =
    if x = 0 then
      `err("div by zero", server acc)
    else
      `res(acc/x, server (acc/x))
  method stop = ()
end
```

```
let rec client xs n = match xs with
| x :: xs ->
  `add(x, object method res _ = client xs n end)
| [] ->
  `div(n, object
    method res ans = printf "result:%d\n" ans;
                     `stop ()
    method err msg = printf "error:%s" msg;
                     `stop ()
  end)
```

The first line defines a function `server` having accumulator parameter `acc`, returning an *object* which offers services `add` and `div` via its *methods* having a parameter `x`. Each service returns *tagged (polymorphic variant)* values `` `res(..) `` or `` `err(..) `` containing the pair of the message value (e.g., `x+acc`) and *continuation* where the recursive calls to `server` take place. For example, given the initial value of `acc` is `9`, the request of `add` with value `33` to the server causes the response of `res` with `42`. The server terminates by a `stop` request. The `client` computes the average of the given list: for example, `(client [10;20;30] 3)` will print `"result: 20"`. Note that the program comprises *pure* OCaml, without any syntactic or type

Submitted to:
PLACES 2022

extensions, achieving full reusability of the handler.

Running a session with the duality witness. The correctness of communication is established via the *protocol combinator* DSL from [6]. The code below constructs the *protocol* `arith_protocol`, which witnesses the reciprocal, safe communication between two handlers, called *duality* in literature [3]. Finally, the *runners* `{cli,srv}_runner` extracted from the protocol via `create_runners` start the handlers:

```
let rec arith_protocol = fix (fun t ->
  choice_cli_3 add_or_div_or_stop
    ( (cli --> srv) add @@ (srv --> cli) res t
    , (cli --> srv) div @@ choice_srv_2 res_or_err ( (srv --> cli) res t
                                                   , (srv --> cli) err t )
    , (cli --> srv) stop finish ))
let (srv_runner, cli_runner) = create_runners arith_protocol  in
Thread.create srv_runner (server 0); Thread.create cli_runner (client [10;20;30] 3)
```

Each thread started via `Thread.create` safely communicate with each other without getting stuck.

Subtyping polymorphism. OCaml typechecker infer the following type (left) for expression `(server 0)`:

```
< add : int -> [> `res of int * 'a ];          < add : int -> [ `res of int * 'b ];
  div : int -> [> `err of string * 'a            div : int -> [ `err of string * 'b
                | `res of int * 'a ];                         | `res of int * 'b ];
  stop : unit > as 'a                            stop : unit; .. > as 'b -> unit
```

while `srv_runner` has the type on the right, exhibiting compatibility between the declared protocol and actual session behaviour. Types `<···>` and `[···]` are the object and polymorphic variant types of OCaml respectively, which can be transliterated to the standard *external* and *internal* choice of the binary session types of [3], respectively. The ascription `as` in the end binds the type variable `'a` (`'b` resp.) in the rest of the type, representing a loop in the session. Symbols `>` in variant types and `..` in object types denotes that *more* tags are allowed there, simulating *subtyping* for session types [2].

Multiparty sessions. The *multiparty* session types [4] drastically expands the expressiveness of protocols. Polymorphic session handlers can be extended to multiparty communication as well, simply by adding *participants'* names to the handlers. The following circulates messages `ping`, `pong` and `pang` from `a` to `b` then `c`, by merely prefixing the destination as a variant tag (`` `a ``, `` `b `` and `` `c ``).

```
let rec fa x  = printf "%d\n" x; `b(`ping(x, `c(object method pang x = fa (x+1))))
let rec fb () = `a(object method ping x = `c(`pong(x+1, fb ())))
let rec fc () = `b(object method pong x = `a(`pang(x+1, fc ())))
```

Multiparty compatibility, top-down vs bottom-up. The *multiparty compatibility* [1], ensuring safe communications among multiple participants analogous to the duality, can be checked by writing protocol combinators [6] (`ring_prot` below left). Recently, `kmclib` [5], proposed by Imai et al., combines OCaml type inference and *bounded model checking* [9] to verify session types of an OCaml program *without* writing protocol combinators (`kmc.runner` below, analogous to `kmc.gen` in [5]). Both *top-down* and *bottom-up* approaches work well with polymorphic session handlers:

```
let ring_prot = fix (fun t -> (* Writing the witness *)    (* Inferring the witness *)
  (a-->b) ping @@ (b-->c) pong @@ (c-->a) pang t)         let (run_a, run_b, run_c) =
let [run_a; run_b; run_c] = create_runners ring_prot;;        [%kmc.runner (a,b,c)];;

Thread.create run_a (fa 42); Thread.create run_b (fb ()); Thread.create run_c (fc ());;
```

Conclusion and related work. We have shown an approach for writing session-typed programs in a callback style, aiming for a safer and concise framework for concurrency, and for easier use for programmers without compromising type safety. Zhou et al. [12] use callback style to implement their refined multiparty session types. The idea of sessions-as-handlers can be seen in [8], written in Haskell. The author also presented an earlier version of this work [7].

References

[1] Pierre-Malo Deniélou & Nobuko Yoshida (2013): *Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types*. In: *ICALP 2013*, pp. 174–186, doi:10.1007/978-3-642-39212-2_18. Available at `http://dx.doi.org/10.1007/978-3-642-39212-2_18`.

[2] Simon Gay & Malcolm Hole (2005): *Subtyping for Session Types in the Pi Calculus*. *Acta Informatica* 42(2), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[3] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–138, doi:10.1007/BFb0053567.

[4] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 273–284, doi:10.1145/1328438.1328472. Available at `https://doi.org/10.1145/1328438.1328472`.

[5] Keigo Imai, Julien Lange & Rumyana Neykova (2022): *Kmclib: Automated Inference and Verification of Session Types*. In: *TACAS 2022*, Lecture Notes in Computer Science. To appear. Extended version available at `https://arxiv.org/abs/2111.12147`.

[6] Keigo Imai, Rumyana Neykova, Nobuko Yoshida & Shoji Yuen (2020): *Multiparty Session Programming With Global Protocol Combinators*. In Robert Hirschfeld & Tobias Pape, editors: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, *LIPIcs* 166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:30, doi:10.4230/LIPIcs.ECOOP.2020.9.

[7] Keigo Imai, Shoji Yuen & Kiyoshi Agusa (2007): *Highly Reliable Network Programming with Session Types in Typed Functional Programming Languages*. Available at `https://researchmap.jp/keigoi1982/presentations/7163551?lang=en`. In Japanese.

[8] Oleg Kiselyov, Simon Peyton Jones & Chung-chieh Shan (2010): *Fun with Type Functions*. In A. W. Roscoe, Clifford B. Jones & Kenneth R. Wood, editors: *Reflections on the Work of C. A. R. Hoare*, Springer, pp. 301–331, doi:10.1007/978-1-84882-912-1_14. Available at `https://doi.org/10.1007/978-1-84882-912-1_14`.

[9] Julien Lange & Nobuko Yoshida (2019): *Verifying Asynchronous Interactions via Communicating Session Automata*. In Isil Dillig & Serdar Tasiran, editors: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, *Lecture Notes in Computer Science* 11561, Springer, pp. 97–117, doi:10.1007/978-3-030-25540-4_6.

[10] Peng Li & Steve Zdancewic (2007): *Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives*. In Jeanne Ferrante & Kathryn S. McKinley, editors: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, ACM, pp. 189–199, doi:10.1145/1250734.1250756. Available at `https://doi.org/10.1145/1250734.1250756`.

[11] Luca Padovani (2019): *Context-Free Session Type Inference*. *ACM Trans. Program. Lang. Syst.* 41(2), pp. 9:1–9:37, doi:10.1145/3229062.

[12] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova & Nobuko Yoshida (2020): *Statically verified refinements for multiparty protocols*. *Proc. ACM Program. Lang.* 4(OOPSLA), pp. 148:1–148:30, doi:10.1145/3428216. Available at `https://doi.org/10.1145/3428216`.