arXiv:2111.12147v2 [cs.PL] 26 Nov 2021

Kmclib: Automated Inference and Verification of Session Types from OCaml Programs

Keigo Imai[1], Julien Lange[2], and Rumyana Neykova[3]

[1] Gifu University, Japan
[2] Royal Holloway, University of London, UK
[3] Brunel University London, UK

Abstract. Theories and tools based on multiparty session types offer correctness guarantees for concurrent programs that communicate using message-passing. These guarantees usually come at the cost of an intrinsically top-down approach, which requires the communication behaviour of the entire program to be specified as a global type.
This paper introduces `kmclib`: an OCaml library that supports the development of *correct* message-passing programs without having to write any types. The library utilises the meta-programming facilities of OCaml to automatically infer the session types of concurrent programs and verify their compatibility (k-MC [13]). Well-typed programs, written with `kmclib`, do not lead to communication errors and cannot get stuck.

Keywords: Multiparty Session Types · Concurrent Programming · OCaml

1 Introduction

Multiparty session types (MPST) [6] are a popular type-driven technique to ensure the correctness of concurrent programs that communicate using message-passing. The key benefit of MPST is to guarantee statically that the components of a program have compatible behaviours, and thus no components can get permanently stuck. Many implementations of MPST in different programming languages have been proposed in the last decade [16,7,10,19,15,2,8,14,5,22], however, all suffer from a notable shortcoming: they require programmers to adopt a top-down approach that does not fit well in modern development practices. When changes are frequent and continual (e.g., continuous delivery), re-designing the program and its specification at every change is not feasible.

Most MPST theories and tools advocate an intrinsically top-down approach. They require programmers to specify the communication (often in the form of a global type) of their programs before they can be type-checked. In practice, type-checking programs against session types is very difficult. To circumvent the problem, most implementations of MPST rely on *external* toolings that generate code from a global type, see e.g., all works based on the Scribble toolchain [21].

In this paper, we present an OCaml library, called `kmclib`, that supports the development of programs which enjoy all the benefits of MPST while avoiding their main drawbacks. The `kmclib` library guarantees that threads in well-typed

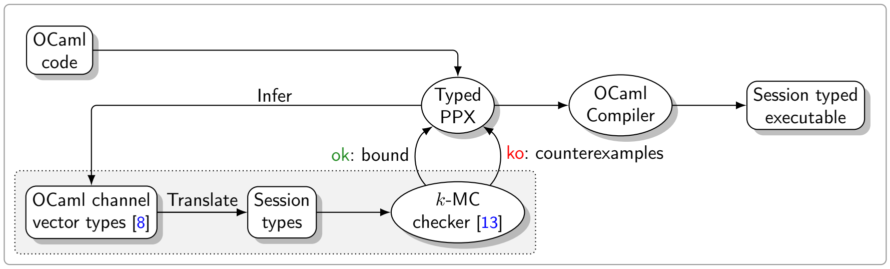


Fig. 1: Workflow of the `kmclib` library.

programs will not get stuck. The library also enables *bottom-up development*: programmers write message-passing programs in a natural way, without having to write session types. Our library is built on top of *Multicore OCaml* [20] that offers highly scalable and efficient concurrent programming, but does not provide any static guarantees wrt. concurrency.

Figure 1 gives an overview of `kmclib`. Its implementation combines the power of the *type-aware* macro system of OCaml (Typed PPX) with two recent advances in the session types area: an encoding of MPST in OCaml (channel vector types [8]) and a session type compatibility checker (k-MC checker [13]). To our knowledge, this is the first implementation of type inference for MPST and the first integration of compatibility checking in a programming language.

The `kmclib` library offers several advantages compared to earlier MPST implementations. **(1)** It is *flexible*: programmers can implement communication patterns (e.g., fire-and-forget patterns [13]) that are not expressible in the synchrony-oriented syntax of global types. **(2)** It is *lightweight* as it piggybacks on OCaml's type system to check and infer session types, hence lifting the burden of writing session types off the programmers. **(3)** It is *user-friendly* thanks to its integration in Visual Studio Code, e.g., compatibility violations are mapped to precise locations in the code. **(4)** It is *well-integrated* into the natural edit-compile-run cycle. Although compatibility is checked by an external tool, this step is embedded as a compilation step and thus hidden from the user.

2 Safe Concurrent Programming in Multicore OCaml

We give an overview of the features and usage of `kmclib` using the program in Figure 2 (top) which calculates Fibonacci numbers. The program consists of three concurrent *threads* (`user`, `master`, and `worker`) that interact using point-to-point message-passing. Initially, the `user` thread sends a request to the `master` to start the calculation, then waits for the `master` to return a work-in-progress message, or the final result. After receiving the result, the `user` sends back a stop message. Upon receiving a new request, the `master` splits the initial computation in two and sends two tasks to a `worker`. For each task that the `worker` receives, it replies with a result. The `master` and `worker` threads are recursive and terminate only upon receiving a stop message.

```
let KMC (uch,mch,wch) = [%kmc.gen (u,m,w)]

let user () =
  let uch = send uch#m#compute 42 in
  let rec loop uch : unit =
    match receive uch#m with
    | `wip(res, uch) ->
      printf "in progress: %d\n" res;
      loop uch
    | `result(res, uch) ->
      printf "result: %d\n" res;
      send uch#m#stop ()
  in loop uch

let worker () =
  let rec loop wch : unit =
    match receive wch#m with
    | `task(num, wch) ->
      loop (send wch#m#result (fib num))
    | `stop((), wch) -> wch
  in loop wch
```

```
let master () =
  let rec loop (mch : [%kmc.check u]) : unit =
    match receive mch#u with
    | `compute(x, mch) ->
      let mch = send mch#w#task (x - 2) in
      let mch = send mch#w#task (x - 1) in
      let `result(r1, mch) = receive mch#w in
      let mch = send mch#u#wip r1 in
      let `result(r2, mch) = receive mch#w in
      loop (send mch#u#result (r1 + r2))
    | `stop((), mch) ->
      send mch#w#stop ()
  in loop mch

let () =
  let ut = Thread.create user () in
  let mt = Thread.create master () in
  let wt = Thread.create worker () in
  List.iter Thread.join [ut;mt;wt]
```

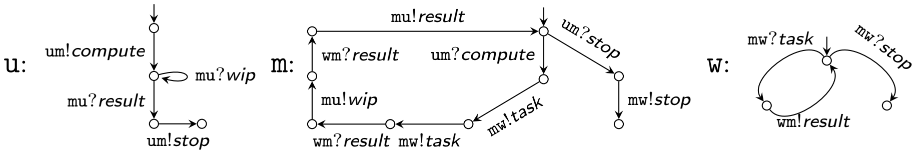
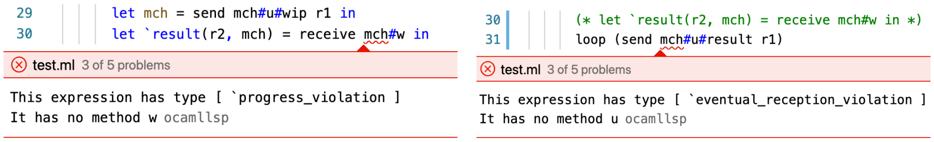


Fig. 2: Example of `kmclib` program (top) and *inferred* session types (bottom).

Figure 2 (bottom) gives a session type for each thread, i.e., the behaviour of each thread wrt. communication. For clarity we represent session types as a communicating finite state machine (CFSM [1]), where ! (resp. ?) denotes sending (resp. receiving). For example, um!*compute* means that the user is sending to the master a message *compute*, while um?*compute* says that the master receives *compute* from the user. Our library infers these CFSM representations from the OCaml code, in Figure 2 (top), and verifies *statically* that the three threads are *compatible*, hence no thread can get stuck due to communication errors. If compatibility cannot be guaranteed, the compiler reports the kind of violations (i.e., *progress* or *eventual reception* error) and their locations in the code. Figure 3 shows how such semantic errors are reported visually in Visual Studio Code.

Albeit simple, the common communication pattern used in Figure 2 cannot be expressed as a global type, and thus cannot be implemented in previous MPST implementations. Concretely, global types cannot express the intrinsic asynchronous interactions between the master and worker threads (i.e., the master may send a second task message, while the worker sends a result).

Programming with `kmclib`. To enable safe message-passing programs, `kmclib` provides two communication primitives, `send` and `receive`, and two primitives for channel creation (`KMC` and `%kmc.gen`). We only give a user-oriented description of these primitives here (see Appendix A an overview of their implementations).



```
let mch = send mch#u#wip r1 in
let `result(r2, mch) = receive mch#w in
```

test.ml 3 of 5 problems

```
This expression has type [ `progress_violation ]
It has no method w ocamllsp
```

```
(* let `result(r2, mch) = receive mch#w in *)
loop (send mch#u#result r1)
```

test.ml 3 of 5 problems

```
This expression has type [ `eventual_reception_violation ]
It has no method u ocamllsp
```

Fig. 3: Examples of type errors.

The crux of `kmclib` is the **session channel creation**: `[%kmc.gen (u,m,w)]` at Line 1. This primitive takes a tuple of *role names* as argument (i.e., `(u,m,w)`) and returns a tuple of communication channels, which are bound to `(uch,mch,wch)`. These channels will be used by the threads implementing roles `user` (Lines 3-13), `worker` (Lines 15-21), and `master` (Lines 22-34). By default, channels are implemented using concurrent queues from Multicore OCaml (`Domainslib.Chan.t`) but other underlying transports can easily be provided.

Threads send and receive messages over these channels using the communication primitives provided by `kmclib`. The send primitive requires three *arguments*: a channel, a destination role, and a message. For instance, the `user` sends a request to the `master` with `send uch#m#compute 20` where `uch` is the user's communication channel, `m` indicates the destination, and `compute 20` is the message (consisting of a label and a payload). Observe that a sending operation returns a new channel which is to be used in the continuation of the interactions, e.g., `uch` bound at Line 4. Receiving messages work in a similar way to sending messages, e.g., see Line 6 where the `user` waits for a message from the `master` with `receive uch#m`. We use OCaml's pattern matching to match messages against their labels and bind the payload and continuation channel. See, e.g., Lines 7-10 where the `user` expects either `wip` or `result` message. The receive primitive returns the payload `res` and a new communication channel `uch`.

New thread instances are spawned in the usual way; see Lines 36-39. The code at Line 40 waits for them to terminate.

Compatibility and error reporting. While the code in Figure 2 may appear unremarkable, it hides a substantial machinery that guarantees that, if a program type-checks, then its constituent threads are safe, i.e., no thread gets permanently stuck and all messages that are sent are eventually received. This property is ensured by `kmclib` using OCaml's type inference and PPX plugins to infer a session type from each thread then check whether these session types are *k-multiparty compatible* (k-MC) [13].

If a system of session types is k-MC, then it is safe [13, Theorem 1], i.e., it has the *progress* property (no role gets permanently stuck in a receiving state) and the *eventual reception* property (all sent messages are eventually received). Checking k-MC notably involves checking that all their executions (where each channel contains at most k messages) satisfy progress and eventual reception.

The k-MC-checker [13] performs a bounded verification to discover the *least* k for which a system is k-MC, up-to a specified upper bound N. In the `kmclib`

API, this bound can be optionally specified with `[%kmclib.gen` *roles* `~bound:`N`]`. The k-MC-checker emits an error if the bound is insufficient to guarantee safety.

The `[%kmc.gen (u,m,w)]` primitive also feeds the results of k-MC checking back to the code. If the inferred session types are k-MC, then channels for roles `u`, `m` and `w` can be generated. If k-MC cannot be guaranteed, then this results in a type error. We have modified the k-MC-checker to return counterexample traces when the verification fails. This helps give actionable feedback to the programmer, as counterexample traces are translated to OCaml types and inserted at the hole corresponding to `[%kmc.gen]`. This has the effect of reporting the precise location of the errors.

To report errors in a function parameter, we provide an *optional* macro for types: `[%kmc.check rolename]` (see faded code in Line 23). Figure 3 shows examples of such error reports. The left-hand-side shows the reported error when Line 26 is commented out, i.e., the master sends one task, but expects two result messages; hence progress is violated since the master gets stuck at Line 30. The right-hand-side shows the reported error when Line 30 is commented out. In this case, variable `mch` in Line 31 (`master`) is highlighted because the `master` fails to consume a message from channel `mch`.

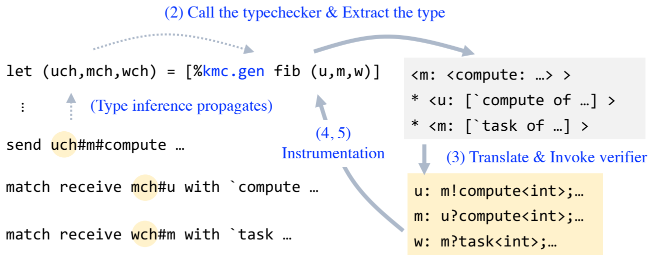
3 Inference of Session Types in `kmclib`

The `kmclib` API. The `kmclib` primitives allow the vanilla OCaml typechecker to infer the session structure of a program, while simultaneously providing a user-friendly communication API for the programmer. To enable inference of session types from concurrent programs, we leverage OCaml's structural typing and row polymorphism. In particular, we reuse the encoding from [8] where input and output session types are encoded as polymorphic variants and objects in OCaml. In contrast to [8] which relies on programmers writing global types prior to type-checking, `kmclib` infers and verifies local session types automatically, without requiring any additional type or annotation.

Typed PPX Rewriter. To extract and verify session types from a piece of OCaml code, the `kmclib` library makes use of OCaml PreProcessor eXtensions (PPX) plugins which provide a powerful meta-programming facility. PPX plugins are invoked during the compilation process to manipulate or translate the abstract syntax tree (AST) of the program. This is often used to insert additional definitions, e.g., pretty-printers, at compile-time.

A key novelty of `kmclib` is the combination of PPX with a form of *type-aware translation*, whereas most PPX plugins typically perform purely syntactic (type-unaware) translations. Figure 4 shows the workflow of the PPX rewriter, overlayed on code snippets from Figure 2. The inference works as follows.

1. The plugin reads the AST of the program code to replace the `[%kmc.gen]` primitive with a *hole*, which can have *any* type.
2. The plugin invokes the *typechecker* to get the *typed* AST of the program. In this way, the type of the hole is *inferred* to be a tuple of *channel object types* whose structure is derived from their usages (i.e., `mch#u#compute`).


(2) Call the typechecker & Extract the type
(Type inference propagates)
(4, 5) Instrumentation
(3) Translate & Invoke verifier

```
let (uch,mch,wch) = [%kmc.gen fib (u,m,w)]
  ⋮
send uch#m#compute …
match receive mch#u with `compute …
match receive wch#m with `task …
```

```
<m: <compute: …> >
* <u: [`compute of …] >
* <m: [`task of …] >
```

```
u: m!compute<int>;…
m: u?compute<int>;…
w: m?task<int>;…
```

Fig. 4: Inferring session types from OCaml code.

To enable this propagation, we introduce the idiom "`let (KMC ...) = ...`" which enforces the type of the hole to be *monomorphic*. Otherwise, the type would be too general and this would spoil the type propagation (See § B).

3. The inferred type is translated to a system of (local) *session types*, which are passed to the k-MC-checker.
4. If the system is k-MC, then it is safe and the plugin *instruments* the code to allocate a fresh channel tuple (i.e., concurrent queues) at the hole.
5. If the system is unsafe, the k-MC-checker returns a *violation trace* which is translated back to an OCaml type and inserted at the hole, to report a more precise error location.

The translation is limited inside the `[%kmc.gen]` expression, retaining a clear correspondence between the original and translated code. It can be understood as a form of *ad hoc polymorphism* reminiscent of type classes in Haskell. Like the Haskell typechecker verifies whether a type belongs to a class or not, the `kmclib` verifies whether the set of session types belongs to the class of k-MC systems.

4 Conclusion

We have developed a practical library for safe message-passing programming. The library enables developers to program and verify arbitrary communication patterns without the need for type annotations or user-operated external tools. Our *automated verification* approach can be applied to other general-purpose programming languages. Indeed it mainly relies on two ingredients: structural typing and metaprogramming facilities. Both are available, with a varying degree of support, in, e.g., Scala, Haskell, TypeScript, and F#.

Our work is reminiscent of automated software model checking which has a long history (see [9] for a survey). There are few works on inference and verification of behavioural types, i.e., [18,11,12,3]. However, Perera et al. [18] only present a prototype research language, while Lange et al. [11,12,3] propose verification procedures for Go programs that rely on external tools which are not integrated with the language nor its type system. To our knowledge, ours is the first implementation of type inference for MPST and the first integration of session types compatibility checking within a programming language.

References

1. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
2. D. Castro, R. Hu, S. Jongmans, N. Ng, and N. Yoshida. Distributed programming using role-parametric session types in Go: statically-typed endpoint apis for dynamically-instantiated communication structures. *PACMPL*, 3(POPL):29:1–29:30, 2019.
3. N. Dilley and Julien Lange. Automated verification of Go programs via bounded model checking. In *International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, July 2021. To appear.
4. J. Garrigue. Relaxing the value restriction. In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2004.
5. P. Harvey, S. Fowler, O. Dardha, and S. J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In A. Møller and M. Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
6. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL 2008*, pages 273–284, 2008.
7. R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE 2016*, pages 401–418, 2016.
8. K. Imai, R. Neykova, N. Yoshida, and S. Yuen. Multiparty session programming with global protocol combinators. In *ECOOP*, volume 166 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
9. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), Oct. 2009.
10. D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP 2016*, pages 146–159, 2016.
11. J. Lange, N. Ng, B. Toninho, and N. Yoshida. Fencing off Go: liveness and safety for channel-based programming. In *POPL 2017*, pages 748–761, 2017.
12. J. Lange, N. Ng, B. Toninho, and N. Yoshida. A static verification framework for message passing in Go using behavioural types. In *ICSE 2018*. ACM, 2018.
13. J. Lange and N. Yoshida. Verifying asynchronous interactions via communicating session automata. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019.
14. A. Miu, F. Ferreira, N. Yoshida, and F. Zhou. Generating Interactive WebSocket Applications in TypeScript. *Electronic Proceedings in Theoretical Computer Science*, 314:12–22, Apr. 2020.
15. R. Neykova, R. Hu, N. Yoshida, and F. Abdeljallal. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F♯. In *CC 2018*. ACM, 2018.
16. N. Ng, J. G. de Figueiredo Coutinho, and N. Yoshida. Protocols by default - safe MPI code generation based on session types. In *CC 2015*, pages 212–232, 2015.
17. L. Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.
18. R. Perera, J. Lange, and S. J. Gay. Multiparty compatibility for concurrent objects. In *PLACES 2016*, pages 73–82, 2016.

19. A. Scalas, O. Dardha, R. Hu, and N. Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP 2017*, pages 24:1–24:31, 2017.
20. K. C. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy. Retrofitting parallelism onto ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):113:1–113:30, 2020.
21. N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC 2013*, volume 8358, pages 22–41. Springer, 2013.
22. F. Zhou, F. Ferreira, R. Hu, R. Neykova, and N. Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020.

A Technical Details on the kmclib API

We explain the main communication primitives of kmclib and their translation to session types. In particular, we reuse the encoding [8] where input and output session types are encoded as polymorphic variants and objects, while loops are naturally handled using *equi-recursive types* in OCaml.

Objects and variants in OCaml are structurally typed, which enables the creation of ad-hoc types. This allows the channel structure to be derived from the usage of channels in `send` and `receive` primitives.

Output types. Sending a message, e.g., in Line 4 of Figure 2, is parsed as `send (uch#m#compute) 42` where the two chained *method calls* yields a port for sending `compute` label to role `m`, which in turn is passed to `send` (together with a payload). This corresponds to an *internal choice* where the `user` specifies a destination for its message and chooses a label from those offered by the receiver.

The inferred type of channel object `uch` is a nested object type of the form `<m: <compute: (int, ch)> out>` where `m` is a method that returns an object that itself provides method `compute` (which returns a port for sending an `int` payload and yielding a continuation channel `ch`). Note that the implementation of these methods is not provided explicitly by the API nor the programmer, instead they are constructed on-demand when invoking `uch#m#compute`; i.e., objects are generated automatically according to the method types that is invoked on them (`#` denotes method invocation). Such object types correspond to session types of the form `m!compute<int>;ch` (the translation is trivial).

Input types. To receive messages, as in Lines 6-12 of Figure 2, we use `uch#m` to return a channel object which effectively corresponds to a port originating from role `m`. This channel is then passed to the `receive` primitive, which returns a *polymorphic variant* on which one needs to pattern match for expected messages.

The inferred type of `wch`, which specifies the expected messages and their respective continuation, is

```
<u: [`compute of int * ch′ | `stop of unit * ch″] inp>
```

This type corresponds to an external choice, i.e., session type of the form

```
{u?compute<int>;ch′} or {u?stop<unit>;ch″}.
```

Linearity. MPST require channels to be used linearly, i.e., each channel must be used exactly once. If a channel is not used, this leads to a multiparty compatibility issue (a message will not be sent/received), and hence our implementation detects such issues statically via k-MC.

The idiomatic *shadowing* with the same variable names (e.g., re-binding of `uch` in Line 4) in OCaml mitigates the risk of using a channel more than once. If the program deviates from this best practice and a channel is used non-linearly, an exception is raised at runtime.

Alternatively, kmclib provides an event-based alternative API (similar to that of [22]), which eliminates the explicit need for linear channels, at the cost of

```
1 let KMC (uch,mch,wch) =
2   let um, mu, mw, wm = Chan.create_unbounded (), Chan.create_unbounded (), ... in
3   let uch = <m = <compute = Internal.make_out um (fun v -> `compute v) ... > > in
4   let mch = <u = Internal.make_inp mu ... (fun v -> `compute v) > in
5   let wch = <m = Internal.make_inp mw ... (fun v -> `stop v) > in
6   make_tuple (uch, mch, wch)
```

Fig. 5: Code from Figure 2, instrumented at [kmc.gen]

losing a direct-style API.[4] We remark that there are other known ways to check linearity statically [8], which can easily be adapted to our library.

B Instrumented Code for Figure 2

Figure 5 shows the instrumented code for Figure 2. Line 2 allocates *raw* channels using Chan.create_unbounded from Multicore OCaml, and Lines 3-5 create objects inhabiting the inferred type. We use shorthand <...> for *in-place* objects object...end and abbreviate the continuations with an ellipsis.

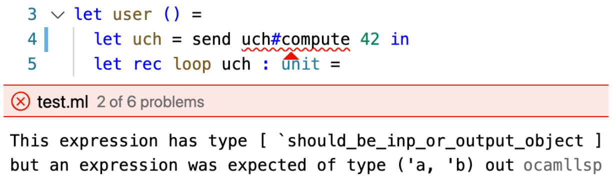
The Internal functions make a channel from raw channels and a continuation. In particular, make_out takes an extra function (fun v -> `label v), allocating a variant tag representing the message label. Also, it uses type casts from Obj module in OCaml, which is a common technique to implement session types in OCaml (cf. [17]).

Line 6 (make_tuple) wraps the resulting tuple with the KMC constructor. As mentioned in § 3, this makes the inferred hole type *monomorphic*. Normally, for the top-level declarations, OCaml generalises the type to be *polymorphic* and the hole type is inferred as $\forall\alpha.\alpha$ (can be instantiated with any type at *any site*) if its occurrence is at the *covariant* position, spoiling the propagation (cf. relaxed value restriction [4]). We avoid this by wrapping the pattern with a type KMC : 'a -> 'a tuple declared explicitly as non-covariant.

C Error Reporting with Type Ascription

It is vital to show the *location* of the error to the programmer when an error is found. To achieve this, the PPX plugin of kmclib instruments an extra *ascription* of an *incompatible type* at the erroneous usage of a channel. For example, see the error at Line 30 in the left of Figure 3, where the PPX plugin assigns the variable mch a type [`progress_violation] (a single variant constructor type whose name is progress_violation), as the $k$-MC-checker detects the input blocking forever at that point. Since it is used as <u: [`result of int * ···] inp > denoting an input of result with an int from the user, the OCaml typechecker reports a type error.

[4] See https://github.com/keigoi/kmclib/blob/tooldemo/test/paper/test_handler.ml for an example.



```
let user () =
  let uch = send uch#compute 42 in
  let rec loop uch : unit =
```

test.ml 2 of 6 problems

```
This expression has type [ `should_be_inp_or_output_object ]
but an expression was expected of type ('a, 'b) out ocamllsp
```

Fig. 6: Example of a format error.

For usability purposes, kmclib detects another kind of error, which we refer to as *format error*. These errors happen when the inferred type of a channel is not even in the form of output or input session type (channel *misuse*). For example, if the one drops the role name (#m) writing send uch#compute 42, the variable uch has the inferred type <compute: (int,$\cdots$) out>, which is not a session type anymore. Figure 6 shows such an error. The highlighted part is assigned a type [`shoud_be_inp_or_out_object] type saying that the expression needs another method call (or the expression should be used as an input). We are planning to improve error messages to be more descriptive, e.g., as [`role_or_label_not_given].

Note that these format checks are all done within the [%kmc.gen] (or [%kmc.check]) primitive. These errors could also be regarded as a “no instance” error in the type class, as such ill-formatted types are not in the class of k-MC systems (they are not even in the class of session type syntax).