Multiparty Session Programming with Global Protocol Combinators

Keigo Imai[1]
Gifu University, Japan
keigoi@gifu-u.ac.jp

Rumyana Neykova
Brunel University London, UK
Rumyana.Neykova@brunel.ac.uk

Nobuko Yoshida
Imperial College London, UK
n.yoshida@imperial.ac.uk

Shoji Yuen
Nagoya University, Japan
yuen@i.nagoya-u.ac.jp

Abstract

Multiparty Session Types (MPST) is a typing discipline for communication protocols. It ensures the absence of communication errors and deadlocks for well-typed communicating processes. The state-of-the-art implementations of the MPST theory rely on (1) *runtime linearity checks* to ensure correct usage of communication channels and (2) external domain-specific languages for specifying and verifying multiparty protocols.

To overcome these limitations, we propose a library for programming with *global combinators* – a set of functions for writing and verifying multiparty protocols in OCaml. Local behaviours for *all* processes in a protocol are inferred *at once* from a global combinator. We formalise global combinators and prove a sound realisability of global combinators – a well-typed global combinator derives a set of local types, by which typed endpoint programs can ensure type and communication safety. Our approach enables fully-static verification and implementation of the whole protocol, from the protocol specification to the process implementations, to happen in the same language.

We compare our implementation to untyped and continuation-passing style implementations, and demonstrate its expressiveness by implementing a plethora of protocols. We show our library can interoperate with existing libraries and services, implementing DNS (Domain Name Service) protocol and the OAuth (Open Authentication) protocol.

2012 ACM Subject Classification Software and its engineering → Concurrent programming structures; Theory of computation → Type structures; Software and its engineering → Functional languages; Software and its engineering → Polymorphism

Keywords and phrases Multiparty Session Types, Communication Protocol, Concurrent and Distributed Programming, OCaml

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.9

Related Version A full version of the paper [25] is available at `http://arxiv.org/abs/2005.06333`

Supplementary Material A source code repository for the accompanying artifact is available at `https://github.com/keigoi/ocaml-mpst/`

Acknowledgements We thank Jacques Garrigue and Oleg Kiselyov for their comments on an early version of this paper. Our work is partially supported by the first author's visitor funding to Imperial College London and Brunel University London supported by Gifu University, VeTSS, JSPS

[1] Corresponding author

© Keigo Imai, Rumyana Neykova, Nobuko Yoshida and Shoji Yuen;
licensed under Creative Commons License CC-BY
34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 9; pp. 9:1–9:30

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

KAKENHI Grant Numbers JP17H01722, JP17K19969 and JP17K12662, JSPS Short-term Visiting Fellowship S19068, EPSRC Doctoral Prize Fellowship, and EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1 and EP/T014709/1.

1 Introduction

Multiparty Session Types. Multiparty Session Types (MPST) [20, 11, 21] is a theoretical framework that stipulates how to write, verify and ensure correct implementations of communication protocols. The methodology of programming with MPST (depicted in Fig. 1(a)) starts from a communication protocol (a *global type*) which specifies the behaviour of a system of interacting processes. The local behaviour (a *local type*) for each endpoint process is then algorithmically *projected* from the protocol. Finally, each endpoint process is implemented in an endpoint host language and type-checked against its respective local type by a session typing system. The guarantee of session types is that a system of well-typed endpoint processes *does not go wrong*, i.e it does not exhibit communication errors such as reception errors, orphan messages or deadlocks, and satisfies session fidelity, i.e. the local behaviour of each process follows the global specification.

The theoretical MPST framework ensures desirable safety properties. In practice, session types implementations that enforce these properties *statically*, i.e at compile-time, are limited to binary (two party protocols) [43, 39, 31, 41]. Extending binary session types implementations to multiparty interactions, which support static linearity checks (i.e., linear usage of channels), is non-trivial, and poses two implementation challenges.

(C1) How global types can be specified and verified in a general-purpose programming language? Checking compatibility of two communicating processes relies on *duality*, i.e., when one process performs an action, the other performs a complementary (dual) action. Checking the compatibility of multiple processes is more complicated, and relies on the existence of a *well-formed* global protocol and the syntax-directed procedure of *projection*, which derives local types from a global specification. A global protocol is considered *well-formed*, if local types can be derived via projection. Since global types are far from the types of a "mainstream" programming language, state-of-the-art MPST implementations [22, 36, 47, 9] use external domain-specific protocol description languages and tools (e.g. the Scribble toolchain [50]) to specify global types and to implement the verification procedure of projection. The usage of external tools for protocol description and verification widens the gap between the specification and its implementations and makes it more difficult to locate protocol violations in the program, i.e. the correspondence between an error in the program and the protocol is less apparent.

(C2) How to implement safe multiparty communication over binary channels? The theory of MPST requires processes to communicate over multiparty channels – channels that carry messages between two or more parties; their types stipulate the precise sequencing of the communication between multiple processes. Additionally, multiparty channels has to be used linearly, i.e exactly once. In practice, however, (1) communication channels are binary, i.e a TCP socket for example connects only two parties, and hence its type can describe interactions between two entities only; (2) most languages do not support typing of linear resources. Existing MPST implementations [22, 36, 47, 9] apply two workarounds. To preserve the order of interactions when implementing a multiparty protocol over binary channels, existing works use code generation (e.g. [50]) and generate local types (APIs) for several (nominal) programming languages. Note that although the interactions order is preserved, most of these implementations [22, 36, 9] still require type-casts on the underlying

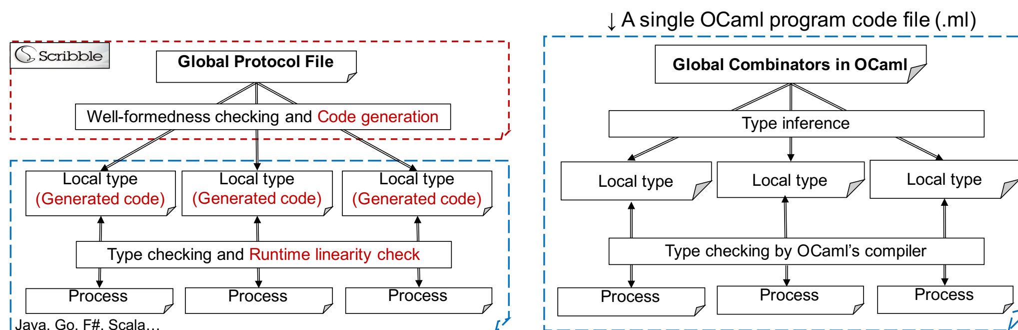

Figure 1 (a) State-of-the-art MPST implementations and (b) `ocaml-mpst` methodology

channels, compromising type safety of the host type system. To ensure linear usage of multiparty channels, *runtime checks* are inserted to detect if a channel has been used more than once. This is because the type systems of their respective host languages do not provide static linearity checking mechanism.

Our approach. This paper presents a library for programming MPST protocols in OCaml that solves the above challenges. Our library, `ocaml-mpst`, allows to specify, verify and implement MPST protocols in a single language, OCaml. Specifically, we address **(C1)** by developing *global combinators*, an embedded DSL (EDSL) for writing global types in OCaml. We address **(C2)** by encoding multiparty channels into *channel vectors* – a data structure, storing a nested sequence of binary channels. Moreover, `ocaml-mpst` verifies *statically* the linear usage of communication channels, using OCaml's strong typing system and supports session delegation.

The key device in our approach is the discovery that in a system with variant and record types, checking compatibility of local types coincides with existence of least upper bound w.r.t. subtyping relation. This realisation enables a fully static MPST implementation, i.e., static checking not only on local but also on global types in a general purpose language.

Programming with `ocaml-mpst` (Fig. 1(b)) closely follows the "top-down" methodology of MPST, but differs from the traditional MPST framework in Fig. 1(a). To use our library, a programmer specifies the global protocol with a set of global combinators. The OCaml typechecker verifies correctness of the global protocol and infers local types from global combinators. A developer implements the endpoint processes using our `ocaml-mpst` API. Finally, the OCaml type checker verifies that the API is used according to the inferred type.

The benefits of `ocaml-mpst` are that it is (1) *lightweight* – it does not depend on any external code-generation mechanism, verification of global protocols is reduced to typability of global combinators; (2) *fully-static* – our embedding integrates with recent techniques for static checking of binary session types and linearly-typed lists [27, 24], which we adopt to implement multiparty session channels and session delegation; (3) *usable* – we can auto-detect and correct protocol violations in the program, guided by OCaml programming environments like Merlin [4]; (4) *extensible* – while most MPST implementations rely on a nominal typing, we embed session types in OCaml's *structural* types, and preserve session subtyping [17]; and (5) *expressive* – we can type strictly more processes than [48] (see § 7).

Contributions. Contributions and the outline of the paper are as follows:

§ 2 gives an overview of programming with `ocaml-mpst`, a library in OCaml for specification, verification and implementations of communication protocols.

§ 3 formalises global combinators, presents their typing system, and proves a *sound realisab-*

```
1 let oAuth = (s -->c) login @@ (c -->a) pwd @@ (a -->s) auth @@ finish (* global protocol*)
```

```
2 (* The client process *)
3 let cliThread () =
4   let ch = get_ch c oAuth in
5   let `login(x, ch) = recv ch#role_S in
6   let ch = send ch#role_A#pwd "pass" in
7   close ch
8
9 (* The service process *)
10 let srvThread () =
11   let ch = get_ch s oAuth in
12   let ch = send ch#role_C#login "Hi" in
13   let `auth(_,ch) = recv ch#role_A in
14   close ch
15
16 (* The authenticator process *)
17 let authThread () =
18   let ch = get_ch a oAuth in
19   let `pwd(code,ch) = recv ch#role_C in
20   let ch = send ch#role_S#auth true in
21   close ch
22
23 (* start all processes *)
24 let () =
25   List.iter Thread.join [
26     Thread.create cliThread ();
27     Thread.create srvThread ();
28     Thread.create authThread ()]
```

Figure 2 Global protocol and local implementations for OAuth protocol [2]

ility of global combinator, i.e. a set of local types inferred from a global combinator can type a channel which embeds a set of endpoint behaviours as OCaml data structures.

§ 4 discusses the design and implementation of global combinators.

§ 5 summarises the `ocaml-mpst` communication library and explains how we utilise advanced features/libraries in OCaml to enable dynamic/static linearity checking on channels.

§ 6 evaluates `ocaml-mpst`. We compare `ocaml-mpst` with several different implementations and demonstrate the expressiveness of `ocaml-mpst` by showing implementations of MPST examples, as well as a variety of real-world protocols. We demonstrate our library can interoperate with existing libraries and services, namely we implement DNS (Domain Name Service) and the OAuth (Open Authentication) protocols on top of existing libraries.

We discuss related work in § 7 and conclude with future work in § 8. Full proofs, omitted definitions and examples can be found in [25]. Our implementation, **`ocaml-mpst`** is available at `https://github.com/keigoi/ocaml-mpst` including benchmark programs and results.

2 Overview of OCaml Programming with Global Combinators

This section gives an overview of multiparty session programming in `ocaml-mpst` by examples. It starts from declaration of global combinators, followed by endpoint implementations. We also demonstrate how errors can be reported by an OCaml programming environment like Merlin [4]. In the end of this section, we show the syntax of global combinators and the constructs of `ocaml-mpst` API in Fig. 5. The detailed explanation of the implementations of the constructs is deferred to § 4.

From global combinators to communication programs. We illustrate *global combinators* starting from a simple authentication protocol (based on OAuth 2.0 [18]). A full version of the protocol is implemented and discussed in § 6. Fig. 2 shows the complete OCaml implementation of the protocol, from the protocol specification (using global combinators) to the endpoint implementations (using `ocaml-mpst` API).

The protocol consists of three parties, a service `s`, a client `c`, and an authenticator `a`. The interactions between the parties (hereafter also called *roles*) proceed as follows: (1) the service `s` sends to the client `c` a `login` message containing a greeting (of type `string`); (2)

[2] We use a simplified syntax that support the in-built communication transport of Ocaml. For the full syntax of the library that is parametric on the transport, see the repository.

the client then continues by sending its password (`pwd`) (of type `string`) to the authenticator `a`; and (3) finally the authenticator `a` notifies `s`, by sending an `auth` message (of type `bool`), whether the client access is authorised.

The global protocol `oAuth` in Line 1 is specified using two global combinators, `-->` and `finish`. The former represents a point-to-point communication between two roles, while the latter signals the end of a protocol. The operator `@@` is a right-associative function application operator to eliminate parentheses, i.e., `(c --> a) pwd @@` *exp* is equivalent to `(c --> a) pwd (`*exp*`)`, where `-->` works as a four-ary function which takes roles `c` and `a` and label `pwd` and continuation *exp*. We assume that `login`, `pwd` and `auth` are predefined by the user as *label objects* with their *payload types* of `string`, `string` and `bool`, respectively[3]. Similarly, `s`, `c` and `a` are predefined *role objects*. We elaborate on how to define these custom labels and roles in § 4.

The execution of the `oAuth` expression returns a tuple of three *channel vectors* – one for each role in the global combinator. Each element of the tuple can be extracted using an index, encoded in role objects (`c`, `s`, and `a`). Intuitively, the role object `c` stores a functional pointer that points to the first element of the tuple, `s` points to the second, and `a` to the third element. The types of the extracted channel vectors reflect the local behaviour that each role, specified in the protocol, should implement. Channel vectors are objects that hide the *actual bare communication channels* shared between every two communicating processes.

Lines 3–21 present the implementations for all three processes specified in the global protocol. We explain the implementation for the client – `cliThread` (Lines 3–7). Other processes are similarly implemented. Line 4 extracts the channel vector that encapsulates the behaviour of the client, i.e the first element of `oAuth`. This is done by using the function `get_ch` (provided by our library) applied to the role object `c` and the expression `oAuth`.

Our library provides two main communication primitives, namely `send` and `recv`. To statically check communication structures using types, we exploit OCaml's *structural* types of objects and polymorphic variants (rather than their nominal counterparts of records and ordinary variants). In Line 5, `ch#role_S` is an invocation of method `role_S` on an object `ch`. The `recv` primitive waits on a *bare channel* returned by the method invocation. The returned value is matched against a variant tag indicating the input label `` `login `` with the pair of the payload value `x` and a continuation `ch` (shadowing the previous usage of `ch`). Then, on Line 6, two method calls on `ch` are performed, e.g `ch#role_A#pwd`, which extract a communication channel for sending a password (`pwd`) to the authenticator. This channel is passed to the `send` primitive, along with the payload value `"pass"`. Then, `let` rebinds the name `ch` to the continuation returned by `send` and on Line 7 the channel is closed. Each operation is guided by the host OCaml type system, via *channel vector type*. For example, the client channel `ch` extracted in Line 4 has a channel vector type (inferred by OCaml type checker) `` <role_S: [`login of string * ``t`] inp>` which denote reception (suffixed by `inp`) from server of a `login` label, then continuing to t, where t is `<role_A:<pwd:(string,close) out>>` denoting sending (`out`) to authenticator of a `pwd` label, followed by closing. Note that the type `<f: t>` denotes an OCaml object with a field `f` of type `t`; `` [`m of t] `` is a (polymorphic) variant type having a tag `m` of type `t`. Finally, in Lines 25–28 all processes are started in new threads.

On the expressiveness of well-typed global protocols. Fig. 3 shows two global protocols that extend `oAuth` with new behaviours. In Fig. 3a, the global combinator `choice_at` specifies a branching behaviour at role `s`. In the first case (Line 3), the protocol proceeds

[3] To be precise, the labels are *polymorphic* on their payload types which are instantiated at the point where they are used.

```
1 let oAuth2 () =
2   (choice_at s (to_s login_cancel)
3     (s, oAuth ())
4     (s, (s -->c) cancel @@
5         (c -->a) quit @@
6         finish))
```

(a) Protocol With Branching

```
1 let oAuth3 () =
2   fix (fun repeat ->
3   (choice_at s (to_s oauth2_retry)
4     (s, oAuth2 ()
5     (s, (s -->c) retry @@
6         repeat))
```

(b) Protocol With Branching & Recursion

Figure 3 Extended `oAuth` protocols

with protocol `oAuth`. In the second case (Line 5) the service sends `cancel`, to the client, and the client sends a `quit` message to the authenticator. The deciding role, `s`, is explicit in each branch. The choice combinator requires a user-defined `(to_s login_cancel)` (Line 2) that specifies concatenation of two objects for sending in branches. Its implementation is straightforward (see § 4). The protocol `oAuth3` in Fig. 3b reuses `oAuth2` and further elaborates its behaviour by offering a retry option. It demonstrates a recursive specification where the `fix` combinator binds the protocol itself to variable `repeat`.

The implementation of the corresponding client code for Fig. 3a is shown on Fig. 4a. The code is similar as before, but uses a pattern matching against multiple tags `` `login `` and `` `cancel `` to specify an *external choice* on the client, i.e the client can receive messages of different types and exhibit different behaviour according to received labels. The behaviour that a role can send messages of different types, which is often referred to as an *internal choice*, is represented as an object with multiple methods.

Our implementation also preserves the subtyping relation in session types [17], i.e the safe replacement of a channel of more capabilities in a context where a channel of less capabilities is expected. Session subtyping is important in practice since it ensures backward compatibility for protocols: a new version of a protocol does not break existing implementations. For example, the client function in Fig. 4a is typable under both protocols `oAuth2` and `oAuth3` since the type of the channel stipulating the behaviour for role `c` in `oAuth2` (receiving either message `` `login `` or `` `cancel ``) is a subtype of the channel for `c` in `oAuth3` (receiving `` `login ``, `` `cancel ``, or `` `retry ``).

Static linearity and session delegation. The implementations presented in Fig. 2, as well as Fig. 4a detect linearity violations at runtime, as common in MPST implementations [22, 47] in a non-substructural type system. We overcome this dynamic checking issue by an alternative approach, listed in Fig. 4b. We utilise an extension (`let%lin`) for linear types in OCaml [24] that statically enforces linear usage of resources by combining the usage of parameterised monads [29, 2, 40] and lenses [16]. Our library is parameterised on the chosen approach, static or dynamic. A few changes are made to avoid explicit handling of linear resources: (1) `ch` in Fig. 4b refers to a *linear* resource and has to be matched against a *linear pattern* prefixed by `#`. (2) Roles and labels are now specified as a *selector* function of the form `(fun x->x#role#label)`.

Our implementation is also the first to support *static* multiparty sessions delegation (the capability to pass a channel to another endpoint): our encoding yields it for free, via existing mechanisms for binary delegation (see § 4).

Errors in global protocol and `ocaml-mpst` endpoint programs. Our framework ensures that a well-typed `ocaml-mpst` program precisely implements the behaviour of its defined global protocol. Hence, if a program does not conform to its protocol, a compilation error is reported. Fig. 6 shows the error reported when swapping the order of send and

```
1 match recv ch#role_S with
2 |`login(pass, ch) ->
3   let ch = send ch#role_A#pwd pass
4   in close ch
5 |`cancel(_,ch) ->
6   let ch = send ch#role_A#quit ()
7   in close ch
```

(a) Dynamic Linearity Checking

```
match%lin recv ch (fun x->x#role_S) with
|`login(pass, #ch) ->
  let%lin #ch = send ch (fun x->x#role_A#pwd) pass
  in close ch
|`cancel(_, #ch) ->
  let%lin #ch = send ch (fun x->x#role_A#quit) ()
  in close ch
```

(b) Static Linearity Checking

Figure 4 Two Modes on Linearity Checking

Global Combinators to Local Types where t_i is a local type at $\mathtt{r}_i$ in g ($1 \le i \le n$)	
Global Combinator	Synopsis
`(`$\mathtt{r}_i$ `-->` $\mathtt{r}_j$`) m` g	Transmission from $\mathtt{r}_i$ to $\mathtt{r}_j$ of label `m` (with a payload).
`choice_at` r_a mrg `(`r_a`,` g_1`)` `(`r_a`,` g_2`)`	Branch to g_1 or g_2 guided by r_a.
`finish`	Finished session.
`fix (fun x ->`g`)`	Recursion. Free occurrences of x is equivalent to g itself.
Local Types and Communication Primitives	
Communication Primitive	Synopsis
`send s#role_r#m`$_k$ e	Send to role `r` label $\mathtt{m}_k$ with payload e, returning continuation.
``let `m(x, s) = receive s#role_r in`` e	Receive from `r` label `m` with payload `x` : v and continue to e with endpoint `s` : t
``match receive s#role_r with`` ``| `m``$_1$`(x`$_1$`, s) ->` e_1 `|` $\cdots$ ``| `m``$_n$`(x`$_n$`, s) ->` e_n	Receive from `r` one of labels $\{\mathtt{m}_i\}$ ($1 \le i \le n$) where payload is v_i and continue with t_i in e_i
`close s`	Closes a session

Figure 5 (a) Global Combinators (top) and (b) Communication APIs of `ocaml-mpst` (bottom)

receive actions (Lines 6 and 5) in the client implementation in Fig. 2. Similarly, errors will also be reported if we misspell any of the methods `pwd`, `role_A`, or `role_C`.

Similarly, an error is reported if the global protocol is *not safe* (which corresponds to an ill-formed MPST protocols [14]) since this may lead to *unsafe* implementations. Consider Fig. 6 (b), where we modify `oAuth2` such that `s` sends a `cancel` message to `a`. This protocol (`oAuth4`) exhibits a race condition: even if all parties adhere to the specified behaviour, `c` can send a `quit` before `s` sends `login`, which will lead to a deadlock on `s`. Our definition of global combinators prevents such ill-formed protocols, and the OCaml compiler will report an error. The actual error message reported in OCaml detects the mismatch between `a` and `c`, indicating violation of the *active role* property in the MPST literature [14] – the sender must send to the same role.

3 Formalisms and Typing for Global Combinators

This section formalises global combinators and their typing system, along a formal correspondence between a global combinator and channel vectors. The aim of this section is to provide a guidance towards descriptions of the implementations presented in § 4,5.

We first give the syntax of global combinators and channel vectors in § 3.1. We then propose a typing system of global combinators in § 3.2, illustrating that the rules check their

well-formedness. We define derivation of channel vectors from global combinators in § 3.3. The main theorem (Theorem 3.11) states that a well-typed global combinator always derives a channel vector which is typable by a corresponding set of local types, i.e. any well-typed global combinator is soundly realisable by a tuple of well-typed channel vectors.

3.1 Global Combinators and Channel Vector Types

Global combinators denote a communication protocol which describes the whole conversation scenario of a multiparty session.

▶ **Definition 3.1** (Global combinators and channel vector types). The syntax of *global combinators*, written $\mathtt{g}, \mathtt{g}', ..$, are given as:

$$\mathtt{g} \ ::= (\mathtt{p} \rightarrow \mathtt{q})\, \mathtt{m}{:}T\ \mathtt{g} \mid \mathtt{choice}\, \mathtt{p}\, \{\mathtt{g}_i\}_{i \in I} \mid \mathtt{fix}\, x \, \texttt{->}\, \mathtt{g} \mid x \mid \mathtt{finish}$$

where the syntax of *payload types* $S, T, \dots$ (also called *channel vector types*) is given below:

$$T, S \quad ::= \quad !T \mid ?T \mid \sharp T \mid T_1 \times \cdots \times T_n \mid \langle \mathtt{l}_i : T_i \rangle_{i \in I} \mid [\mathtt{l}_i_T_i]_{i \in I} \mid \mu \mathbf{t}.T \mid \mathbf{t} \mid \bullet$$

The formal syntax of global combinators comes from Scribble [50] and corresponds to the standard global types in MPSTs [37]. We assume a set of participants ($\mathfrak{R} = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \cdots\}$), and that of alphabets ($\mathbb{A} = \{\mathtt{ok}, \mathtt{cancel}, \cdots\}$). ***Communication combinator*** $(\mathtt{p} \rightarrow \mathtt{q})\, \mathtt{m}{:}T\ \mathtt{g}$ states that participant $\mathtt{p}$ can send a message of type T with label $\mathtt{m}$ to participant $\mathtt{q}$ and that the interaction described in $\mathtt{g}$ follows. We require $\mathtt{p} \neq \mathtt{q}$ to prevent self-sent messages. We omit the payload type when *unit* type $\bullet$, and assume T is *closed*, i.e. it does not contain free recursive variables. ***Choice combinator*** $\mathtt{choice}\, \mathtt{p}\, \{\mathtt{g}_i\}_{i \in I}$ is a branching in a protocol where $\mathtt{p}$ makes a decision (i.e. an output) on which branch the participants will take. ***Recursion*** $\mathtt{fix}\, x \, \texttt{->}\, \mathtt{g}$ is for recursive protocols, assuming that variables $(x, x', \dots)$ are guarded in the standard way, i.e. they only occur under the communication combinator. ***Termination*** $\mathtt{finish}$ represents session termination. We write $\mathtt{p} \in \mathrm{roles}(\mathtt{g})$ (or simply $\mathtt{p} \in \mathtt{g}$) iff, for some $\mathtt{q}$, either $\mathtt{p} \rightarrow \mathtt{q}$ or $\mathtt{q} \rightarrow \mathtt{p}$ occurs in $\mathtt{g}$.

▶ **Example 3.2.** The global combinator $\mathtt{g}_{\mathrm{Auth}}$ below specifies a variant of an authentication protocol in Fig. 3 where $T = \mathtt{string}$ and client sends auth to server, then server replies with either ok or cancel.

$$\mathtt{g}_{\mathrm{Auth}} = (\mathtt{c} \rightarrow \mathtt{s})\, \mathtt{auth}{:}T\ \big(\mathtt{choice}\, \mathtt{s}\, \{(\mathtt{s} \rightarrow \mathtt{c})\, \mathtt{ok}{:}T\ \mathtt{finish},\ (\mathtt{s} \rightarrow \mathtt{c})\, \mathtt{cancel}{:}T\ \mathtt{finish}\}\big)$$

Channel vector types abstract behaviours of each participant using standard data structure and channels. We assume *labels* $\mathtt{l}, \mathtt{l}', \dots$ range over $\mathfrak{R} \cup \mathbb{A}$. Types $!T$ and $?T$ denote

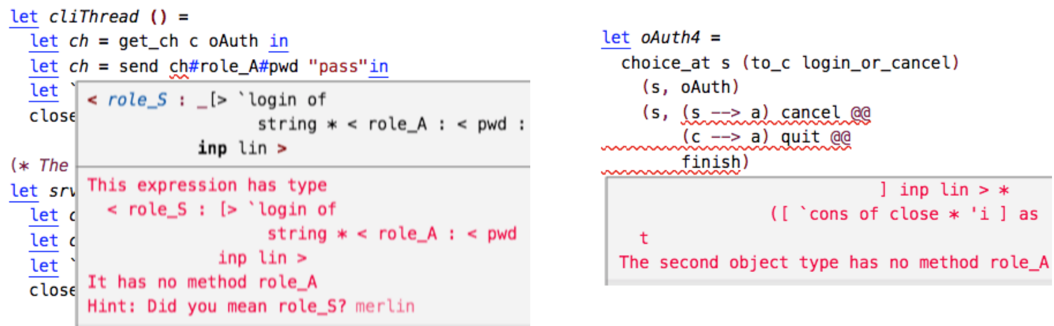

Figure 6 Type Errors Reported by Visual Studio Code (Powered by Merlin), in (a) Local Type (left) and (b) Global Combinator (right)

output and ***input channel types***, with a value or channel of type T (note that the syntax includes *session delegation*). $\sharp T$ is an ***io***-type which is a subtype of both input or output types [46]. $T_1 \times \cdots \times T_n$ is an n-ary ***tuple type***. $\langle \mathtt{l}_i : T_i \rangle_{i \in I}$ is a ***record type*** where each field $\mathtt{l}_i$ has type T_i for $i \in I$. $[\mathtt{l}_i_T_i]_{i \in I}$ is a ***variant type*** [46] where each $\mathtt{l}_i$ is a possible *tag* (or *constructor*) of that type and T_i is the argument type of the tag. In both record and variant types, we assume the fields and tags are distinct (i.e. in $\langle \mathtt{l}_i : T_i \rangle_{i \in I}$ and $[\mathtt{l}_i_T_i]_{i \in I}$, we assume $\mathtt{l}_i \neq \mathtt{l}_j$ for all $i \neq j$). The symbol $\bullet$ denotes a unit type. Type $\mathbf{t}$ is a variable for recursion. A ***recursive type*** takes an equi-recursive viewpoint, i.e. $\mu\mathbf{t}.T$ is viewed as $T\{\mu\mathbf{t}.T/\mathbf{t}\}$. Recursion variables are guarded and payload types are closed.

Channel vectors: Session types as record and variant types. The execution model of MPST assumes that processes communicate by exchanging messages over input/output (I/O) channels. Each channel has the capability to communicate with multiple other processes. *A local session type* prescribes the local behaviour for a role in a global protocol by assigning a type to the communication channel utilised by the role. More precisely, a local session type specifies the exact order and payload types for the communication actions performed on each channel (see Fig. 1(a)). In practice, processes communicate on a low-level *bi-directional* I/O channels (*bare* channels), which are used for synchronisation of two (but *not* multiple) processes. Therefore, to implement local session types in practice, a process should utilise multiple bare channels, preserving the order, in which such channels should be used. We encode local session types as channel vector types, which *wrap* bare channels (represented in our setting by $?T, !T, \sharp T$ types) in record and variant types. This is illustrated in the following table, with the corresponding local session types for reference.

Behaviour	Channel vector type	Local session type [49]
Selection (Output choice)	$\langle \mathtt{q}{:}\langle \mathtt{m}_i{:}!S_i \times T_i \rangle_{i\in I} \rangle$	$\mathtt{q} \oplus_{i \in I} \mathtt{m}_i(S_i).T_i$
Branching (Input choice)	$\langle \mathtt{q}{:}?[\mathtt{m}_i_S_i \times T_i]_{i\in I} \rangle$	$\mathtt{q} \&_{i \in I} \mathtt{m}_i(S_i).T_i$
Recursion	$\mu\mathbf{t}.T$, $\mathbf{t}$	$\mu\mathbf{t}.T$, $\mathbf{t}$
Closing	$\bullet$	**end**

Intuitively, the behaviour of sending a message is represented as a record type, which stores inside its fields a bare output channel and a continuation; the input channel required when receiving a message is stored in a variant type. Type $\langle \mathtt{q}{:}\langle \mathtt{m}_i{:}!S_i \times T_i \rangle_{i\in I} \rangle$ is read as: to send label $\mathtt{m}_i$ to $\mathtt{q}$, (1) the channel vector should be 'peeled off' from the nested record by extracting the field $\mathtt{q}$ then $\mathtt{m}_i$; then (2) it returns a pair $!S_i \times T_i$ of an output channel and a continuation. Type $\langle \mathtt{q}{:}?[\mathtt{m}_i_S_i \times T_i]_{i\in I} \rangle$ says that (1) the process extracts the value stored in the field $\mathtt{q}$, then reads on the resulting input channel ($?$) to receive a variant of type $[\mathtt{m}_i_S_i \times T_i]_{i\in I}$; then, (2) the tag (constructor) $\mathtt{m}_i$ of the received variant indicates the label which $\mathtt{q}$ has sent, and the former's argument S_i is the payload, and the latter T_i is the continuation.

The anti-symmetric structures between output types $\langle \mathtt{q}{:}\langle \mathtt{m}_i{:}!S_i \times T_i \rangle_{i\in I} \rangle$ and input types $\langle \mathtt{q}{:}?[\mathtt{m}_i_S_i \times T_i]_{i\in I} \rangle$ (notice the placements of $!$ and $?$ symbol in these types) come from the fact that an output is an *internal choice* where output labels are proactively chosen via projection on a record field, while an input is an *external choice* where input labels are reactively chosen via pattern-matching among variant constructors.

3.2 Typing Global Combinators

A key finding of our work is that compatibility of local types can be checked using a type system with record and variant subtyping. Before explaining how each combinator ensures compatibility of types, we give an intuition of well-formed global protocols following [14].

Well-formedness and choice combinator. A well-formed global protocol ensures that a protocol can be correctly and *safely* realised by a system of endpoint processes. Moreover, a set of processes that follow the prescribed behaviour is *deadlock-free*. Well-formedness imposes several restrictions on the protocol structure, notably on *choices*. This is necessary because some protocols, such as `oAuth4` in Fig. 6(b) (§ 2), are unsafe or inconsistent. More precisely, a protocol is well-formed if local types can be generated for all of its roles, i.e the *endpoint projection* function [14, Def. 3.1][25] is defined for all roles. Our encoding allows the well-formedness restrictions to be checked *statically*, by the OCaml typechecker. Below, we explain the main syntactic restrictions of endpoint projection, which are imposed on *choices* and checked statically:

R1 (**active role**) in each branch of a choice, the first interaction is from the same sender role (*active role*) to the same receiver role (*directed output*).

R2 (**deterministic choice**) output labels from an active role are pairwise distinct (i.e., protocols are deterministic)

R3 (**mergeable**) the behaviour of a role from all branches should be mergeable, which is ensured by the following restrictions:

- **M1** two input choices are merged only if (1) their sender roles are the same (*directed input*), and (2) their continuations are recursively mergeable if labels are the same.
- **M2** two output choices can be merged only if they are the same.

Intuitively, the conditions in **R3** ensure that a process is able to determine unambiguously which branch of the choice has been taken by the active role, otherwise the process should be *choice-agnostic*, i.e it should preform the same actions in all branches. Requirement **R3** is known in the MPST literature as *recursive full merging* [14].

Typing system for global combinators. Deriving channel vector types from a global combinator corresponds to the *end point projection* in multiparty session types [21]. Projection of global protocols relies on the notion of merging (**R3**). As a result of the encoding of local types as channel vectors with record and variants, the *merging* relation coincides with the *least upper bound* (join) in the subtyping relation. This key observation allows us to embed well-formed global protocols in OCaml, and check them using the OCaml type system.

Next we give the typing system of global combinators, explaining how each of the typing rules ensures the verification conditions **R1**-**R3**. The typing system uses the following subtyping rules.

▶ **Definition 3.3.** The subtyping relation $\boxed{\leqslant}$ is *co*inductively defined by the following rules.

$$\text{[Osub-}\bullet\text{]}\ \frac{}{\bullet \leqslant \bullet} \quad \text{[Osub-OutCh]}\ \frac{}{\sharp T \leqslant !T} \quad \text{[Osub-Out]}\ \frac{S \leqslant T}{!T \leqslant !S} \quad \text{[Osub-RcdDepth]}\ \frac{S_i \leqslant T_i \quad i \in I}{\langle \mathtt{l}_i : S_i \rangle_{i\in I} \leqslant \langle \mathtt{l}_i : T_i \rangle_{i \in I}} \quad \text{[Osub-Var]}\ \frac{S_i \leqslant T_i \quad i \in I}{[\mathtt{l}_i_S_i]_{i\in I} \leqslant [\mathtt{l}_i_T_i]_{i\in I\cup J}}$$

$$\text{[Osub-InpCh]}\ \frac{}{\sharp T \leqslant ?T} \quad \text{[Osub-Inp]}\ \frac{S \leqslant T}{?S \leqslant ?T} \quad \text{[Osub-Tup]}\ \frac{S_i \leqslant T_i \quad i \in I}{S_1 \times \cdots \times S_n \leqslant T_1 \times \cdots \times T_n} \quad \text{[Osub-}\mu\text{L]}\ \frac{S\{\mu \mathbf{t}.S/\mathbf{t}\} \leqslant T}{\mu \mathbf{t}.S \leqslant T} \quad \text{[Osub-}\mu\text{R]}\ \frac{S \leqslant T\{\mu \mathbf{t}.T/\mathbf{t}\}}{S \leqslant \mu \mathbf{t}.T}$$

Among those, the rules [Osub-μL] and [Osub-μR] realise equi-recursive view of types. The only non-standard rule is [Osub-RcdDepth] which does not allow fields to be removed in the super type. This simulates OCaml's lack of row polymorphism where positive occurrences of objects are not allowed to drop fields. Note that the negative occurrences of objects in OCaml, which we use in process implementations, for example, do have row polymorphism, which correspond to standard record subtyping: $\frac{S_i \leqslant T_i \quad i \in I}{\langle \mathtt{l}_i : S_i \rangle_{i\in I\cup J} \leqslant \langle \mathtt{l}_i : T_i \rangle_{i \in I}}$. We use standard record subtyping, when typing processes. Since it permits removal of fields, it precisely simulates session subtyping on outputs. Typing rules for processes are left to [25].

The typing rules for global combinators (Fig. 7) are defined by the typing judgement of the form $\Gamma \vdash_{\mathbb{R}} \mathtt{g} : T$ where Γ is a type environment for recursion variables (definition follows),

$$\frac{\text{[OTG-COMM]}\quad \Gamma \vdash_{\mathbb{R}} \mathtt{g} : (T_1 \times \cdots \times T_i \times \cdots \times T_j \times \cdots \times T_n) \quad \mathtt{p}_i, \mathtt{p}_j \in \mathbb{R}}{\Gamma \vdash_{\mathbb{R}} (\mathtt{p}_i \to \mathtt{p}_j)\, \mathtt{m}{:}S\ \mathtt{g} : \left(T_1 \times \cdots \times \langle \mathtt{p}_j{:}\langle \mathtt{m}{:}!S{\times}T_i\rangle\rangle \times \cdots \times \langle \mathtt{p}_i{:}?[\mathtt{m}_S{\times}T_j]\rangle \times \cdots \times T_n\right)}$$

$$\frac{\text{[OTG-CHOICE]}\quad \begin{array}{c}\Gamma \vdash_{\mathbb{R}} \mathtt{g}_i : T_1{\times}\cdots{\times}T_{a-1}{\times}\langle \mathtt{q}{:}\langle \mathtt{m}_k{:}!S_k{\times}T'_k\rangle_{k\in K_i}\rangle{\times}T_{a+1}{\times}\cdots{\times}T_n \\ K_j \cap K_{j'} = \emptyset \text{ for all } j \neq j' \quad \forall i \in I \quad \mathtt{p}_a \in \mathbb{R}\end{array}}{\Gamma \vdash_{\mathbb{R}} \mathtt{choice}\ \mathtt{p}_a\ \{\mathtt{g}_i\}_{i\in I} : \left(T_1{\times}\cdots{\times}T_{a-1}{\times}\langle \mathtt{q}{:}\langle \mathtt{m}_k{:}!S_k{\times}T'_k\rangle_{k\in\bigcup_{i\in I}K_i}\rangle{\times}T_{a+1}{\times}\cdots{\times}T_n\right)} \qquad \frac{\text{[OTG-}x\text{]}}{\Gamma, x{:}T \vdash_{\mathbb{R}} x : T}$$

$$\frac{\text{[OTG-finish]}}{\Gamma \vdash_{\mathbb{R}} \mathtt{finish} : \bullet{\times}\cdots{\times}\bullet} \qquad \frac{\text{[OTG-SUB]}\ \Gamma \vdash_{\mathbb{R}} \mathtt{g} : S \quad S \leqslant T}{\Gamma \vdash_{\mathbb{R}} \mathtt{g} : T} \qquad \frac{\text{[OTG-fix]}\quad \Gamma, x{:}\mathbf{t}_{x1}{\times}\cdots{\times}\mathbf{t}_{xn} \vdash_{\mathbb{R}} \mathtt{g} : T_1{\times}\cdots{\times}T_n}{\Gamma \vdash_{\mathbb{R}} \mathtt{fix}\ x \mathtt{->} \mathtt{g} : \mathrm{tfix}(\mathbf{t}_{x1}, T_1){\times}\cdots{\times}\mathrm{tfix}(\mathbf{t}_{xn}, T_n)}$$

where $\mathbb{R} = \mathtt{p}_1, \ldots, \mathtt{p}_n$ and, $\mathrm{tfix}(\mathbf{t}, \mathbf{t}') = \bullet$ and $\mathrm{tfix}(\mathbf{t}, T) = \mu\mathbf{t}.T$ otherwise.

Figure 7 The typing rules for global combinators $\boxed{\Gamma \vdash_{\mathbb{R}} \mathtt{g} : T}$

$\mathbb{R} = \mathtt{p}_1, \ldots, \mathtt{p}_n$ is the sequence of roles which participate in $\mathtt{g}$, and $T = T_1 \times \cdots \times T_n$ is a product of channel vector types where each T_i indicates a protocol which the role $\mathtt{p}_i$ must obey. We use the product-based encoding to closely model our our implementation and to avoid fixing the number of roles n of `finish` combinator by using *variable-length tuples* (see [25]).

▶ **Definition 3.4** (Global combinator typing rules). A *typing context* Γ is defined by the following grammar: $\Gamma ::= \emptyset \mid \Gamma, x{:}T$. The judgement $\Gamma \vdash_{\mathbb{R}} \mathtt{g} : T$ is defined by the rules in Fig. 7. We say $\mathtt{g}$ is *typable with* $\mathbb{R}$ if $\Gamma \vdash_{\mathbb{R}} \mathtt{g} : T$ for some Γ and T. If Γ is empty, we write $\vdash_{\mathbb{R}} \mathtt{g} : T$.

The rule [OTG-COMM] states that $\mathtt{p}_i$ has an output type $\langle \mathtt{p}_j{:}\langle \mathtt{m}{:}!S{\times}T_i\rangle\rangle$ to $\mathtt{p}_j$ with label $\mathtt{m}$, a payload typed by S and continuation typed by T_i; a dual input type $\langle \mathtt{p}_i{:}?[\mathtt{m}_S{\times}T_j]\rangle$ from $\mathtt{p}_j$ and continuation typed by T_j; and the rest of the roles are unchanged.

Rule [OTG-SUB] is the key to obtain full merging using the subtyping relation, and along with the rule [OTG-CHOICE], is a key to ensure the protocol is realisable, and free of communication errors. The rule [OTG-CHOICE] requires (1) role $\mathtt{p}_a$ to have an output type to the same destination role $\mathtt{q}$, which satisfies **R1**. The output labels $\{\mathtt{m}_k\}_{k\in K_i}$ are mutually disjoint at each branch $\mathtt{g}_i$, and are merged into a single record, which ensures that the choice is deterministic (**R2**). All other types stay the same, up to subtyping. Following requirement **M1** of **R3**, a non-directed external choices are prohibited. This is ensured by encoding the sender role of an input type as a record field, As the two different destination role labels would result in two record types with no join, following subtyping rule [OSUB-RCDDEPTH], a non-directed external choices are safely reported as a type error. Non-directed internal choices are similarly prohibited (**M2**). On the other hand, directed external choices are allowed, as stipulated by **M1**, and ensured by the subtyping relation on variant types [OSUB-VAR]. For example, the two input types $\langle \mathtt{q}{:}?[\mathtt{m}_1_S_1{\times}T_1]\rangle$ and $\langle \mathtt{q}{:}?[\mathtt{m}_2_S_2{\times}T_2]\rangle$ can be unified as $\langle \mathtt{q}{:}?[\mathtt{m}_i_S_i{\times}T_i]_{i\in 1,2}\rangle$.

The rest of the rules are standard. Rule [OTG-fix] is for recursion; it assigns the recursion variable x a sequence of distinct fresh type variables in the continuation which is later looked up by [OTG-x]. In $\mathrm{tfix}(\mathbf{t}, T)$, we assign a unit type if the role does not contribute to the recursion (i.e., $T = \mathbf{t}'$ for any $\mathbf{t}'$), or forms a recursive type $\mu\mathbf{t}.T$ otherwise.

▶ **Example 3.5** (Typing a global combinator). We show that the global combinator $\mathtt{g}_{\mathrm{Auth}} = (\mathtt{c} \to \mathtt{s})\ \mathtt{auth}\ (\mathtt{choice}\, \mathtt{s}\, \{(\mathtt{s} \to \mathtt{c})\ \mathtt{ok}\ \mathtt{finish},\ (\mathtt{s} \to \mathtt{c})\ \mathtt{cancel}\ \mathtt{finish}\})$ has the following type under $\mathtt{s}, \mathtt{c}$:

$$\langle \mathtt{c}{:}?[\mathtt{auth}_T{\times}\langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T{\times}\bullet, \mathtt{cancel}{:}!T{\times}\bullet\rangle\rangle]\rangle{\times}\langle \mathtt{c}{:}\langle \mathtt{auth}{:}!T{\times}\langle \mathtt{s}{:}?[\mathtt{ok}_T{\times}\bullet, \mathtt{cancel}_T{\times}\bullet]\rangle\rangle\rangle$$

First, see that $\mathtt{g}_1 = ((\mathtt{s} \to \mathtt{c})\ \mathtt{ok}\ \mathtt{finish})$ has a typing derivation as follows (note that we omit the payload type T in global combinators):

$$\frac{\vdash_{\mathtt{s,c}} \mathtt{finish} : \bullet \times \bullet}{\vdash_{\mathtt{s,c}} (\mathtt{s} \to \mathtt{c})\ \mathtt{ok}\ \mathtt{finish} : \langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T\times\bullet\rangle\rangle \times \langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet]\rangle}$$

For $g_2 = ((\mathtt{s} \to \mathtt{c})\ \mathtt{cancel}\ \mathtt{finish})$ we have similar derivation. Then, type of role $\mathtt{c}$ (the second of the tuple) is adjusted by [Otg-Sub], $\langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet]\rangle \leqslant \langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet, \mathtt{cancel_}T\times\bullet]\rangle$ and $\langle \mathtt{s}{:}?[\mathtt{cancel_}T\times\bullet]\rangle \leqslant \langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet, \mathtt{cancel_}T\times\bullet]\rangle$, thus we have:

$$\vdash_{\mathtt{s,c}} g_1 : \langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T\times\bullet\rangle\rangle\times\langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet, \mathtt{cancel_}T\times\bullet]\rangle$$
$$\vdash_{\mathtt{s,c}} g_2 : \langle \mathtt{c}{:}\langle \mathtt{cancel}{:}!T\times\bullet\rangle\rangle\times\langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet, \mathtt{cancel_}T\times\bullet]\rangle$$

Then, by [Otg-Choice], we have the following derivation:

$$\frac{\vdash_{\mathtt{s,c}} g_1 : \langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T\times\bullet\rangle\rangle \times \left\langle \mathtt{s}{:}?\begin{bmatrix}\mathtt{ok_}T\times\bullet,\\ \mathtt{cancel_}T\times\bullet\end{bmatrix}\right\rangle \quad \vdash_{\mathtt{s,c}} g_2 : \langle \mathtt{c}{:}\langle \mathtt{cancel}{:}!T\times\bullet\rangle\rangle \times \left\langle \mathtt{s}{:}?\begin{bmatrix}\mathtt{ok_}T\times\bullet,\\ \mathtt{cancel_}T\times\bullet\end{bmatrix}\right\rangle}{\vdash_{\mathtt{s,c}} \mathtt{choice}\,\mathtt{s}\ \{g_1, g_2\} : \langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T\times\bullet, \mathtt{cancel}{:}!T\times\bullet\rangle\rangle \times \langle \mathtt{s}{:}?[\mathtt{ok_}T\times\bullet, \mathtt{cancel_}T\times\bullet]\rangle}$$

Note that, in the above premises, the first element of the tuple specifying the behaviour of choosing role $\mathtt{s}$, namely $\langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T\times\bullet\rangle\rangle$ and $\langle \mathtt{c}{:}\langle \mathtt{cancel}{:}!T\times\bullet\rangle\rangle$, are disjointly combined into $\langle \mathtt{c}{:}\langle \mathtt{ok}{:}!T\times\bullet, \mathtt{cancel}{:}!T\times\bullet\rangle\rangle$ in the conclusion. Then, by applying [Otg-Comm] again, we get the type for g_{Auth} presented above.

3.3 Evaluating Global Combinators to Channel Vectors

Channel vectors are data structures which are created from a global combinator at initialisation, and used for sending/receiving values from/to participants. Channel vectors implement multiparty communications as nested binary io-typed channels.

▶ **Definition 3.6** (Channel vectors). *Channel vectors* $(c, c', \ldots)$ and *wrappers* $(h, h', \ldots)$ are defined as:

$$c, c' ::= \quad v, \ldots \mid s, s', \ldots \mid (c_1, \ldots, c_n) \mid [\mathtt{l}{=}c] \mid \langle \mathtt{l}_i{=}c_i\rangle_{i\in I} \mid \mu x.c \mid [s_i@h_i]_{i\in I}$$
$$h, h' ::= \quad [\,] \mid [\mathtt{l}{=}h] \mid (c_1, \ldots, h_k, \ldots, c_n) \mid \langle \mathtt{l}_1{=}c_1, \ldots, \mathtt{l}_k{=}h, \ldots, \mathtt{l}_n{=}c_n\rangle \qquad \mathtt{l} ::= \mathtt{p} \mid \mathtt{m}$$

Channel vectors c are either ***base values*** v or ***runtime values*** generated from global combinators which include ***names*** (simply-typed binary channels) $s, s', \ldots$, ***tuples*** $(c_1, \ldots, c_n)$, ***variants*** $[\mathtt{l}{=}c]$, ***records*** $\langle \mathtt{l}_i{=}c_i\rangle_{i\in I}$, and ***recursive values*** $\mu x.c$ where x is a bound variable.

We introduce an extra runtime value, ***wrapped names*** $[s_i@h_i]_{i\in I}$, inspired by Concurrent ML's `wrap` and `choose` functions [45], which are a sequence $[\ldots]_{i\in I}$ of pairs of input name s_i and a ***wrapper*** h_i. A wrapper h contains a single hole $[\,]$. An input on wrapped names $[s_i@h_i]_{i\in I}$ is *multiplexed* over the set of names $\{s_i\}_{i\in I}$. When a sender outputs value c' on name s_j $(j \in I)$, the corresponding input waiting on $[s_i@h_i]_{i\in I}$ yields a value $h_j[c']$ where the construct $h[c]$ denotes a value obtained by replacing the hole $[\,]$ in h with c (i.e. applying function h to c). We write $[\mathtt{l}_i{=}(\underline{s_i}{,}c_i)]_{i\in I}$ for $[s_i@[\mathtt{l}_i{=}([\,]{,}c_i)]]_{i\in I}$.

▶ **Definition 3.7** (Typing rules for channel vectors). Fig. 8 gives the typing rules for channel vectors and wrappers. The typing judgement for (1) channel vectors has the form $\Gamma \vdash c : T$; (2) wrappers has the form $\Gamma \vdash h : H$ where the type for wrappers is defined as $H ::= T[S]$; We assume that all types in Γ are closed.

The rules for channel vectors are standard where the subtyping relation in rule [Otc-Sub] is defined at Definition 3.3. For wrappers, rule [Otc-WrapInp] types wrapped names where the payload type S' of input channel s is the same as the hole's type, and all wrappers have the same result type T. Rule [Otc-Wrapper] checks type of a channel vector $c = h[x]$ and replaces x with the hole $[\,]$.

Evaluation of global combinators is the key to implement a multiparty protocol to a series of binary, simply-typed communications based on channel vectors. We define $[\![g]\!]^{s}_{\mathbb{R}}$ where $\mathbb{R}$

$$\frac{[\text{OTC-}s]}{\Gamma, s{:}\sharp T \vdash s : \sharp T} \quad \frac{[\text{OTC-}x]}{\Gamma, x{:}T \vdash x : T} \quad \frac{[\text{OTC-()}]}{\Gamma \vdash () : \bullet} \quad \frac{[\text{OTC-SUB}]\ \Gamma \vdash c{:}S \quad S \leqslant T}{\Gamma \vdash c : T} \quad \frac{[\text{OTC-TUP}]\ \Gamma \vdash c_i : T_i \ \forall i, 1 \le i \le n}{\Gamma \vdash (c_1, ..., c_n) : T_1 \times \cdots \times T_n}$$

$$\frac{[\text{OTC-VARIANT}]\ \Gamma \vdash c : T}{\Gamma \vdash [l{=}c] : [l_T]} \quad \frac{[\text{OTC-RECORD}]\ \Gamma \vdash c_i : T_i \ \forall i \in I}{\Gamma \vdash \langle l_i{=}c_i \rangle_{i \in I} : \langle l_i : T_i \rangle_{i \in I}} \quad \frac{[\text{OTC-}\mu]\ \Gamma, x{:}\mu\mathbf{t}.T \vdash c : T\{\mu\mathbf{t}.T/\mathbf{t}\}}{\Gamma \vdash \mu x.c : \mu\mathbf{t}.T}$$

$$\frac{[\text{OTC-WRAPINP}]\ \Gamma \vdash s_i : ?S_i \quad \Gamma \vdash h_i : T[S_i] \ \forall i{\in}I}{\Gamma \vdash [s_i @ h_i]_{i \in I} : ?T} \quad \frac{[\text{OTC-WRAPPER}]\ \Gamma, x{:}T' \vdash c : T \quad c{=}h[x] \quad x {\notin} \text{fv}(h)}{\Gamma \vdash h : T[T']}$$

Figure 8 The typing rules for channel vectors and wrappers $\boxed{\Gamma \vdash c : T}$ $\boxed{\Gamma \vdash h : H}$

$$\begin{array}{rcl}
[\![(\mathtt{p}_j \to \mathtt{p}_k)\,\mathtt{m}{:}S\ \mathtt{g}]\!]^s_{\mathbb{R}} & = & \\
\multicolumn{3}{l}{\Big([\![\mathtt{g}]\!]^s_{\mathbb{R}}(1), \dots, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(j{-}1),\ \Big\langle \mathtt{p}_k {=} \Big\langle \mathtt{m}{=}(s_{\{\mathtt{p}_j,\mathtt{p}_k,\mathtt{m},i\}}, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(j)) \Big\rangle \Big\rangle, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(j{+}1),} \\
\multicolumn{3}{l}{\quad \dots, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(k{-}1), \Big\langle \mathtt{p}_j {=} \Big[\mathtt{m}{=}(\underline{s_{\{\mathtt{p}_j,\mathtt{p}_k,\mathtt{m},i\}}}, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(k)) \Big] \Big\rangle,\ [\![\mathtt{g}]\!]^s_{\mathbb{R}}(k{+}1), \dots, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(n) \Big)} \\
& & \text{where } i \text{ is fresh.} \\
[\![\mathtt{choice}\,\mathtt{p}_a\,\{\mathtt{g}_i\}_{i \in I}]\!]^s_{\mathbb{R}} & = & \\
\multicolumn{3}{l}{\Big(\bigsqcup_{i \in I} ([\![\mathtt{g}_i]\!]^s_{\mathbb{R}}(1)), \dots, \bigsqcup_{i \in I} ([\![\mathtt{g}_i]\!]^s_{\mathbb{R}}(a{-}1)), \langle \mathtt{q}{=}\langle \mathtt{m}_k{=}c_k \rangle_{k \in K} \rangle, \bigsqcup_{i \in I} ([\![\mathtt{g}_i]\!]^s_{\mathbb{R}}(a{+}1)), \dots, \bigsqcup_{i \in I} ([\![\mathtt{g}_i]\!]^s_{\mathbb{R}}(n)) \Big)} \\
& & \text{where } \text{unfold}^*\big([\![\mathtt{g}_i]\!]^s_{\mathbb{R}}(a)\big) = \langle \mathtt{q}{=}\langle \mathtt{m}_k{=}c_k \rangle_{k \in K_i} \rangle \text{ and } K = \bigcup_{i \in I} K_i \\
[\![\mathtt{fix}\, x \,\texttt{->}\, \mathtt{g}]\!]^s_{\mathbb{R}} & = & (\text{fix}(x_1, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(1)), \dots, \text{fix}(x_n, [\![\mathtt{g}]\!]^s_{\mathbb{R}}(n))) \\
[\![x]\!]^s_{\mathbb{R}} & = & (x_1, \dots, x_n) \qquad [\![\mathtt{finish}]\!]^s_{\mathbb{R}} = ((), \dots, ())
\end{array}$$

Figure 9 Evaluation of global combinators $\boxed{[\![\mathtt{g}]\!]^s_{\mathbb{R}}}$

is a sequence of roles in $\mathtt{g}$ and s is a *base name* freshly assigned to an initiation expression at runtime. The generated channels are interconnected to each other and the created channel vectors are distributed and shared among expressions running in parallel, enabling them to interact via binary names.

The followings are basic operations on records, tuples and recursive values which are used to define evaluations of global combinators.

▶ **Definition 3.8** (Operations). **(1)** The *unfolding* $\text{unfold}^*(c)$ of a recursive value is defined by the smallest n such that $\text{unfold}^n(c) = \text{unfold}^{n+1}(c)$, and $\text{unfold}(\cdot)$ is defined as:

$$\text{unfold}(\mu x.c) \quad = \quad c\{\mu x.c/x\} \qquad\qquad \text{unfold}(c) \quad = \quad c \quad \text{otherwise}$$

where $f^{n+1}(x) = f(f^n(x))$ for $n \geq 2$ and $f^1(x) = f(x)$. **(2)** $c\#\mathtt{l}$ denotes the ***record projection***, which projects on field $\mathtt{l}$ of record value c, defined as: $\langle \mathtt{l}_i{=}c_i \rangle_{i \in I}\#\mathtt{l}_k = \text{unfold}^*(c_k)$, where $\#$ is left-associative, i.e. $c\#\mathtt{l}_1\#...\#\mathtt{l}_n = ((...(c\#\mathtt{l}_1)\#...)\#\mathtt{l}_n)$. **(3)** The i-th projection on a tuple, $c(i)$ is defined as $(c_1, \dots, c_n)(i){=}c_i$ for $1 \leq i \leq n$. **(4)** $\text{fix}(x, x'){=}()$; otherwise $\text{fix}(x, c){=}\mu x.c$.

▶ **Definition 3.9** (Evaluation of a global combinator). Given $\mathbb{R}$ and fresh s, the *evaluation* $[\![\mathtt{g}]\!]^s_{\mathbb{R}}$ of global combinator $\mathtt{g}$ is defined in Fig. 9. We write $[\![\mathtt{g}]\!]^s$ if $\mathbb{R} = \text{roles}(\mathtt{g})$.

The evaluation for communication $(\mathtt{p}_j \to \mathtt{p}_k)\,\mathtt{m}{:}S\ \mathtt{g}$ connects between $\mathtt{p}_j$ and $\mathtt{p}_k$ by the name $s_{\{\mathtt{p}_j,\mathtt{p}_k,\mathtt{m},i\}}$ by wrapping j-th and k-th channel vector with an output and an input structure, respectively. The name $s_{\{\mathtt{p}_j,\mathtt{p}_k,\mathtt{m},i\}}$ is indexed by two role names $\mathtt{p}_j$, $\mathtt{p}_k$, label $\mathtt{m}$ and an index i so that (1) it is only shared between two roles $\mathtt{p}_j$ and $\mathtt{p}_k$, (2) communication only occurs when it tries to communicate a specific label $\mathtt{m}$, and (3) both the sender and the receiver agree on the payload type. Here, the index i is used to distinguish between names generated from the same label $\mathtt{m}'$ but different payload type $\mathtt{m}{:}T$ and $\mathtt{m}{:}T'$, ensuring consistent typing of generated channel vectors. The choice combinator $\mathtt{choice}\,\mathtt{p}_a\,\{\mathtt{g}_i\}_{i \in I}$

extracts the output channel vector (i.e. the nested records of the form $\langle \texttt{q}{=}\langle \texttt{m}_k{=}c_k\rangle_{k\in K_i}\rangle$) at $\texttt{p}_a$ from each branch $\texttt{g}_i$, and merges them into a single output. Channel vectors for the other roles are merged by $c_1 \sqcup c_2$ where merging for the outputs is an intersection of branchings from c_1 and c_2, while merging of the inputs is their union. We explain merging by example (Example 3.10) and leave the full definition in [25].

For the recursion combinator, function $\text{fix}(x_i, c_i)$ forms a recursive value for repetitive session, or voids it as `()` if it does not contain any names.

▶ **Example 3.10** (Global combinator evaluation)**.** Let $s_1 = s_{\{\texttt{c},\texttt{s},\texttt{ok},0\}}$, $s_2 = s_{\{\texttt{c},\texttt{s},\texttt{cancel},0\}}$ and $s_3 = s_{\{\texttt{s},\texttt{c},\texttt{auth},0\}}$. Then:

$$
\begin{aligned}
&[\![\texttt{g}_{\text{Auth}}]\!]^s \\
&= [\![(\texttt{c} \to \texttt{s})\ \texttt{auth}\ (\texttt{choice}\,\texttt{s}\,\{(\texttt{s} \to \texttt{c})\ \texttt{ok}\ \texttt{finish},\ (\texttt{s} \to \texttt{c})\ \texttt{cancel}\ \texttt{finish}\})]\!]^s \\
&\quad \left(\begin{array}{l}
\text{Here, we have } \left\{\begin{array}{l}
\texttt{g}_L = (\texttt{c} \to \texttt{s})\ \texttt{ok}\ \texttt{finish}, \quad \texttt{g}_R = (\texttt{c} \to \texttt{s})\ \texttt{cancel}\ \texttt{finish}, \\
[\![\texttt{g}_L]\!]^s = \langle\langle \texttt{s}{=}[\texttt{ok}{=}(\underline{s_1},\texttt{()})]\rangle, \langle \texttt{c}{=}\langle \texttt{ok}{=}(s_1,\texttt{()})\rangle\rangle\rangle, \\
[\![\texttt{g}_R]\!]^s = \langle\langle \texttt{s}{=}[\texttt{cancel}{=}(\underline{s_2},\texttt{()})]\rangle, \langle \texttt{c}{=}\langle \texttt{cancel}{=}(s_2,\texttt{()})\rangle\rangle\rangle,
\end{array}\right\}, \\
\text{concatenating } \left\{\begin{array}{l}
\text{unfold}^*([\![\texttt{g}_L]\!]^s(2)) = [\![\texttt{g}_L]\!]^s(2) = \langle \texttt{s}{=}\langle \texttt{ok}{=}c_{L2}\rangle\rangle, c_{L2}{=}(s_1,\texttt{()}), \\
\text{unfold}^*([\![\texttt{g}_R]\!]^s(2)) = [\![\texttt{g}_R]\!]^s(2) = \langle \texttt{s}{=}\langle \texttt{cancel}{=}c_{R2}\rangle\rangle, c_{R2}{=}(s_2,\texttt{()})
\end{array}\right\}
\end{array}\right) \\
&= (\langle \texttt{s}{=}\langle \texttt{auth}{=}(s_3,[\![\texttt{g}_L]\!]^s(1) \sqcup [\![\texttt{g}_R]\!]^s(1))\rangle\rangle,\ \langle \texttt{c}{=}[\texttt{auth}{=}(\underline{s_3},\langle \texttt{c}{=}\langle \texttt{ok}{=}c_{L2},\texttt{cancel}{=}c_{R2}\rangle\rangle)]\rangle) \\
&= \left(\begin{array}{l}
\langle \texttt{s}{=}\langle \texttt{auth}{=}(s_3,\langle \texttt{s}{=}[\texttt{ok}{=}(\underline{s_1},\texttt{()}), \texttt{cancel}{=}(\underline{s_2},\texttt{()})]\rangle)\rangle\rangle, \\
\quad \langle \texttt{c}{=}[\texttt{auth}{=}(\underline{s_3},\langle \texttt{c}{=}\langle \texttt{ok}{=}(s_1,\texttt{()}), \texttt{cancel}{=}(s_2,\texttt{()})\rangle\rangle)]\rangle
\end{array}\right)
\end{aligned}
$$

The following main theorem states that if a global combinator is typable, the generated channel vectors are well-typed under the corresponding local types.

▶ **Theorem 3.11** (Realisability of global combinators)**.** If $\vdash_{\mathbb{R}} \texttt{g} : T$, then $[\![\texttt{g}]\!]^s_{\mathbb{R}} = c$ is defined and $\{s_i{:}S_i\}_{s_i \in \text{fn}(c)} \vdash c : T$ for some $\{\widetilde{S_i}\}$.

This property offers the type soundness and communication safety for `ocaml-mpst` endpoint programs: a statically well-typed `ocaml-mpst` program will satisfy subject reduction theorem and never performs a non-compliant I/O action w.r.t. the underlying binary channels. We leave the formal definition of `ocaml-mpst` endpoint programs, operational semantics, typing system, and the subject reduction theorem in [25].

4 Implementing Global Combinators

We give a brief overview on the type manipulation techniques that enable type checking of global combinators in native OCaml. § 4.1 gives a high-level intuition of our approach, § 4.2 illustrates evaluation of global combinators to channel vectors in pseudo OCaml code, and § 4.3 presents the typing of global combinators in OCaml. Furthermore, in [25], we develop *variable-length tuples* using state-of-art functional programming techniques, e.g., GADT and polymorphic variants, to improve usability of `ocaml-mpst`.

4.1 Typing Global Combinators in OCaml: A Summary

In Fig. 10 we illustrate the type signature of each global combinator, which is a transliteration of the typing rules (Fig. 7) into OCaml. In the figure, OCaml type $(t_{r_1} * \cdots * t_{r_n})$ corresponds to a n-tuple of channel vector types $t_{r_1} \times \cdots \times t_{r_n}$. The implementation makes use of *variable-length tuples* to represent tuples of channel vectors, and therefore the developer does not have to explicitly specify the number of roles n (see [25]). A few type-manipulation techniques are expanded later in § 4.3. Henceforth, we only make a few remarks, regarding some discrepancies with the implementation.

Global Combinator	Type
`finish`	`(close` $*$ $\cdots$ $*$ `close)`
`(`r_i `-->` r_j`)` `m` g	Given g : $(t_{r_1} * \cdots * t_{r_n})$, Return $(t_{r_1} * \cdots *$ `<`r_j`:<m:('v*`t_{r_i}`) out>>` $* \cdots *$ `<`r_i`:[>`\``m of 'v*`$t_{r_j}$`] inp>` $* \cdots * t_{r_n})$

`choice_at` r_a mrg
`(`r_a`,` g_1`)`
`(`r_a`,` g_2`)`
	Given $1 \le a \le n$,
	g_1 : $(t_{r_1} * \cdots * t_{r_{a-1}} *$ `<`r_b`:<m`$_i$`:(`v_i`,`s_i`) out>`$_{i\in I}$`>` $* t_{r_{a+1}} * \cdots * t_{r_n})$,
	g_2 : $(t_{r_1} * \cdots * t_{r_{a-1}} *$ `<`r_b`:<m`$_j$`:(`v_j`,`s_j`) out>`$_{j\in J}$`>` $* t_{r_{a+1}} * \cdots * t_{r_n})$, and
	mrg : a concatenator ensuring the two label sets are mutually disjoint ($I \cap J = \emptyset$),
	Return $(t_{r_1} * \cdots * t_{r_{a-1}} *$ `<`r_b`:<m`$_k$`:(`v_k`,`s_k`) out>`$_{k\in I\cup J}$`>` $* t_{r_{a+1}} * \cdots * t_{r_n})$

`fix (fun` x `->` g`)`	Given g : $(t_{r_1} * \cdots * t_{r_n}$ under assumption that x : $(t_{r_1} * \cdots * t_{r_n})$, x is guarded in g Return $(t_{r_1} * \cdots * t_{r_n})$
`closed_at` r_a g	Given g : $(t_{r_1} * \cdots * t_{r_{a-1}} *$ `close` $* t_{r_{a+1}} * \cdots * t_{r_n})$ and $1 \le a \le n$, Return $(t_{r_1} * \cdots * t_{r_{a-1}} *$ `close` $* t_{r_{a+1}} * \cdots * t_{r_n})$

Figure 10 Type of Global Combinators in OCaml

OCaml types	Types in § 3
`<r:[>`\``m`$_i$ `of` $v_i$`*`$t_i$`]`$_{i\in I}$ `inp>`	$\langle \mathtt{r}:?[\mathtt{m}_i_S_i \times T_i]_{i\in I}\rangle$
`<r:<m`$_i$`:(`v_i`,`t_i`) out>`$_{i\in I}$`>`	$\langle \mathtt{r}:\langle \mathtt{m}_i:!S_i \times T_i\rangle_{i\in I}\rangle$
`close` (=`unit`)	$\bullet$
t `as 'x`	$\mu x.T$

Channel vector types in OCaml. The OCaml syntax of channel vector types is given on the right. The difference with its formal counterparts are minimal. In particular, records are implemented using OCaml object types, and record fields correspond to object methods, i.e. `role_q` is a method. In type `[>`\``m`$_i$ `of` $t_i$`]`$_{i\in I}$, the symbol `>` marks an *open* polymorphic variant type which can have more tags. The types `inp` and `out` stand for an input and output types with a payload type v_i and a continuation t_i. Recursive channel vector types are implemented using OCaml equi-recursive types.

On branching and compatibility checking. As we explained in § 3.2, branching is the key to ensure the protocol is realisable, and free of communication errors. To ensure that the choice is deterministic, it must be verified that the set of labels in each branch are disjoint. Since OCaml objects do not support *concatenation* (combining of multiple methods e.g., [57, 19]), and cannot automatically verify that the set of labels (encoded as object methods) are disjoint, the user has to manually write a disjoint merge function *mrg* that concatenates two objects with different methods into one (see [25] for examples). This part can be completely automated by PPX syntactic extension in OCaml. On compatibility checking of non-choosing roles, external choice `<`r`: [>`\``m1 of` $\cdots$ `] inp>` and `<`$r$`: [>`\``m2 of` $\cdots$ `] inp>`, the types can be recursively merged by OCaml type inference to `<`$r$`: [>`\``m1 of` $\cdots$ `|`\``m2 of` $\cdots$ `] inp>` thanks to the row polymorphism on polymorphic variant types (`>`), while non-directed external choices and other incompatible combination of types (e.g., input and output, input and closing, and output and closing) are statically excluded.

On unguarded recursion. The encoding of recursion `fix (fun` x `->` g`)` has two caveats w.r.t the typing system: (1) OCaml does not check if a recursion is guarded, thus for example `fix (fun x -> x)` is allowed. We cannot use OCaml value recursion, because global combinators generate channels at run-time. (2) Even if a loop is guarded, Hindley-Milner type inference may introduce arbitrary local type at some roles. For example, consider the global protocol `fix (fun x -> (`r_a `-->` r_b`) msg x)` which specifies an infinite loop for roles $\notin \{r_a, r_b\}$, and does not specify any behaviour for any other roles. To prevent undefined behaviour, the typing rule marks the types of the roles that are not used as closed $\mathrm{tfix}(\mathbf{t}, T)$.

```
1  let (-->) r_i r_j m g =
2   (* extract the continuations *)
3    let (c_r1, c_r2, ... , c_rn) = g in
4    let s = Event.new_channel () in
5     (* create an output channel vector *)
6    let c_ri = (<r_j = <m = (s,c_ri)> >) in
7     (* create an input channel vector *)
8    let c_rj = (<r_i =
9        Event.wrap s (fun x -> `m(x,c_rj)) >) in
10   (c_r1, c_r2, ... , c_rn)
```

```
let choice_at r_a mrg g_1 g_2 =
 let (c1_r1, c1_r2, ... , c1_rn) = g_1 in
 let (c2_r1, c2_r2, ... , c2_rn) = g_2 in
 let c_ra =
   (concatenate c1_ra and c2_ra using mrg) in
 let c_r1 = merge c1_r1 c2_r1 in
 let c_r2 = merge c1_r2 c2_r2 in
 (* .. repeatedly merge each r_i ≠ r_a .. *)
 let c_rn = merge c1_rn c2_rn in
 (c_r1, c_r2, ... , c_rn)
```

Figure 11 Implementation of communication combinator and (a) branching combinator (b)

Unfortunately, in type inference, we do not have such control, and the above protocol will introduce a polymorphic type `'t`$_{r_i}$ for role $r_i \notin \{r_a, r_b\}$, which can be instantiated by *any* local type.

Fail-fast policy. We regard the above intricacies on recursion as a *fact of life* in any programming language, and provide a few workarounds. For (1), we adopt a "fail-fast" policy: Our library throws an exception if there is an unguarded occurrence of a recursion variable. This check is performed when evaluating a global combinator before any communication is started. As for (2), we require the programmer to adhere to a coding convention when specifying an infinite protocol. They have to insert additional combinator `closed_at` r_a g, which consistently instantiates type variable `'t`$_{r_a}$ with `close`, leaving other roles intact. If the programmer forgets this insertion, fail-fast approach applies, and our library throws a runtime exception before the protocol has started. In addition, self-sent messages `(r -->r)msg` for any `r` are reported as an error at runtime.

4.2 Implementing Global Combinator Evaluation

Following § 3.3, in Fig. 11, we illustrate the implementation of the global combinators, by assuming that method names and variant tags are *first class* in this pseudo-OCaml. Communication combinator `(-->)` is presented in Fig. 11 (a) where the communication combinator $((r_i$ `-->` $r_j)\ m\ g)$ yields two reciprocal channel vectors of type `<`r_j`:<`m`: (`v`,`t_{r_i}`) out>>` and `<`r_i`:[>`` ` ``m `of` v`*`t_{r_j}`] inp>`.

The implementation starts by extracting the continuations (the channel vectors) at each role (Line 3). Line 4 creates a fresh new channel `s` of a polymorphic type `'v channel` shared among two roles, which is a source of type safety regarding *payload* types. Line 6 creates an output channel vector. We use a shorthand `<`m `=` e`>` to represent an OCaml object `object method` m `=` e `end`. Thus, it is bound to `c`$_{r_i}$, by nesting the pair `(s,c`$_{r_i}$`)` inside two objects, one with a method role, and another with a method label, forming type `<`r_j`:<`m`: ('v ,`t_{r_i}`) out>>`. Similarly, Line 8 creates an input channel vector `c`$_{r_j}$, by wrapping channel `s` in a polymorphic variant using `Event.wrap` from Concurrent ML and nesting it in an object type, forming type `<`r_i`:[>`` ` ``m `of 'v*`t_{r_j}`] inp>`. This wrapping relates tag m and continuation t_j to the input side, enabling external choice when merged. Finally, the newly updated tuple of channel vectors is returned (Line 10).

Fig. 11 (b) illustrates the choice combinator `choice_at`. Line 6–9 specifies that the channel vectors at non-choosing roles are *merged*, using a `merge` function. Intuitively, `merge` does a type-case analysis on the type of channel vectors, as follows: (1) for an input channel vector, it makes an *external choice* among (wrapped) input channels, using the `Event.choose`

function from Concurrent ML; (2) for an output channel vector, the bare channel is *unified* label-wise, in the sense that an output on the unified channel can be observed on both input sides, which is achieved by having channel type around a reference cell; and (3) handling of channel vector of type `close` is trivial.

First-class methods. Method names r_i, r_j and m and the variant tag m occurring in `((`r_i `-->` r_j`)` $m\,g$`)` are assumed in § 4.1 to be first-class values. Since such behaviour is not readily available in vanilla OCaml, we simulate it by introducing the type `method_` (Line 2 in Fig. 12), which creates values that behave like method objects. The type is a record with a *constructor function* `make_obj` and a *destructor function* `call_obj` (see example in Lines 3–6). We use that idea to implement labels and roles as object methods. The encoding of local types stipulates that labels are object methods (in case of internal choice) and as variant tags (in case of external choice). Hence, the `label` type (Line 9 in Fig. 12), is defined as a pair of a first-class method, i.e using `method_`, and a *variant constructor function.* While object and variant constructor functions are needed to compose a channel vector in `(-->)`, object destructor functions are used in `merge` in `choice_at`, to extract bare channels inside an object. Variant destructors are not needed, as they are destructed via pattern-matching and merging is done by `Event.choose` of Concurrent ML. Roles are defined similarly to labels. See example in Line 15 (the full definition of `role` type is available in [25]).

4.3 Typing Global Combinators via Polymorphic Lenses

This section shows one of our main implementation techniques – the use of *polymorphic lenses* [16, 42] for *index-based updates* on tuple types. This is essential to the implementation of the typing of Fig. 10 in OCaml. To demonstrate our technique, we sketch the type of the branching combinator, in a simplified form. The types of all combinators, incorporating first-class methods and variable-length tuples, can be found in [25]. The branching combinator demonstrates our key observation that merging of local types can be implemented using row polymorphism in OCaml, which simulates the least upper bound on channel vector types.

Intuitively, a lens is a functional pointer, often utilised to access and modify elements of a nested data structure. In our implementation, lenses provide a way to *update* a channel vector in a tuple $(t_{r_1} * \cdots * t_{r_n})$. The type of the lens `('g0, 't0, 'g1, 't1) idx` itself points to an element in a specific position in a tuple, by denoting that “an element `'t0` is in a tuple `'g0`” in a type-parametric way. Furthermore, this polymorphic lens is capable to express

```
1 (* the definition of the type method_*)
2 type ('obj, 'mt) method_ = {make_obj: 'mt -> 'obj; call_obj: 'obj -> 'mt}
3 (* example usage of method_: *)
4 val login_method : (<login : 'mt>, 'mt) method_ (* the type of login_method *)
5 let login_method =
6   {make_obj=(fun v -> object method login = v end); call_obj=(fun obj -> obj#login)}
7
8 (* the definition of the type label*)
9 type ('obj, 'ot, 'var, 'vt) label = {obj: ('obj, 'ot) method_; var: 'vt -> 'var}
10 (* example usage of label *)
11 val login : (<login : 'mt>, 'mt, [> `login of 'vt], 'vt) label
12 let login = {obj=login_method; var=(fun v -> `login(v))}
13
14  (* example usage of role: *)
15  let s = {index=Zero;
16  label={make_obj=(fun v -> object method role_S=v end); call_obj=(fun o -> o#role_S)}}
```

Figure 12 Implementation of first-class methods and labels

Dynamic	Static
<role_q: <m: ('v,'t) out>>	<role_p: <m: ('v data,'t) out>> lin (base value) <role_p: <m: ('s lin,'t) out>> lin (delegation)
<role_p: [`m of 'v * 't] inp>	<role_p: [`m of 'v data * 't lin] inp lin> lin (base value) <role_p: [`m of 's lin * 't lin] inp lin> lin (delegation)
close	close lin

Figure 13 Channel Vector Types with (a) Dynamic and (b) Static Linearity Checks

updating the *type* of an element, from `'t0` in tuple `'g0` to `'t1`, which will update `'g0` itself to `'g1`. More precisely, the `idx` type has two operations:

`get: ('g0,'t0,_,_) idx -> 'g0 -> 't0` and `put: ('g0,_,'g1,'t1) idx -> 'g0 -> 't1 -> 'g1`.

For example, a lens pointing to the first element of a 3-tuple has the type `(('x*'a*'b), 'x, ('y*'a*'b), 'y) idx`.

The branching combinator `choice_at` r_a *mrg* (r_a, g_1) (r_a, g_2) is declared in following way:

```
val choice_at : ('g0, close, 'g, 'tlr) idx -> (* the index of the selecting role *)
    ('tlr, 'tl, 'tr) disj -> (* the type of disjoint merge function *)
    ('gl, 'tl, 'g0, close) idx * 'gl -> (* the type of the first tuple *)
    ('gr, 'tr, 'g0, close) idx * 'gr -> (* the type of the second tuple *)
    'g (* the type of the result tuple *)
```

The type variables in the above is resolved *a la* logic programs in Prolog, where several type variables are unified to compose a tuple type of channel vectors. It requires that both continuation tuples `'gl` and `'gr` should be of the same type, *except for* the position of active role r_a. The two `idx` types paired with continuations force this unification, by putting `close` at r_a in `'gl` and `'gr`. Thus, the result type `'g0` is shared among both lenses, so that it contains only types of non-choosing roles and `close`. Each element in `'g0` is then pairwise merged[4]. The result type of the combinator `'g` is obtained by modifying the merged tuple of channel vectors `'g0` by updating the type of the active role r_a from `close` to `'tlr`, which is the result type of the object concatenation function *mrg*. Function *mrg* takes the channel vector types for the role r_a in `g1` and `g2`, namely `'tl` and `'tr`, and returns the result type `'tlr`. The signature of the combinator also explains the extra occurrence roles paired with each branch. Since we need lens r_a within three *different instantiations* for different element types `'tl`, `'tr` and `'tlr` at the position r_a, we need three occurrences of the same lens.

5 Dynamic and Static Linearity Checks in the Communication API

To ensure that an implementation faithfully implements a well-formed, safe global protocol, MPST theory requires that all communication channels are used linearly. Similarly, the safety of our library depends on the linear usage of channels. Our library offers two mechanisms for checking that a channel is used linearly: *static* and *dynamic*. Here, we briefly explain each of these mechanisms, by comparing their API usages in Fig. 14 and types in Fig. 13, where the dynamic version stays on the left while the static one is on the right.

Dynamic Linearity Checking. Dynamic checking, where linearity violations are detected at runtime, is proposed by [55] and [22], and later adopted by [41, 47]. In `ocaml-mpst`, dynamic linearity checking is implemented by wrapping the input and output channels, with a boolean flag that is set to true once the channel has been used. If linearity is violated,

[4] We have implemented the type-case analysis for `merge` mentioned in § 4.2 via a wrapper called *mergeable* around each channel vector, which bundles a channel vector and its *merging strategy*.

i.e a channel is accessed after the linearity flag has been set to true, then an exception `InvalidEndpoint` will be raised. Note that our library correctly handles output channels between several alternatives being used *only once*; for example, from a channel vector c of type `<r:<ok:(string,close) out; cancel:(string,close) out>>`, the user can extract two channels c`#r#ok` and c`#r#cancel` where an output must take place on either of the two bare channels, but not both. In addition, our library wraps each bare channel with a *fresh* linearity flag on each method invocation, since in recursive protocols, a bare channel is often *reused*, as the formalism (§ 3) implies.

Static Linearity Checking with Monads and Lenses. The static checking is built on top of `linocaml` [24]: a library implementation of linear types in OCaml which combines the usage of *parameterised monads* [2] and polymorphic lenses (see § 4.3), to enable static type-checking on the linear usage of channels. In particular, we reuse several techniques from [24, 27]. A parameterised monad, which we model by the type `((`*pre*`,`*post*`,`*v*`) monad)`, denotes a computation of type v with a *pre*- and a *post*-condition, and they are utilised to track the creation and consumption of resources at the type level. A well-known restriction of parameterised monads in the context of session types, is that they support communication on a single channel only, and hence are incapable of expressing session delegation and/or interleaving of multiple session channels. To overcome this limitation, the *slot monad* proposed in [24, 27] extends the parameterised monad to denote *multiple* linear resources in the *pre*- and *post*-conditions. The resources are represented as a sequence, and each element is modified using polymorphic lenses [42].

We incorporate the above-mentioned techniques of `linocaml` so that, instead of having a single channel vector in the *pre* and *post* conditions, we can have a sequence of channel vectors, and we use lenses to *focus* on a channel vector at a particular *slot*. If we do not require delegation or interleaving, then the length of the sequence is one and the monadic operations always update the first element of the sequence. In particular, as in [27], if a channel is delegated i.e sent through another channel, that slot (index) of the sequence is updated to `unit`, marking it as consumed.

The `ocaml-mpst` API, for static linearity checking, is given in Fig. 14(b), where $\texttt{s}_i$, and $\texttt{s}_j$ in delegation, denote *lenses* pointing at i-th and j-th slot in the monad. The binary channels in the channel vector, used within the monadic primitives `send` and `receive`, are of the types given in Fig. 13(b). Functions `send` and `receive` both take (1) a lens s_i pointing to a channel vector; and (2) a selector function which extracts, from the channel vector at index s_i, a channel `('v data, 't`$_1$`) out` for output and `'a inp` for input. Type `data` denotes unrestricted (non-linear) payload types, whose values are matched against ordinary variables. The result of the monadic primitives is returned as a value of either type `'t lin` for output or `'a lin` for input, which is matched by `match%lin` or `let%lin`, ensuring the channels (and payloads, in case of delegation) are used linearly. A `lin` type must be matched against *lens-pattern* prefixed by `#`. Note that, `linocaml` overrides the `let` syntax and `#` pattern, in the way that `let%lin #`$\texttt{s}_i$`=`*exp* updates the index $\texttt{s}_i$, in the sequence of channel vectors, with the value returned from *exp*.

To realise session delegation, we have implemented a separate monadic primitive, `deleg_send` s_i `(fun x->x#p#l)` s_j, presented in Fig. 14(b). The primitive extracts the channel vector at position s_i and then updates the channel vector at position s_j. As a result, the slot for s_j is returned and used in further communication, the slot s_i is updated to `unit`. An example program that uses `ocaml-mpst` static API is given in Fig. 4(b).

Dynamic	Static (monadic)
`let s = send s#role_q#m v in` e	`let%lin #s`$_i$ `= send s`$_i$ `(fun x -> x#role_q#m) v in` e
`let s = send s#role_q#m s' in` e	`let%lin #s`$_i$ `= deleg_send s`$_i$ `(fun x -> x#role_q#m) s`$_j$ `in` e
`match receive s#role_p with` `` |`m ``$_1$`(x,s) ->` e_1 `` |`m ``$_2$`(s',s) ->` e_2	`match%lin receive s`$_i$ `(fun x->#role_p) with` `` |`m ``$_1$`(x,#s`$_i$`) ->` e_1 `` |`m ``$_2$`(#s`$_j$`,#s`$_i$`) ->` e_2 (delegation)
`close s`	`close` s_i

Figure 14 OCaml API for MPST with Dynamic (a) and Static (b) linearity checks

6 Evaluation

We evaluate our framework in terms of run-time performance (§ 6.1) and applications (§ 6.2, § 6.3). We compare the performance of `ocaml-mpst` with programs written in a continuation-passing-style (following the encoding presented in [53]) and untyped implementations (Bare-OCaml) that utilise popular communication libraries. In summary, `ocaml-mpst` has negligible overhead in comparison with *unsafe* implementations (Bare-OCaml), and CPS-style implementations. We demonstrate the applicability of `ocaml-mpst` by implementing a lot of use cases. In § 6.3, we show the implementation of the OAuth protocol, which is the first application of session types over `http`.

6.1 Performance

The runtime overhead of `ocaml-mpst` stems from the implementation of channel vectors, more specifically: (1) extracting a channel from an OCaml object when performing a communication action, and (2) either (2.1) dynamic linearity checks or (2.2) more closures introduced by the usage of a slot monad for static checking.

Our library is parameterised on the underlying communication transport. We evaluate its performance in case of synchronous, asynchronous and distributed transports. Specifically, we use the following communication libraries:

(1) **`ev`**: OCaml's standard `Event` channels which implements channels shared among POSIX-threads;
(2) **`lwt`**: Streams between *lightweight-threads* [56], which are more efficient for I/O-intensive application in general, and broadly-accepted by the OCaml communities, and
(3) **`ipc`**: UNIX pipes distributed over UNIX processes.

Note that **`ev`** is synchronous, while the other two are asynchronous. Also, due to current OCaml limitation, POSIX-threads in a process cannot run simultaneously in parallel, which particularly affects the overall performance of (1). As OCaml garbage collector is not a concurrent GC, only a single OCaml thread is allowed to manipulate the heap, which in general limits the overall performance of multi-threaded programs written in OCaml. For (3), we generate a single pipe for each pair of processes, and maintain a mapping between a local channel and its respective dedicated UNIX pipe. In addition, we also implement an optimised variant of `ocaml-mpst` in the case of `lwt`, denoted as **`lwt-single`** in Fig. 15; it reuses a single stream among different payload types, instead of using different channels for types. In particular, we cast a payload to its required payload type utilising `Obj.magic`, as proposed and examined by [40, 26]. Our benchmarks are generalisable because each microbenchmark exhibits the worst-case scenario for its potential source of overhead.

We compare implementations, written using (1) `ocaml-mpst` static API, (2) `ocaml-mpst` dynamic API, (3) a Bare-OCaml implementation using untyped channels as provided by the corresponding transport library, and (4) a CPS implementation, following the encoding in

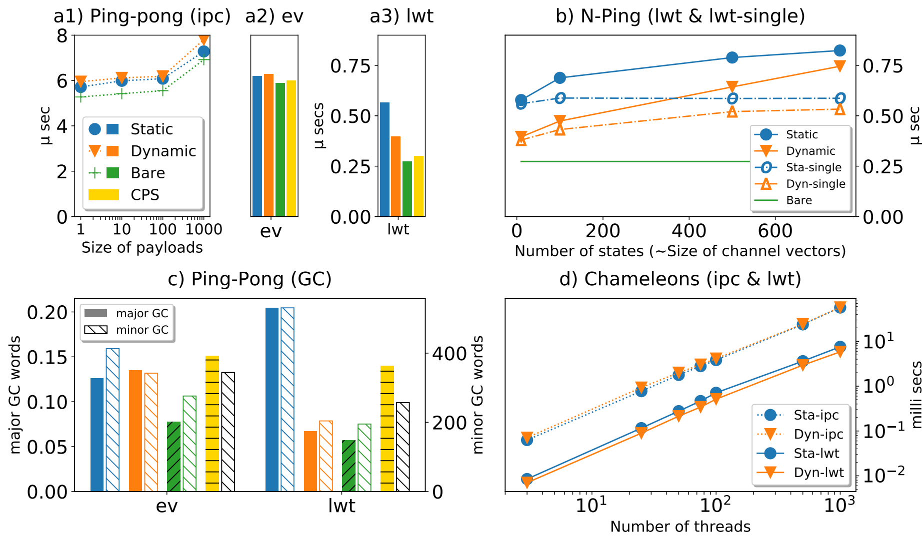

Figure 15 Runtime performance vs GC time performance

[47]. We have implemented the encoding manually such that a channel is created at each communication step, and passed as a continuation. Fig. 15 reports the results on three microbenchmarks.

Setup. We use the native *ocamlopt* compiler of OCaml 4.08.0 with Flambda optimiser[5]. Our machine configurations are Intel Core i7-7700K CPU (4.20GHz, 4 cores), Ubuntu 17.10, Linux 4.13.0-46-generic, 16GB. We use `Core_bench`[6], a popular benchmark framework in OCaml, which uses its built-in linear regression for estimating the reported costs. We repeat each microbenchmark for 10 seconds of quota where `Core_bench` takes hundreds of samples, each consists of up to 246705 runs of the targeted OCaml function, we obtain the average of execution time with fairly narrow 95% confidence interval.

Ping-pong benchmark measures the execution time for completing a recursive protocol between two roles, which are repeatedly exchanging request-response messages of increasing size (measured in 16 bit integers). The example is communication intensive and exhibits no other cost apart from the (de)serialisation of values that happens in the `ipc` case, hence it demonstrates the pure overhead of channel extraction, dynamic checks and parameterised monads. In the case of a shared memory transports (ev and lwt), we report the results of a payload of one integer since the size of the message does not affect the running time.

The slowdown of `ocaml-mpst` is negligible (approx. 5% for Dynamic vs Bare-OCaml, and 13% for Static vs Bare-OCaml) when using either `ev`, Fig. 15 (a1), or `ipc`, Fig. 15(a2), as a transport, since the overhead cost is overshadowed by latency. The shared memory case using `lwt`, Fig. 15(a3), represents the worse case scenario for `ocaml-mpst` since it measures the pure overhead of the implementation of many interactions purely done on memory with minimal latency. The slowdown in the static version is expected [27] and reflects the cost of monadic closures, as the current implementation does not optimise them away. The

[5] https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html
[6] https://blog.janestreet.com/core_bench-micro-benchmarking-for-ocaml/

linearity monad is implemented via a state monad [24], which incurs considerable overhead. The OCaml Flambda optimiser could remove more closures if we annotate the program with inlining specifications. The slowdown (although negligible) in comparison with CPS is surprising since we pre-generate all channels up-front, while the CPS-style implementation creates a channel at each interaction step. Our observation is that the compiler is optimised for handling large amounts of immutable values, while OCaml objects (utilised by the channel vector abstraction) are less efficient than normal records and variants.

Fig. 15 (c) reports on the memory consumption (in terms of words in the major and minor heap) for executing the protocol. Channel vectors with dynamic checking have approximately the same memory footprint as Bare-OCaml, and significantly less footprint when compared with a CPS implementation.

n-Ping is a protocol of increasing size, `nping` global combinator forming repeated composition of the communication combinators defined by `g`$_i$ `= (a-->b) ping @@(b-->a) pong @@g`$_{i-1}$, `g`$_0$ `= t` and `nping = fix (fun t ->g`$_n$`)`, where n corresponds to the number of `ping` and `pong` states. In contrast to Ping-Pong, this example generates a large number of channels and large channel vector objects, evaluating how well `ocaml-mpst` scales w.r.t the size of the channel vector structure. We show the results for transports `lwt` and `lwt-single` in Fig. 15 (b). The static version of `lwt-single` has a constant overhead from Bare-OCaml. Although the static checking implementation is in general slower, the relative overhead, in comparison with dynamic checking, decreases as the protocol length increases.

Chameleons protocol specifies that `n` roles ("chameleons") connect to a central broker, who picks pairs and sends them their respective reference, so they can interact peer-to-peer. The example tests delegation (central broker sends a reference) and creation of many concurrent sessions (peer-to-peer interaction of chameleons). The results reported in Fig. 15 (d) show that the implementation of delegation with static linearity checking scales as well as its dynamic counterpart. The cost of linearity (monadic closures) is less than the cost of dynamic checks for many concurrent sessions over `lwt` transport.

6.2 Use Cases

We demonstrate the expressiveness and applicability of `ocaml-mpst` by specifying and implementing protocols for a range of applications, listed in Fig. 16. We draw the examples from three categories of benchmarks: (1) *session benchmarks* (examples 1-9), which are gathered from the session types literature; (2) *concurrent algorithms* from the Savina benchmark suit [28] (examples 10-13); and (3) *application protocols* (examples 14-16), which focus on well-established protocols that demonstrate interoperability between `ocaml-mpst` implemented programs and existing client/servers. For each use case we report on Lines of Code (LoC) of global combinators and the compilation time (`CT` reported in milliseconds). We also report if the example requires full-merge [13] (FM) – a well-formedness condition on global protocols that is not supported in [47], but supported in `ocaml-mpst`.

Examples 1-9 are gathered from the official Scribble test suite[7] [52], and we have converted Scribble protocols to global protocol combinators. Examples 10-13 are concurrent algorithms and are parametric on the number of roles (n). To realise the scatter-gather pattern required in the examples, we have added two new constructs, `scatter` and `gather`, which correspond to a subset of the parameterised role extension for MPST protocols [9].

[7] https://github.com/scribble/scribble-java

Example (role)	LoC	CT(ms)	FM
1. 2-Buyer [22]	15	45	✓
2. 3-Buyer [22]	21	47	✓
3. Fibonacci [22]	8	38	×
4. SAP-Negotiation [22]	17	46	×
5. Supplier Info [22]	50	85	✓
6. SH [43, 22]	27	58	✓
7. Distributed Calc [22]	12	41	×
8. Travel Agency [22]	16	66	✓

Example (role)	LoC	CT(ms)	FM
9. Game [47]	17	49	×
10. MapReduce [28]	5	33	×
11. Nqueen [28]	12	55	×
12. Santa [38, 24]	14	42	×
13. Sleeping Barber [22]	15	43	✓
14. SMTP [22]	54	124	×
15. OAuth	26	60	✓
16. DNS	11	57	×

Figure 16 Implemented Use cases (LoC: Lines of code, CT: Compiling Time, FM: Full merge.)

To test the applicability of `ocaml-mpst` to real-world protocols we have specified, using global combinators, a core subset of three Internet protocols (examples 14-16), namely the Simple Mail Transfer Protocol (SMTP), the Domain Network System (DNS) protocol and the OAuth protocol. Using the `ocaml-mpst` APIs, it was straightforward to implement compliant clients in OCaml that interoperate with popular servers. In particular, we have implemented an SMTP client that interoperates with the Microsoft exchange server and sends an e-mail, an OAuth authorisation service that connects to a Facebook server and authenticates a client, and a DNS client and a server, which are implemented on top of a popular DNS library in OCaml (`ocaml-dns`). Note that DNS has sessions, as the DNS protocol has an ID field to discriminate sessions; and a request forwarding in the DNS protocol involves more than two participants (i.e. servers).

6.3 Session Types over HTTP: Implementing OAuth

In this section, we discuss more details about `ocaml-mpst` implementation of OAuth[8], which is an Internet standard for authentication. OAuth is commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords by providing a specific authorisation flow. Fig. 17 shows the specification of the global combinator, along with an implementation for the authorisation server. We have specified a subset of the protocol, which includes establishing a secure connection and conducting the main authentication transaction. Using `OAuth` as an example, we also discuss practically motivated extensions, *explicit connection handling* akin to the one in [23], to the core global combinators. We present that a common pattern when HTTP is used as an underlying transport.

Extension for handling stateless protocols. The protocol has a very similar structure to the `oAuth` protocol, presented in § 2. However, the original OAuth protocol is realised over a RESTful API, which means that every session interaction is either an HTTP request or an HTTP response. To handle HTTP connections, we have implemented a thin wrapper around an HTTP library, `Cohttp`[9], and we make HTTP actions explicit in the protocol by proposing two new global combinators, *connection establishing* combinator `(-!->)` and *disconnection* combinator `(-?->)`. Session types represent the types of the communication channel after a session (a TCP connection in the general case) has been established. Since RESTful protocols, realised over HTTP transport, are stateless, a connection is "established" at every HTTP Request. We explicitly encode this behaviour by replacing the –> combinator that denotes

[8] https://oauth.net/2/
[9] `https://github.com/mirage/ocaml-cohttp`

```
1 let fb_oauth =
2  (c -!-> s) (get "/start_oauth") @@
3  (s -?-> c) _302 @@ (* 302: HTTP redirect *)
4  (c -!-> a) (get "/login_form") @@
5  (a -?-> c) _200 @@
6  (c -!-> a) (post "/auth") @@
7  choice_at a (to_c success_or_fail)
8  (a,(a -?-> c) (_200_success ...) @@
9     (c -!-> s) (success is_ok "/callback") @@
10    (s -!-> a) (get "/access_token") @@
11    (a -?-> s) _200 @@
12    (s -?-> c) _200 @@
13    finish)
14 (a,(a -?-> c) (_200_fail ...) @@
15    (c -!-> s) (fail is_fail "/callback") @@
16    (s -?-> c) _200 @@
17    finish)
```

```
18 let fb_acceptor = H.start_server 8080 "/mpst-oauth"
19 let rec facebook_oauth_consumer () =
20   let ch = get_ch s fb_oauth in
21   let sid = string_of_int (Random.int ()) in
22   let conn = fb_acceptor sid in
23   let `get(_, ch) = receive (ch conn)#role_C in
24   let redir_url = fb_redirect_url sid "/callback" in
25   let ch = send ch#role_C#_302 redir_url in
26   let conn = fb_acceptor sid in
27   let ch = match receive (ch conn)#role_C with
28     | `success(_,ch) ->
29        let conn_p = H.http_connector
30            "https://graph.facebook.com/v2.11/oauth" in
31        let ch = send (ch conn_p)#role_A#get [] in
32        let `_200(auinfo,ch) = receive ch#role_A in
33        send ch#role_C#_200 "auth succeeded"
34     | `fail(_,ch) -> send ch#role_C#_200 "auth failed"
35   in close ch; facebook_oauth_consumer ()
```

Figure 17 Global Combinators and Local Implementations for OAuth (excerpt)

that one role is sending to another, with two new combinators. The combinator -!-> means establishing a connection and piggybacking a message, while -?-> denotes piggybacking a message and disconnect. This simple extension allows us to faithfully encode HTTP Request and HTTP Response. For example, `a-!->b` requires that role `a` connects on an HTTP port to `b` and then `a` sends a message to `b`, hence implementing HTTP Response; on the other hand `a-?->b` specifies an HTTP Response.

Implementation. The global combinator `fb_oauth` is given in Fig. 17 (a). As before, the protocol consists of three parties, a service `s`, a client `c`, and an authorisation server `a`. First, `c` connects to `s` via a relative path `"/start_oauth"` (Line 2). Then `s` redirects `c` to `a` using HTTP redirect code `_302` (Line 3). As a result the client sees a login form at `"/login_form"` (Lines 4-5), where they enter their credentials (Line 6). Based on the validity of the credentials received by `c`, `a` sends `_200_success` (Line 8) or `_200_fail`. If the credentials are valid, `c` proceeds and connects to `s` on path `"/callback"` (Line 9), requesting to get access to a secure page. The service `s` then retrieves an *access token* from `a` on URL `"/access_token"` (Lines 10-11), and navigates the client to an authorised page, finishing the session (Lines 12-13). If the credentials are not valid, the client reports the failure to `s` (Lines 15-16), and the session ends (Line 17).

The server role of `fb_oauth` is faithfully implemented in Lines 18-35 which provides an OAuth application utilising Facebook's authentication service. Line 18 starts a thread which listens on a port `8080` for connections. Essentially it starts a web service at an absolute URL `"/mpst-oauth"` (i.e. relative URLs like `"/callback"` are mapped to `"https://.../mpst-oauth/callback"`). The recursive function `facebook_oauth_consumer` starting from Line 19 is the main event loop for `s`. Line 20 extracts a channel vector from the global combinator `fb_oauth`, of which type is propagated to the rest of the code. Then it generates a session id via a random number generator `(Random.int ())` (Line 21), and waits for an HTTP request from a client on `fb_acceptor` (Line 22). When a client connects, the connection is bound to the variable `conn` associated with the pre-generated session id. Note that the channel vector expects a connection since no connection has been set for the client yet. Here, the connection is supplied to the channel vector via function application `(ch conn)`. On Line 24, expression `(fb_redirect_url sid "/callback")` prepares a redirect URL to an authentication page of a Facebook Provider (`https://www.facebook.com/dialog/oauth`) After sending back (HTTP Response) the redirect url to the client with `_302` label (Line 25), the connection is implicitly closed by the library. Note that we do not need to supply a connection to the

channel vector on Line 25; because a connection already exists, we have already received an HTTP request from the user and Line 25 simply performs HTTP response. The next lines proceed as expected following the protocol, with the only subtlety that we thread the connection object in subsequent send/receive calls.

The full source code of the benchmark protocols and applications and the raw data are available from the project repository.

7 Related Work

We summarise the most closely related works on session-based languages or multiparty protocol implementations. See [52] for recent surveys on theory and implementations.

The work most closely related to ours is [47], which implements multiparty session interactions over binary channels in Scala built on an encoding of a multiparty session calculus to the π-calculus. The encoding relies on *linear decomposition* of channels, which is defined in terms of *partial projection*. Partial projection is restrictive, and rules out many protocols presented in this paper. For example, it gives an undefined behaviour for role `c` and `s` for protocols `oAuth2` and `oAuth3` in Fig. 3. Programs in [47] have to be written in a continuation passing style where a fresh channel is created at each communication step. In addition, the ordering of communications across separate channels is not preserved in the implementation, e.g. sending a `login` and receiving a `password` in the protocol `oAuth` is decomposed to two separate elements which are not causally related. This problem is mitigated by providing an external protocol description language, Scribble [50], and its API generation tool, that links each protocol state using a call-chaining API [22]. The linear usage of channels is checked at runtime.

An alternative way to realise multiparty session communications over binary channels is using an orchestrator – an intermediary process that forwards the communication between interacting parties. The work [6] suggests addition of a medium process to relay the communication and recover the ordering of communication actions, while the work [7] adds annotations that permit processes to communicate directly without centralised control, resembling a proxy process on each side. Both of the above works are purely theoretical.

Among multiparty session types implementations, several works exploit the equivalence between local session types and communicating automata to generate session types APIs for mainstream programming languages (e.g., Java [22, 30], Go [9], F# [47]). Each state from state automata is implemented as a class, or in the case of [30], as a type state. To ensure safety, state automata have to be derived from the same global specification. All of the works in this category use the Scribble toolchain to generate the state classes from a global specification. Unlike our framework, a local type is not inferred automatically and the subtyping relation is limited since typing is nominal and is constrained by the fixed subclassing relation between the classes that represent the states. All of these implementations also detect linearity violations at runtime, and offer no static alternative.

In the setting of binary session types, [27] propose an OCaml library, which uses a slot monad to manipulate binary session channels. Our encoding of global combinators to simply-typed binary channels enable the reuse of the techniques presented in [27], e.g. for delegations and enforcement of linearity of channels.

FuSe [41] is another library for session programming in OCaml. It supports a runtime mechanism for linearity violations, as well as a monadic API for a single session without delegation. The implementation of FuSe is based on the encoding of binary session-typed process into the linear π-calculus, proposed by [12]. The work [48] also implements this

encoding in Scala, and the work [47] extends the encoding and implementations to the multiparty session types (as discussed in the first paragraph).

Several Haskell-based works [43, 39, 31] exploit its richer typing system to statically enforce linearity with various expressiveness/usability trade-offs based on their session types embedding strategy. These works depend on type-level features in Haskell, and are not directly applicable to OCaml. A detailed overview of the different trade-off between these implementations in functional languages is given in Orchard and Yoshida's chapter in [52]. Based on logically-inspired representation of session types, embedding higher-order binary session processes using contextual monads is studied in [54]. This work is purely theoretical.

Outside the area of session-based programming languages, various works study protocol-aware verification. Brady et al. [5] describe a discipline of protocol-aware programming in Idris, in which adherence of an implementation to a protocol is ensured by the host language dependent type system. Similarly, [51] proposes a programming logic, implemented in the theorem prover Coq, for reasoning on protocol states. A more lightweight verification approach is developed in [1] for a set of protocol combinators, capturing patterns for distributed communication. However, the verification is done only at runtime. The work [8] presents a global language for describing choreographies and a global execution model where the program is written in a global language, and then automatically projected using code generation to executable processes (in the style of BPMN). All of the above works either develop a new language or are built upon powerful dependently-typed host languages (Coq, Idris). Our aim is to utilise the MPST framework for specification and verification of distributed protocols, proposing a type-level treatment of protocols which relies solely on existing language features.

8 Conclusion and Future Work

In this work, we present a library for programming multiparty protocols in OCaml, which ensures *safe* multiparty communication over binary I/O channels. The key ingredient of our work is the notion of global combinators – a term-level representation of global types, that automatically derive channel vectors – a data structure of nested binary channels. We present two APIs for programming with channel vectors, a monadic API that enables static verification of linearity of channel usage, and one that checks channel usage at runtime. OCaml is intensively used for system programming among several groups and companies in both industry and academia [35, 3, 32, 33, 34, 15, 10, 44]. We plan to apply `ocaml-mpst` to such real-world applications.

We formalise a type-checking algorithm for global protocols, and a sound derivation of channel vectors, which, we believe, are applicable beyond OCaml. In particular, TypeScript is a promising candidate as it is equipped with a structural type system akin to the one presented in our paper.

To our best knowledge, this is the first work to enable MPST protocols to be written, verified, and implemented in a single (general-purpose) programming language and the first implementation framework of statically verified MPST programs. By combining protocol-based specifications, static linearity checks and structural typing, we allow one to implement communication programs that are extensible and type safe by design.

References

1 Kristoffer Just Arndal Andersen and Ilya Sergey. Distributed protocol combinators. In *Practical Aspects of Declarative Languages - 21th International Symposium, PADL 2019, Lisbon,*

Portugal, January 14-15, 2019, Proceedings, volume 11372 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2019. doi:10.1007/978-3-030-05998-9_11.

2 Robert Atkey. Parameterized Notions of Computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.

3 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177, 2003. URL: http://doi.acm.org/10.1145/945445.945462, doi:10.1145/945445.945462.

4 Frédéric Bour, Thomas Refis, and Gabriel Scherer. Merlin: a language server for ocaml (experience report). *PACMPL*, 2(ICFP):103:1–103:15, 2018. doi:10.1145/3236798.

5 Edwin Charles Brady. Type driven development of concurrent communicating systems. *Computer Science*, 18(3), 7 2017. doi:10.7494/csci.2017.18.3.1413.

6 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 74–95. Springer, 2016. doi:10.1007/978-3-319-39570-8_6.

7 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

8 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

9 David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, volume 3, pages 29:1–29:30. ACM, 2019.

10 Patrick Chanezon. Docker for mac and windows beta: the simplest way to use docker on your laptop, March 2016. https://blog.docker.com/2016/03/docker-for-mac-windows-beta/.

11 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015. URL: http://dx.doi.org/10.1007/978-3-319-18941-3_4, doi:10.1007/978-3-319-18941-3_4.

12 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session Types Revisited. In *PPDP '12: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, pages 139–150, New York, NY, USA, 2012. ACM. doi:10.1145/2370776.2370794.

13 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

14 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.

15 Fabrice Le Fessant. MLDonkey, 2002. http://mldonkey.sourceforge.net/.

16 J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007. doi:10.1145/1232420.1232424.

17 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebr. Meth. Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

18 Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL: https://rfc-editor.org/rfc/rfc6749.txt, doi:10.17487/RFC6749.

19 Robert Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 19x91*, pages 131–142, 1991. doi:10.1145/99583.99603.

20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. URL: http://doi.acm.org/10.1145/2827695, doi:10.1145/2827695.

22 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016. URL: http://dx.doi.org/10.1007/978-3-662-49665-7_24, doi:10.1007/978-3-662-49665-7_24.

23 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *LNCS*, pages 116–133, 2017. doi:10.1007/978-3-662-54494-5_7.

24 Keigo Imai and Jacques Garrigue. Lightweight linearly-typed programming with lenses and monads. *Journal of Information Processing*, 27:431–444, 2019. doi:10.2197/ipsjjip.27.431.

25 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators, 2020. URL: http://arxiv.org/abs/2005.06333.

26 Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: A session-based library with polarities and lenses. In *COORDINATION*, volume 10319 of *LNCS*, pages 99–118. Springer, 2017. doi:10.1007/978-3-319-59746-1_6.

27 Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: a Session-based Library with Polarities and Lenses. *Sci. Comput. Program.*, 172:135–159, 2018. doi:10.1016/j.scico.2018.08.005.

28 Shams Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE*, pages 67–80. ACM, 2014.

29 Oleg Kiselyov. Simple variable-state monad, December 2006. Mailing list message. http://www.haskell.org/pipermail/haskell/2006-December/018917.html.

30 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, pages 146–159, 2016. URL: http://doi.acm.org/10.1145/2967973.2968595, doi:10.1145/2967973.2968595.

31 Sam Lindley and J. Garrett Morris. Embedding Session Types in Haskell. In *Haskell 2016: Proceedings of the 9th International Symposium on Haskell*, pages 133–145. ACM, 2016. doi:10.1145/2976002.2976018.

32 Anil Madhavapeddy. Xen and the art of OCaml. In *Commercial Uses of Functional Programming (CUFP)*, September 2008.

33 Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014. URL: http://doi.acm.org/10.1145/2541883.2541895, doi:10.1145/2541883.2541895.

34 Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), March 2014. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

35 Yaron Minsky. OCaml for the Masses. *Commun. ACM*, 54(11):53–58, 2011. URL: http://doi.acm.org/10.1145/2018396.2018413, doi:10.1145/2018396.2018413.

36 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In

Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria, pages 128–138. ACM, 2018. doi:10.1145/3178372.3179495.

37 Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. In *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, pages 236–259, 2019. doi:10.1007/978-3-030-21485-2_14.

38 Nick Benton. Jingle Bells: Solving the Santa Claus Problem in Polyphonic C♯, 2003. Available at https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/santa.pdf.

39 Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL 2016: 43th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 568–581. ACM, 2016. doi:10.1145/2837614.2837634.

40 Luca Padovani. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming*, 27:e4, 2016.

41 Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, 2019. doi:10.1145/3229062.

42 Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming*, 1(2):Article 7, 2017. doi:10.22152/programming-journal.org/2017/1/7.

43 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell'08*, pages 25–36, New York, NY, USA, 2008. ACM. doi:http://doi.acm.org/10.1145/1411286.1411290.

44 Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, pages 377–397, 2016. URL: http://dx.doi.org/10.1007/978-3-319-47958-3_20, doi:10.1007/978-3-319-47958-3_20.

45 John H. Reppy. Concurrent ML: Design, Application and Semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, pages 165–198, 1993. doi:10.1007/3-540-56883-2_10.

46 Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

47 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP*, 2017. doi:10.4230/LIPIcs.ECOOP.2017.24.

48 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28, 2016. URL: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.21, doi:10.4230/LIPIcs.ECOOP.2016.21.

49 Alceste Scalas and Nobuko Yoshida. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–29. ACM, 2019.

50 Scribble home page, 2019. http://www.scribble.org.

51 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018. doi:10.1145/3158116.

52 António Ravara Simon Gay, editor. *Behavioural Types: from Theory to Tools*. River Publisher, 2017. URL: https://www.riverpublishers.com/research_details.php?book_id=439.

53 The Scala Development Team. The Scala Programming Language. http://scala.epfl.ch/index.html, 2004.

54 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.

55 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010. doi:10.1007/978-3-642-11957-6_29.

56 Jérôme Vouillon. Lwt: a cooperative thread library. In *Proceedings of the ACM Workshop on ML*, pages 3–12. ACM, 2008. Available at https://github.com/ocsigen/lwt. URL: http://doi.acm.org/10.1145/1411304.1411307, doi:10.1145/1411304.1411307.

57 Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991. doi:10.1016/0890-5401(91)90050-C.