Multiparty Session Type-safe Web Development with Static Linearity

Jonathan King
Imperial College London & Habito

Nicholas Ng
Imperial College London

Nobuko Yoshida
Imperial College London

Modern web applications can now offer desktop-like experiences from within the browser, thanks to technologies such as WebSockets, which enable low-latency duplex communication between the browser and the server. While these advances are great for the user experience, they represent a new responsibility for web developers who now need to manage and verify the correctness of more complex and potentially stateful interactions in their application.

In this paper, we present a technique for developing interactive web applications that are statically guaranteed to communicate following a given protocol. First, the global interaction protocol is described in the Scribble protocol language – based on multiparty session types. Scribble protocols are checked for well-formedness, and then each role is projected to a Finite State Machine representing the structure of communication from the perspective of the role. We use source code generation and a novel type-level encoding of FSMs using multi-parameter type classes to leverage the type system of the target language and guarantee only programs that communicate following the protocol will type check.

Our work targets PureScript – a functional language that compiles to JavaScript – which crucially has an expressive enough type system to provide *static* linearity guarantees. We demonstrate the effectiveness of our approach through a web-based Battleship game where communication is performed through WebSocket connections.

1 Introduction

A common trait amongst modern JavaScript-based interactive web apps is continuous stateful communication between the clients and the servers to keep the interactions responsive. This is in stark contrast to more traditional web pages, the related REST architecture [5], where a stateless HTTP request-response is sufficient to retrieve static web pages and their associated resources from the servers. These usecases led to the emergence of advanced communication transports over HTTP connections such as WebSockets [23] protocol or the WebRTC [16] project, which provide web apps with full-duplex communication channels (between browser-server and browser-browser respectively) from within the web browser. In addition to the performance improvements by reducing connections per HTTP request to a single persistent connection, they enable structured, bidirectional communication patterns not possible or convenient with only stateless HTTP connections.

As the complexity of interactions in a web app approaches that of a networked desktop application, it becomes increasingly important to ensure that the web app is free of communication errors that may lead to its incorrect execution. Hence the implementation should be verified against a protocol specification.

Consider a simple turn-based board game *Battleship* between two players. Each player starts the game by placing battle ships (contiguous rectangles) on a 2D grid, where the ships configuration is not revealed to the opponent. Players then take turns to guess the coordinate of opponent's ships, where the opponent respond if it is a *hit* or a *miss*, and the game continues until all ships of one player have been sunk. We will use this game as our running example in the rest of the paper.

F. Martins, D. Orchard (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2019)
EPTCS 291, 2019, pp. 35–46, doi:10.4204/EPTCS.291.4

© King, Ng, Yoshida
This work is licensed under the
Creative Commons Attribution License.

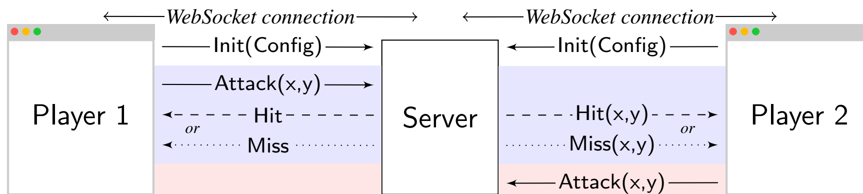

Figure 1: An architecture and message pattern of web-based *Battleship*.

A web-based implementation of the game may use the architecture depicted in Figure 1. The players' clients and the game server are connected by bidirectional WebSocket connections. Each connection follows a predefined structured communication protocol, where the sequence and the label of the messages passed between the participants are deterministic. Figure 1 also shows a snapshot of the messaging pattern between the participants, which is divided into three phases: initialisation, Player 1's turn, and Player 2's turn. During initialisation, both players send an Init(Config) message to the server with their ship configurations. The game then enters the second phase where Player 1 indicates the coordinate to attack with a Attack(x,y) message, followed by a reply from the server which is either a Hit or a Miss message. At the same time, the Server forwards the (x,y) to Player 2 as a Hit(x,y) or a Miss(x,y) message respectively. Finally, the roles of Player 1 and 2 are reversed, and the game alternates between Player 1 and 2, until a winner can be decided.

While implementations of the game may use different user interfaces (e.g. web forms, graphical with HTML5 canvas), a correct implementation of the game client should conform to the aforementioned predefined communication protocol for the communication aspects of the game. We use *Multiparty Session Types* (MPST) [10] to specify and verify communication protocols. Since our target endpoint language, JavaScript, is dynamically typed, to apply the code generation methodology from MPST, we could use a statically typed language and cross compile the language to JavaScript. Tools such as OCaml's `Js_of_ocaml` [31, 3] compiler or Haskell's GHCJS [9] compiles the respective language into JavaScript, but the generated binaries are large in size and these languages have their own runtimes on top of JavaScript which can complicate integration with existing JavaScript code. Alternatively, we can apply MPST to typed languages that are designed for JavaScript generation, examples include Microsoft's TypeScript [24] or Google's Dart [6]. Their type systems are, however, fairly basic compared to functional programming languages, which restricts the static guarantees we can provide.

In this work we use *PureScript* [29], a pure functional language inspired by Haskell, which compiles to human readable JavaScript and doesn't have a runtime. It has good library support for web development and a library for cooperatively scheduling asynchronous effects, of which further details are given in § 3.2.

Contributions This paper presents a type-safe web application development work flow following the MPST framework: **(1)** A first *multiparty* session-based code generation work flow targeting interactive web applications; **(2)** A novel encoding of Endpoint FSMs [11] using multi-parameter type classes; and **(3)** A lightweight session runtime using the encoding which also statically prevents non-linear usage of communication channels.

Figure 2 presents our proposed type-safe web application development work flow. We implement our web application development framework in PureScript on the top of the Scribble framework.

2 The Scribble protocol language

Our development work flow extends Scribble [2, 32], a protocol specification language and code generation framework based on MPST. Development starts by specifying the overall communication structure

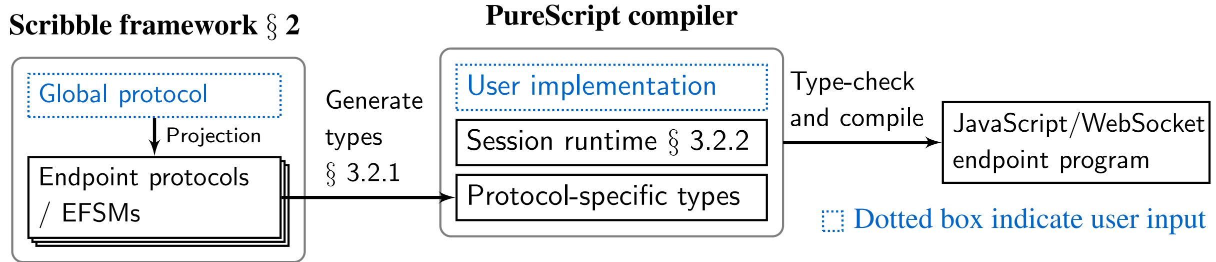

Figure 2: Overview of development work flow.

of the target application as a *global* protocol in Scribble, which is then validated by the Scribble tool chain to ensure the protocol is well-formed. Scribble protocols are organised into *modules* where each module contains declaration of *message payload types*, and one or more global protocol definitions. We explain the syntax and structure of Scribble protocols using the Battleship game Scribble protocol in Listing 1. First, we declare the module of the protocol with the keyword `module` and the module name (e.g. line 1). It is followed by message type declaration statements using the `type ... as` keywords. Messages in Scribble is written as `Label(payloads)`, where `Label` is an identifying label for the message, and `payloads` is a list of types for the payloads of the message. For example, line 3 declares a new payload type named `Location`, and specifies that the concrete type corresponds to a PureScript data type `Game.Battleships.Location`. It is later used on line 6 in the protocol body as the payload type of the `Attack(Location)` message. By associating the message types used in a protocol with a concrete type from the implementation language, messages specified in the protocol can be verified against the implementation of the protocol, which we will discuss in more details in the next section (§ 3.2.1).

```
1  module Game;
2  type <purescript> "Config" from "Game.BattleShips" as Config; // Ship configuration
3  type <purescript> "Location" from "Game.BattleShips" as Location; // Ship position
4
5  global protocol Game(role Atk, role Svr, role Def) {
6      Attack(Location) from Atk to Svr;
7      choice at Svr { // Svr knows if it's a hit
8          Hit(Location) from Svr to Atk; Hit(Location) from Svr to Def;
9          do Game(Def, Svr, Atk);
10     } or {
11         Miss(Location) from Svr to Atk; Miss(Location) from Svr to Def;
12         do Game(Def, Svr, Atk);
13     } or {
14         choice at Svr {
15             Sunk(Location) from Svr to Atk; Sunk(Location) from Svr to Def;
16             do Game(Def, Svr, Atk);
17         } or {
18             Winner() from Svr to Atk; Loser() from Svr to Def;
19         }
20     }
21 }
```

Listing 1: Main body of the Battleships protocol.

Line 5 defines the global protocol `Game` with three roles: an attacker (`Atk`), the server (`Svr`), and a defender (`Def`). In the body of the protocol, `Atk` first sends a coordinate to attack (i.e. a `Attack(Location )` message) to `Svr`, using a message passing statement on line 6. After receiving the message, assuming `Svr` holds the coordinates of the ship configurations of both the players, `Svr` will decide whether the

coordinate is a hit or a miss. Depending on the outcome, the protocol will exhibit alternative behaviours. For instance, if it is a *hit*, `Svr` will send a `Hit` message to `Atk` to notify it of the outcome, and also a `Hit` message with the coordinate being attacked to `Def`; similarly if it is a *miss*, `Miss` message will be sent instead. This is written in Scribble as a `choice` statement (line 7–20). The syntax `choice at Svr` means the choice is being made at the role `Svr`, and also that `Svr` will the first sender of message in each of the possible branches: the *hit* branch on lines 8–9 or the *miss* branch on lines 11–12. At the end of the *hit* branch (line 9), we use a `do` statement to recursively call the `Game` protocol to move to the next round of the game. A `do` statement includes role parameters to the protocol, where in this branch the current `Atk` role and `Def` role continues as `Atk` and `Def` respectively, such that the attacker can launch consecutive attacks if they had been hit. However, in the last line of the *miss* branch (line 12), `Atk` and `Def` are swapped, meaning that if there is a miss, then the current defender gets a chance to attack in the next round of game. Notice that in our protocol, we also describe a third branch in addition to the *hit* branch and *miss* branch on lines 14–19. This branch describes the situation when a battleship is sunk (i.e. all coordinates of a ship are hit). There are two possible outcomes, so we use a nested `choice` statement to describe the two branches: the first branch is the game continues as a *hit* branch, but with a `Sunk` message in place of a `Hit` message; the other branch is the endgame – i.e. all battleships of one player are sunk – in this branch, a winner is declared with a `Winner` and `Loser` message, and there are no `do` statements in this branch as the game ends immediately with no need for a next round of game.

We use the Scribble tool chain to check that the Scribble protocol is well-formed. For example, a `choice` statement is well-formed only if the first message of all branches are sent by the choice maker, otherwise the roles may be left in an inconsistent state. A well-formed global protocol can then be *projected* automatically into an *endpoint protocol* for each role in the protocol. An endpoint protocol is a localised version of the global protocol which includes only the interactions if they directly involve the target role. Scribble can represent an endpoint protocol as an equivalent *Endpoint Finite State Machine* (EFSM), where the communication interactions of the protocol are represented by transitions in the EFSM between the protocol states. We use the definitions of EFSMs from Scribble in [11] as a basis.

3 Endpoint programming with EFSMs

The EFSMs derived from the global protocol represent the local communication behaviour at each of the endpoints, and is used as a guidance for developers to implement their application endpoint. To integrate the EFSMs in the user's programming work flow, our approach interprets states and transitions in the EFSMs as (uninhabited) *types* and type-class *instances* in the target programming language. We first present an encoding of the types of ESFM transition such as **send** and **receive** as type classes (§ 3.1), then we apply the encoding to generate types and instances for the user's application (§ 3.2.1).

3.1 Transitions as Type Classes

A key contribution of this paper is our encoding of EFSMs from a Scribble protocol into type checking constraints with *multi-parameter type classes* (MPTCs). Type classes [8] were introduced to allow functions to be overloaded, where a type is polymorphic but constrained to be an *instance* of the class. Examples of common type classes include `Eq a`, `Show a` and `Monoid a`. MPTCs [15] extend this by allowing more than one type parameter in a type class definition. Combined with *functional dependencies* between type parameters, which describe that a subset of the parameters uniquely determine another, it is possible to encode *relations* between types.

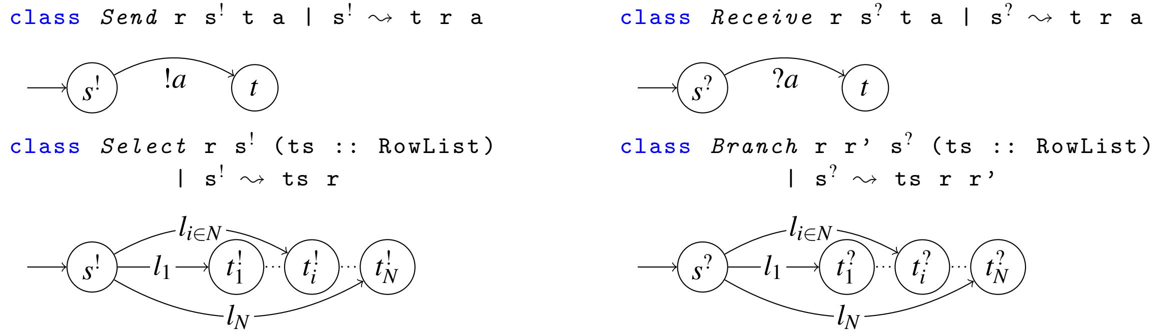

Figure 3: Send (top left), Receive (top right), Label selection (bottom left), and Label branching (bottom right) transition type classes and their state diagrams.

By representing kinds of transitions as MPTCs, it is possible to both constrain the usage of the corresponding functions and bring into context additional information about the transition, such as the next state, the type of the value communicated and the role involved. Our encoding exploits the properties of EFSMs derived from a well-formed protocol by the Scribble toolchain. The properties guaranteed by EFSMs as noted in [11] include: (1) there is exactly one initial state; (2) there is at most one terminal state; and (3) every state in an EFSM is one of three kinds: an *output (resp. input) state* where every transition is an output (resp. input) or a terminal state.

We first consider EFSM transitions where the current state has only a single transition (hence a single successor state). Figure 3 (top row) illustrates the type class definitions for output and input states. The type parameters s, t, and a are the type representing the *current* and the *successor* state of the transition, and the message *payload type* of the output and input action. Our current state s determines all other parameters, which the functional dependency $s \rightsquigarrow t\ r\ a$ describes. A similar pair of transition type classes exist for `Connect` and `Disconnect` for establishing and terminating connections (i.e. connection actions from [12]) respectively but omitted here due to space constraints.

For output $S^!$ and input states $S^?$ in the EFSM which have multiple transitions and successor states (i.e. branching and selection), each of the transitions can be identified by their message payload *label*, used for determining the selected branch between the sender and the receiver. Our encoding makes the label selection explicit, by splitting each of the output (or input) transitions into two parts: a label send (resp. receive), then the actual output (resp. input) action. A new intermediate state T is introduced between the two transitions and acts as the output (or input) state for the actual output (resp. input) transition for each branch, such that transitions of T can be encoded into `Send` or `Recv` type classes above. The original multi-transition states are no longer output and input states after the transformation, as the transitions now perform label send and receive instead of output and input actions. We encode the set of *label send* and *receive* transitions from the same state as `Branch` and `Select` type classes, as depicted in Figure 3 (bottom row) for a type-safe mapping between the labels and the chosen branches. The type parameter s is the initial state of the transition, and ts is a *row list* of tuples $(l_i, t_i)_{i \in |ts|}$ containing type-level string label l_i, and its corresponding continuation state t_i for the branch. Instances of ts are used to express the finite number of possible branches in the EFSM, such that branches with undefined labels in the EFSM cannot be used. Similar to the `Send` and `Recv` type classes, the type classes are annotated with their functional dependencies, indicating that instances of the initial state s uniquely determines ts.

```
import Game.BattleShips (Config, Location)
...
data Init = Init Config
...
foreign import data BattleShips :: Protocol
foreign import data P2 :: Role
instance roleNameP2 :: RoleName P2 "P2"
...
foreign import data S34 :: Type
foreign import data S34Connected :: Type
foreign import data S36 :: Type
...
instance initialP2 :: Initial P2 S34
instance terminalP2 :: Terminal P2 S35
instance connectS34 :: Connect P2 GameServer S34 S34Connected
instance sendS34 :: Send GameServer S34Connected S36 Init
instance branchS36 :: Branch P2 GameServer S36
  (Cons "loser" S36Loser (Cons "miss" S36Miss (Cons "hit" S36Hit Nil)))
...
```

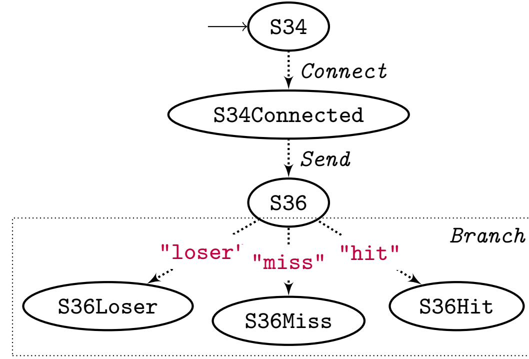

Figure 4: Example fragment of generated types and its corresponding EFSM.

3.2 Implementation

3.2.1 Types generation

Given a valid Scribble protocol, our framework generates from the corresponding EFSM a new data type to represent each state, and the transitions between the states are instantiated as type instances of type classes described in § 3.1. The types generated by the framework is a static guidance for users to use the **session runtime** to perform communication in way that conforms to the input protocol. A fragment of the EFSM and its corresponding generated types for the Battleships protocol is given in Figure 4. Message payload types are imported to the PureScript module from the specified path in the protocol, and are defined by the user. For example, the `Config` type in Figure 4 corresponds to the `type` declaration of the same name in the protocol (line 2 in Listing 1), imported from user-defined `Game.Battleships.Config`. `Role` and `RoleName` are also generated from the protocol, where the latter provides metadata (a symbol representation) of a role, and is used by the runtime as a key for accessing communication channels. Finally, the type class instances `initialP2` and `terminalP2` are the initial and terminal transitions (no predecessor and successor states respectively); `connectS34`, `sendS34` and `branchS36` are normal transitions for `connect`, `send` and `branch` actions. For each `branch` transition, a `RowList` (a way to inductively represent a row type) is used to describe labels that can be chosen from and their corresponding successor, e.g. `loser` label corresponds to the `S36Loser` successor state.

3.2.2 Session runtime

The session runtime is a library of communication combinators that can only be used to construct correct protocol implementations. These are: `connect`/`disconnect` (for managing connections between participants), `send`/`receive` (for point-to-point message passing), `choice`/`branch` (for branching and selection) and `session` (for running the session).

```
newtype Session m c i t a = Session ((Channels c i) -> m (Tuple (Channels c t) a))

bind :: Monad m => Session m c i s a -> (a -> Session m c s t b) -> Session m c i t b
```

A `Session` (above) is a continuation that consumes a `Channel` type, indexed by an *initial* state channel type `s`, and (effectfully) produces a channel in a *terminal* state `t` with a result `a`. This definition

```
session :: forall r c p s t m a.
     Transport c p
  => Initial r s
  => Terminal r t
  => MonadAff m
  => Proxy c
  -> Role r
  -> Session m c s t a
  -> m a
```

```
choice :: forall r r' rn c s ts u funcs row m p a.
     Branch r r' s ts
  => RoleName r' rn
  => IsSymbol rn
  => Terminal r u
  => Transport c p
  => Continuations (Session m c) ts u a funcs
  => ListToRow funcs row
  => MonadAff m
  => Record row -> Session m c s u a
```

```
send :: forall r rn c a s t m p.
     Send r s t a
  => RoleName r rn
  => IsSymbol rn
  => Transport c p
  => EncodeJson a
  => MonadAff m
  => a -> Session m c s t Unit
```

```
receive :: forall r rn c a s t m p.
     Receive r s t a
  => RoleName r rn
  => IsSymbol rn
  => Transport c p
  => DecodeJson a
  => MonadAff m
  => Session m c s t a
```

Figure 5: Sample types and primitives in the session runtime.

is not exported outside of the module, so the only way to construct a `Session` is through using one of the communication combinators. We can compose two sessions where the terminal state of the first is the initial of the second, resulting in a session that starts at initial state of the first and ends in the terminal state of the second using (an indexed [1]) `bind`. Given a session whose initial and terminal states match that of a protocol (provided by the `Initial` and `Terminal` constraints) using `session` we can 'run' it (to produce a monadic value). The user is not required to provide concrete states, as the type checker can determine them by solving the constraints.

In Figure 5 we provide the types of some session combinators. `send` can be read as "given a value `a` can be encoded as JSON and there is a role `r` you can send the value to, by transitioning from state `s` to `t`, then it will produce the session that starts at `s` and terminates at `t` producing the `Unit` value". `receive` is similar except instead the session *produces* the value received. `choice` can be read as "given a record providing continuations for each of the branches `ts` from the state `s` that reach the terminal state `u` producing a value `a`, then it will produce a session that starts at `s` and terminates at `u` producing `a`". The runtime will then select the appropriate continuation based on the message received. The `MonadAff` class used by all of the combinators can be thought of like `MonadIO` in Haskell and allows the asynchronous communication effects to be *lifted* into an arbitrary monad stack. A concrete example of their use in our framework is shown in § 4.

Implementing branching In our `choice` combinator we need to decode the JSON, which we will apply the continuation to, however we only know what type it should be decoded to at runtime based on the message we receive. One solution would be to manually decode the JSON in the continuation, however this is unsatisfactory as the runtime library should consistently abstract this. We solve this by inserting `Receive` states, which a branch continuation must begin with – this works because we statically know all possible types the value could be and enumerate them all. This is like receiving a label selecting the branch, followed by the value, however in practice a single message is communicated. The runtime selection relies on the JSON encoding of message data type, however as these are generated this is safe.

Transport abstraction In our example communication is performed over WebSocket connections, however this can be generalised to a reliable, order-preserving asynchronous channels. The type variable `c` in `Session` and `Channel` allows this parameterisation. Communication in our runtime is implemented

through abstract primitives defined in a `Transport` type class, allowing library users to provide their own *transport* layer. With this abstraction we have additionally implemented support for using `AVar`'s (similar to Haskell's `MVar`) to perform communication locally through shared memory.

Linear usage of channel Through our careful choice of library design, inspired by existing work using Indexed Monads [1], channels are not directly accessed by the programmer. This means *reuse* is impossible and by requiring continuation to a terminal state to 'run' a session, *use* is guaranteed. A faulty runtime implementation could still violate this, however this only has be to verified once by the library author rather than the user.

4 Case study

We present an implementation of the Battleship running example in PureScript using our framework. The full implementation can be found in [14]. Most web frameworks (e.g. React or Halogen) are event driven, where an *update* function handles events fired by user interaction. There is no knowledge however about the order in which events are received, or that only a subset of events are possible in a given state (e.g. if currently a button is disabled). This means that it is not possible to have a session that spans more than one event in an event driven framework, while still preserving static linearity guarantees by construction. We therefore use the Concur UI framework that constructs UIs sequentially, which is a perfect fit for inherently sequential sessions.

Widgets Concur is built around composing Widgets. A Widget is something that has a view, can internally update in response to some events, and will return some value. Consider the following snippet of code, which defines a `play` button that displays `"Play game"` and returns once is has been pressed.

```
play :: forall a. Widget HTML Unit
play = button' [unit <$ onClick] [text "Play game"]
```

We can then sequence this button with the text `"Game over!"`, which will be displayed *after* the button has been pressed. Note that the type `"forall a. a"` means this widget can never return (as it is impossible to produce a value of this type) and so will be displayed forever.

```
example :: forall a. Widget HTML a
example = do
  play
  text "Game over!"
```

Lifting widgets Our runtime combinators are parameterised over any `MonadAff`[1] so that we can pick `Widget HTML` from the Concur UI framework. `lift` lets us *lift* a widget into a session, which produces a value while remaining in the same session state.

```
lift :: forall c i f a. Functor f => f a -> Session f c i i a
```

We build a widget that plays the game as Player 1, with interleaved sessions and user input. We begin by first connecting to the `GameServer`, followed by running the `setupGameWidget` to allow the user to place their ship. This configuration is then sent to the `GameServer` and the player attacks. (Note: we use PureScript's support for rebinding `bind` and `pure` in a `do` block).

```
battleShipsWidgetP1 url :: URL -> Widget HTML Unit
battleShipsWidgetP1 = session
  (Proxy :: Proxy WebSocket)
```

[1]The `MonadAff` class is an asynchronous effect monad, in some respects similar to `MonadIO` in Haskell.

```
(Role :: Role P1) $ do
  connect (Role :: Role GameServer) url
  config <- lift setupGameWidget
  send $ Init config
  let pb = mkBoard config
  let ob = mempty :: Board OpponentTile
  attack pb ob
```

Limitations It is difficult to extract common 'source code' to a single function, as its type is dependent on where it is used in the protocol. This can result in duplicated identical code, but with a different type signature. Technically it is possible to write out the state polymorphic version, however you would need to chain all the FSM constraints required by the runtime combinators which is tedious. This is a fundamental limitation of the FSM representation compared to actually embedding session types in the language. Additionally when splitting an implementation into multiple functions the user will need to provide types for the initial and terminal states of each sub-protocol. For the same practical reasons, this will be a concrete generated type (e.g. `S20`), meaning any changes to the Scribble protocol will require the type to be updated manually.

5 Related work

Scribble-based code generation Code generation from Scribble is an effective way of applying multiparty session types in mainstream programming languages.

Notably, Scribble-Java [11, 12] was the earliest work to propose hybrid session verification, by generating Java API from Scribble to statically type check user's I/O action usages against the generated APIs, combined with runtime checking of linear channel usages. The work focussed on desktop applications and only support TCP and HTTP as the communication transport. Scribble-Scala [30] uses Scribble to generate Scala APIs that use the lchannel library through a linear decomposition of multiparty session types. The implementation uses the Akka actor framework and supports all transport abstractions provided by Akka. Neykova et al. [25] implemented in F# a session type provider to support on-demand compile-time protocol by generating protocol-specific .NET types from Scribble. Pabble [26] generates C/MPI skeleton code from parameterised Scribble for correct-by-construction role-parameterised MPI parallel programming but do not use types to check for conformance. The approach was revisited in [4] using a new distributed formalisation of parameterised Scribble and applied to Go in the Scribble-Go toolchain, with support for TCP and in-process shared memory transport. StMungo [17, 18] uses Scribble to generate typestate definitions for static type checking of communication protocols in Java.

This is the first work that applies the session-based API generation approach to WebSocket transport, and targets JavaScript applications for the web.

Session types in functional languages with linearity There are many approaches to embed session types natively in advanced type systems found in functional programming languages. The most challenging aspects of the embedding to support full session type verification is ensuring linear usage of channel resources. A more comprehensive survey of session types with linearity in functional languages can be found in [27], we highlight a few works that are closely related to our approach. The Links web programming language [21, 20] is a functional language designed for tierless web programming, and recently adds a support for *binary* session types in the style of GV [19], and has an extension to support linear types. Adding support for multiparty session types would require to extend the core calculus of the language. Padovani's FuSe [28, 22] infers and type checks usage of binary session-based communication

in OCaml, and uses a hybrid static/dynamic linearity check similar to the hybrid verification approach in [11]. Session-ocaml [13] implements session types in OCaml with lenses overcoming a linearity issue but only treats binary session types.

To the best of our knowledge, this work is the first which implements (i.e. generates code from) *multiparty session types* with *fully static* linearity guarantees.

6 Conclusion and Future work

We have presented a type-safe web application development work flow following the MPST framework, by encoding Endpoint FSMs as type classes and generating PureScript code from the Endpoint FSMs.

Future work include applying the approach to Haskell, although we suspect the ergonomics would be slightly worse without native row types support. The code generation process can be automated by a typechecker plugin [7], and also provide better error message by directly traversing the FSM during type checking. While WebSockets are bidirectional, a connection can only be opened by the browser, which prevents use in peer-to-peer browser communication. WebRTC could be explored as a solution for this use case. In our work we treat user interaction as a source of input separate to the session, however by describing interaction as a session, treating widgets as roles, we may benefit from properties that well typed protocols provide (i.e. progress). We believe this is an interesting design space where eventually the need for (a subset of) UI testing could be replaced by type checking.

Acknowledgements

We thank the anonymous reviewers for their feedback. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, and EP/N028201/1.

References

[1] Robert Atkey (2009): *Parameterised Notions of Computation*. *J. Funct. Program.* 19(3-4), pp. 335–376, doi:10.1017/S095679680900728X.

[2] The scribble authors (2008): *Scribble homepage*. `https://www.scribble.com`.

[3] Vincent Balat, Pierre Chambart & Grégoire Henry (2012): *Client-server Web applications with Ocsigen*. In: *WWW'12 dev track: Proceedings of the 21nd international conference on World Wide Web*, Lyon, France, p. 59. Available at `http://hal.archives-ouvertes.fr/hal-00691710`.

[4] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng & Nobuko Yoshida (2019): *Distributed Programming Using Role Parametric Session Types in Go*. In: *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, ACM, pp. 1–30, doi:10.1145/3290342.

[5] Roy Thomas Fielding (2000): *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.

[6] Google (2011): *Dart programming language*. `https://www.dartlang.org/`.

[7] Adam Gundry (2015): *A Typechecker Plugin for Units of Measure: Domain-specific Constraint Solving in GHC Haskell*. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, ACM, New York, NY, USA, pp. 11–22, doi:10.1145/2804302.2804305.

[8] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones & Philip Wadler (1996): *Type Classes in Haskell*. *ACM TOPLAS* 18(2), pp. 109–138, doi:10.1145/227699.227700.

[9] Hamish Mackenzie, Victor Nazarov, Luite Stegeman (2010): *GHCJS*. `https://github.com/ghcjs/ghcjs`.

[10] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. JACM 63, pp. 1–67, doi:10.1145/2827695.

[11] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification through Endpoint API Generation*. In: 19th International Conference on Fundamental Approaches to Software Engineering, LNCS 9633, Springer, pp. 401–418, doi:10.1007/978-3-662-49665-7_24.

[12] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*. In: 20th International Conference on Fundamental Approaches to Software Engineering, LNCS 10202, Springer, pp. 116–133, doi:10.1007/978-3-662-54494-5_7.

[13] Keigo Imai, Nobuko Yoshida & Shoji Yuen (2017): *Session-ocaml: a session-based library with polarities and lenses*. In: 19th International Conference on Coordination Models and Languages, LNCS 10319, Springer, pp. 99–118, doi:10.1007/978-3-319-59746-1_6.

[14] Jonathan King (2019): *Scribble Battleships*. `https://github.com/jonathanlking/scribble-battleships`.

[15] Mark P. Jones (2000): *Type Classes with Functional Dependencies*. In: ESOP, Springer, pp. 230–244, doi:10.1007/3-540-46425-5_15.

[16] Justin Uberti and Peter Thatcher (2011): *WebRTC*. `https://webrtc.org`.

[17] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking Protocols with Mungo and StMungo*. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, PPDP '16, ACM, New York, NY, USA, pp. 146–159, doi:10.1145/2967973.2968595.

[18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2018): *Typechecking protocols with Mungo and StMungo: A session type toolchain for Java*. Science of Computer Programming 155, pp. 52 – 75, doi:10.1016/j.scico.2017.10.006. Available at `http://www.sciencedirect.com/science/article/pii/S0167642317302186`.

[19] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In Jan Vitek, editor: Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[20] Sam Lindley & J. Garrett Morris (2017): *Lightweight functional session types*. Behavioural Types: from Theory to Tools, pp. 265–286, doi:10.13052/rp-9788793519817.

[21] Links contributors (2006): *The Links Programming Language*. `http://links-lang.org`.

[22] Hernán Melgratti & Luca Padovani (2017): *An OCaml Implementation of Binary Sessions*. Behavioural Types: from Theory to Tools, pp. 265–286, doi:10.13052/rp-9788793519817.

[23] Alexey Melnikov & Ian Fette (2011): *The WebSocket Protocol*. RFC 6455, doi:10.17487/RFC6455. Available at `https://rfc-editor.org/rfc/rfc6455.txt`.

[24] Microsoft Corporation (2014): *TypeScript Language Specification*. `http://typescriptlang.org/`.

[25] Rumyana Neykova, Raymond Hu, Nobuko Yoshida & Fahd Abdeljallal (2018): *A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#*. In: 27th International Conference on Compiler Construction, ACM, pp. 128–138, doi:10.1145/3178372.3179495.

[26] Nicholas Ng, Jose G.F. Coutinho & Nobuko Yoshida (2015): *Protocols by Default: Safe MPI Code Generation based on Session Types*. In: 24th International Conference on Compiler Construction, LNCS 9031, Springer, pp. 212–232, doi:10.1007/978-3-662-46663-6_11.

[27] Dominic Orchard & Nobuko Yoshida (2017): *Session Types with Linearity in Haskell*. Behavioural Types: from Theory to Tools, pp. 219–242, doi:10.13052/rp-9788793519817.

[28] Luca Padovani (2017): *A Simple Library Implementation of Binary Sessions*. Journal of Functional Programming 27, doi:10.1017/S0956796816000289.

[29] PureScript contributors (2017): *PureScript Language Specification*. `http://www.purescript.org/`.

[30] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. In: *31st European Conference on Object-Oriented Programming*, *LIPIcs* 74, Schloss Dagstuhl, pp. 24:1–24:31, doi:10.4230/LIPIcs.ECOOP.2017.24.

[31] Jérôme Vouillon & Vincent Balat (2013): *From Bytecode to JavaScript: the Js_of_ocaml Compiler*. *Software: Practice and Experience*, doi:10.1002/spe.2187.

[32] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2013): *The Scribble Protocol Language*. In Martín Abadi & Alberto Lluch-Lafuente, editors: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, *LNCS* 8358, Springer, pp. 22–41, doi:10.1007/978-3-319-05119-2_3.