MEng Individual Project

Imperial College London

Department of Computing

Type-safe Webservices Generation: Interim Report

Author:
Anson Miu

Supervisor:
Prof. Nobuko Yoshida

Second Marker:
TBC

January 22, 2020

Contents

Chapter 1

Introduction

Modern interactive web applications aim to provide a highly responsive user experience by minimising the communication latency between clients and servers. Whilst the HTTP request-response model is sufficient for retrieving static assets, applying the same stateless communication approach for interactive use cases (such as a multiplayer game) introduces undesirable performance overhead from having to frequently set up new connections for client-server interactions. Developers have since adopted other communication transport abstractions over HTTP connections such as the WebSockets protocol [4] to enjoy low-latency full-duplex client-server communication in their applications over a single persistent connection. Enabling more complex communication patterns caters for more interactive use cases (such as real-time multiplayer games), but introduces additional concerns to the developer with respect to implementation correctness.

Consider a classic turn-based board game of *Noughts and Crosses* between two players. Both players are identified by either noughts or crosses respectively, and take turns to place a mark on an unoccupied cell of a 3-by-3 grid until one player wins (when their markers form one straight line on the board) or a stalemate is reached (when all cells are occupied). A web-based implementation may involve players connected to a game server via WebSocket connections and interacting with the game from their web browser, which serve a *single-page application* (SPA) of the game client written in a popular framework like *React.js* [3]. SPAs feature a single HTML page and dynamically renders content via JavaScript in the browser. Players take turns to make a move on the game board and the server implements the game logic to progress the game forward until a result (either a win/loss or draw) can be declared.

Whilst WebSockets make this web-based implementation possible, it introduces the developer to a new family of communication errors, even for this simple game. In addition to the usual testing for game logic correctness, the developer needs to test against *deadlocks* (e.g. both players waiting for each other to make a move at the same time) and *communication mismatches* (e.g. player 1 sending a boolean to the game server instead of the board coordinates). The complexity of these errors, which correlate to the complexity of tests required against these errors, scale with the complexity of the communication patterns involved.

Multiparty Session Types (MPST) [16] provide a framework for formally specifying a structured communication pattern between concurrent processes and verifying implementations for correctness with respect to the communications aspect. By specifying the client-server interactions of our game as a protocol and verifying the implementations against the protocol for well-formedness, MPST theory guarantees well-formed implementations to be free from communication errors.

We see the application of the MPST methodology to verifying interactive web applications written in TypeScript to be an interesting design space - to what extent can the MPST methodology be applied such that we leverage TypeScript language features to maximise static safety guarantees whilst preserving a flexible, natural and idiomatic workflow for TypeScript developers? Such a workflow would ultimately decrease the overhead for incorporating MPST into mainstream web development, which reduces development time by programmatically verifying implementations for communication correctness.

We present a workflow for developing type-safe interactive SPAs motivated by the MPST framework: **(1)** An endpoint API code generation workflow targeting TypeScript-based web applications for multiparty sessions; **(2)** An encoding of session types in server-side TypeScript that enforces static linearity; and **(3)** An encoding of session types in browser-side TypeScript using the React framework that guarantees affine usage of communication channels.

Chapter 2

Background

2.1 Session Types

Web applications are one of many examples of distributed systems in practice. Distributed systems are built upon the interaction between concurrent processes, which can be implemented using the two main communication abstractions in *shared memory* and *message passing*.

Shared memory provides processes with the impression of a logical single large monolithic memory but requires programmers to understand consistency models in order to correctly reason about the consistency of shared state.

Message passing interprets the interaction between processes as the exchange of messages, and best describes the communication transports found in web applications, ranging from the stateless request-response client-server interactions via HTTP to full-duplex communication channels via the WebSocket protocol [4].

The process algebra π-calculus introduced by Milner in [11] provides a formalism of the message passing abstraction in terms of the basic building blocks of sending and receiving processes, along with inductively defined continuation processes. The composition of these primitives allow us to describe more complex communication sessions. Session types define the typing discipline for the π-calculus and provide reliability guarantees for communication sessions; the latter addresses a key challenge when reasoning about the correctness of distributed systems.

Many studies are done on the practical applications of session types, from developing languages providing native session type support [13] to implementing session types in existing programming languages across different paradigms. Implementations of the latter approach differ by how they leverage the design philosophy and features provided by the programming language. For example, King et al. leveraged the expressive type system of PureScript to perform static session type checking in [10], whilst Neykova and Yoshida introduced dynamic approaches to check the conformance of Python programs with respect to session types in [12].

2.1.1 Asynchronous π-calculus

The π-calculus models concurrent computation, where processes can execute in parallel and communicate via shared names. We first consider the asynchronous π-calculus introduced by Honda and Tokoro in [6]. Among the many flavours of the calculus which vary depending on the application domain, we outline the variant as presented in [14].

Figure 1 defines the syntax of processes in asynchronous π-calculus; the asynchrony comes from the lack of continuation in the output process.

- $\mathbf{0}$ is the nil process and represents inactivity.
- $\bar{u}\langle v\rangle$ is the output process that will send value v on u.
- $u(x).P$ is the input process that, upon receiving a message on u, will bind the message to x and carry on executing P under this binding.
- $P \mid Q$ represents the parallel composition of processes executing simultaneously.
- $!P$ represents the parallel composition of infinite instances of P; more specifically, $!P \equiv P \mid !P$.
- $(\nu a)\ P$ represents a name restriction where any occurrence of a in P is local and will not interfere with other names outside the scope of P.

$$
\begin{array}{rlr}
P, Q ::= & & \text{Processes} \\
& \mathbf{0} & \text{Nil Process} \\
\mid & \bar{u}\langle v\rangle & \text{Output} \\
\mid & u(x).P & \text{Input} \\
\mid & P \mid Q & \text{Parallel Composition} \\
\mid & !P & \text{Replication} \\
\mid & (\nu a)\ P & \text{Restriction} \\
u, v ::= & & \text{Identifiers} \\
& a, b, c, \ldots & \text{Names} \\
\mid & x, y, z, \ldots & \text{Variables}
\end{array}
$$

Figure 1: Syntax of Asynchronous π-calculus

The operational semantics model the interaction between parallel processes. Whilst [14] presents the full operational semantics, we highlight the [Comm] reduction rule which specifically models message passing: if the parallel composition of an input process and output process share the same name, the composition reduces to the continuation of the input process, substituting the variable x with the message received v. We omit the definitions of substitution, free variables and free names, α-equivalence and structural congruence; the interested reader may refer to [14].

$$\frac{}{\bar{a}\langle v\rangle \mid a(x).P \;\longrightarrow\; P[v/x]}\ [\textsc{Comm}]$$

We additionally define a process P to be *stuck* if P is not the nil process and P cannot be reduced any further. For example, the process $P = a(x).\mathbf{0} \mid \bar{b}\langle v\rangle$ is stuck as the parallel composition of an input process and an output process that do not share the same name cannot be reduced using [Comm]. In practice, a stuck process contains communications that will never be executed.

2.1.2 Binary Session Types

A *binary session* is a parallel composition of two processes, each representing a distinct participant. In the context of web applications, a binary session would describe the interactions between client and server. Without loss of generality, a *session* represents the sequence of send and receive actions of a single participant.

We introduce a *synchronous* session calculus inspired by [16]. Figure 2 defines the syntax formally; we briefly discuss the main components and how it differs from the variant introduced in [14]:

- **Synchronous communication**: Processes that send a message will have a continuation which will be executed upon a successful send.

- **Polyadic communication**: A vector of values can be communicated at once.

- **Branching and selection**: A branching process can offer a set of branches, each defined by its own label identifier and continuation process. A selection process can select a branch by sending the corresponding label identifier alongside the payload to the branching process.

- **Labelled messages**: a label identifier is attached to all messages; the input process in §2.1.1 is generalised as a branching process that offers one branch.

The [Comm] rule in the operational semantics for this calculus exemplifies these new additions: given a binary session between distinct participants $\mathbf{p}$ and $\mathbf{q}$ where $\mathbf{q}$ offers a set of labelled branches, if $\mathbf{p}$ selects a label offered by $\mathbf{q}$ and sends a vector of expressions $e_1,\ldots,e_n$ that evaluate[1] to the corresponding vector of values $v_1,\ldots,v_n$, the session reduces to a session with the continuation from the selection process composed with the continuation from the selected branch of the branching process,

[1] We adopt the operational semantics for expression evaluation $e \downarrow v$ as defined in [14].

the latter having the variables $x_1, \dots, x_n$ substituted with the vector of values $v_1, \dots, v_n$ received.

$$\frac{\exists j \in I. l_j = l \quad e_1 \downarrow v_1 \quad \dots \quad e_n \downarrow v_n \qquad \mathbf{p} \neq \mathbf{q}}{\mathbf{p} :: \mathbf{q} \lhd l\langle e_1 \dots e_n \rangle.\ P \mid \mathbf{q} :: \mathbf{p} \rhd \{l_i(x_1 \dots x_n) : Q_i\}_{i \in I} \longrightarrow \mathbf{p} :: P \mid \mathbf{q} :: Q_j[v_k/x_k]_{k=1}^{n}} \ [\textsc{Comm}]$$

$v ::=$		$\underline{n} \mid \texttt{true} \mid \texttt{false}$	Values
$e, e' ::=$			Expressions
		v	Values
	$\mid$	x	Variables
	$\mid$	$e + e' \mid e - e'$	Arithmetic Operators
	$\mid$	$e = e' \mid e < e' \mid e > e'$	Relational Operators
	$\mid$	$e \wedge e' \mid e \vee e' \mid \neg e$	Logical Operators
	$\mid$	$e \oplus e'$	Non-Determinism
$\mathbf{p} ::=$		$\mathbf{Client}, \mathbf{Server}$	Participant
$P, Q ::=$			Processes
		$\mathbf{0}$	Nil Process
	$\mid$	$\mathbf{p} \rhd \{l_i(x_1 \dots x_n) : P_i\}_{i \in I}$	Branching
	$\mid$	$\mathbf{p} \lhd l\langle e_1 \dots e_n \rangle.\ P$	Selection
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional
	$\mid$	$\mu X.\ P$	Recursive Process
	$\mid$	X	Process Variable
$l, l' ::=$		"$\texttt{str}$"	Label Identifiers
$\mathcal{M} ::=$		$\mathbf{p} :: P \mid \mathbf{q} :: Q$	Binary Session

Figure 2: Syntax of Session Calculus with Branching, Selection and Recursion

Additionally, the calculus introduces:

- **Conditionals**: If $e \downarrow \texttt{true}$, the process $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ reduces to P; if $e \downarrow \texttt{false}$, the process $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ reduces to Q.

- **Recursion**: Following the equirecursive approach, the occurrence of the process variable X in the recursive process can be expanded into the process transparently; more specifically, $\mu X.P \equiv P[(\mu X.P)/X]$.

Session types represent the type theory for our session calculus. We define the syntax of session types for binary sessions in figure 3.

$$
\begin{array}{rlr}
U ::= & \texttt{int} \mid \texttt{bool} & \text{Sorts} \\
S ::= & & \text{Session Types} \\
 & \mathbf{end} & \text{Termination} \\
\mid & \mathbf{p}\&\{l_i(U_1 \dots U_n) : S_i\}_{i \in I} & \text{Branching} \\
\mid & \mathbf{p} \oplus \{l_i(U_1 \dots U_n) : S_i\}_{i \in I} & \text{Selection} \\
\mid & \mu\mathbf{t}.\ S & \text{Recursive Type} \\
\mid & \mathbf{t} & \text{Type Variable}
\end{array}
$$

Figure 3: Syntax of Session Types

We derive the type of a process with a *typing judgement* of the form $\Gamma \vdash P : S$, which reads, *under the typing context* Γ*, process* P *has session type* S.

The *typing context* records typing assumptions used as part of the derivation: in the case of binary session types, the context maps expressions to sorts, and process variables to session types. A typing judgement is constructed in terms of inference rules defined inductively on the structure of processes and expressions.

We present the rules for [Ty-Sel] and [Ty-Bra], the remaining rules follow from [14] and can be trivially defined as they leverage the syntactic similarities between session types and our session calculus.

$$
\frac{\forall i \in I. \qquad \Gamma, x_1 : U_1, \dots, x_n : U_n \vdash P_i : S_i}{\Gamma \vdash \mathbf{p} \triangleright \{l_i(x_1 \dots x_n) : P_i\}_{i \in I} : \mathbf{p}\&\{l_i(U_1 \dots U_n) : S_i\}_{i \in I}} \ [\textsc{Ty-Bra}]
$$

$$
\frac{\Gamma \vdash e_1 : U_1 \quad \dots \quad \Gamma \vdash e_n : U_n \quad \Gamma \vdash P : S}{\Gamma \vdash \mathbf{p} \triangleleft l\langle e_1 \dots e_n \rangle.P : \mathbf{p} \oplus \{l(U_1 \dots U_n) : S\}} \ [\textsc{Ty-Sel}]
$$

The definition of stuck processes from §2.1.1 motivate the discussion of communication errors that may occur during interactions among participants. We outline two of the main classes of errors:

- **Deadlock**: Progress cannot be made when the two participants expect to be receiving a message from each other at the same time.

- **Communication mismatch**: Progress cannot be made when the selection process sends a message with a label identifier not offered by the branching process; likewise, the payload sent must be compatible with the sort expected by the branching process for the selected branch.

Session types ensure that well-typed binary sessions are guaranteed to be free from these communication errors through the concept of *duality*. Duality defines a notion of *compatibility* between processes: two session types are dual with respect to each other if the communication between them (i.e. pairs of sending and receiving actions) always match (i.e. with respect to the selected label and message payload type). We define $\overline{S}$ as the dual type of S in Table 1.

$$\begin{aligned}
\overline{\mathbf{end}} &= \mathbf{end}\\
\overline{\mathbf{p}\&\{l_i(U_1\dots U_n):S_i\}_{i\in I}} &= \mathbf{q}\oplus\left\{l_i(U_1\dots U_n):\overline{S_i}\right\}_{i\in I}\\
\overline{\mathbf{p}\oplus\{l_i(U_1\dots U_n):S_i\}_{i\in I}} &= \mathbf{q}\&\left\{l_i(U_1\dots U_n):\overline{S_i}\right\}_{i\in I}\\
\overline{\mu\mathbf{t}.S} &= \mu t.\overline{S}\\
\overline{\mathbf{t}} &= \mathbf{t}
\end{aligned}$$

Table 1: Duality of Binary Session Types involving participants **p** and **q**

Consequently, a binary session is well-typed if the participating processes are typed to be dual with respect to each other: we illustrate this in [MTy].

$$\frac{\cdot\vdash P:S \qquad \cdot\vdash Q:\overline{S}}{\vdash \mathbf{p}::P \mid \mathbf{q}::Q}\ [\textsc{MTy}]$$

The definition of duality alone restricts the definition of well-typed binary sessions to those where the two processes are derived to be *exactly* dual types of one another. Consider the pair of session types below:

$$\begin{aligned}
S_{\text{Client}} &= \mathbf{Server}\oplus\{\text{Succ}(\texttt{int}):\mathbf{Server}\&\{\text{Res}(\texttt{int}):\mathbf{end}\}\}\\
S_{\text{Server}} &= \mathbf{Client}\&\{\text{Succ}(\texttt{int}):\mathbf{Client}\oplus\{\text{Res}(\texttt{int}):\mathbf{end}\},\ \text{Quit}():\mathbf{end}\}
\end{aligned}$$

Whilst $\overline{S_{\text{Client}}} \neq S_{\text{Server}}$, this pair of session types is intuitively compatible as the client is selecting a branch offered by the server, where the session types for the continuations of this branch for both participants are indeed dual.

This motivates the concept of *subtypes*, which allows a process to be typed by its "supertype" when required. $\leqslant$[2] defines the subtyping relation: $S \leqslant S'$ reads S *is a subtype of* S', and is defined coinductively on the structure of session types.

We present the inference rules for [Sub-Bra] and [Sub-Sel] inspired by [16] but adapted for polyadic communication; the intuition behind subtyping and subsorting is outlined below:

[2] The $\leqslant$ operator is also an overloaded relation on sorts to express subsorting.

- **Branching**: The supertype of a branching process offers a subset of the branches and expects more specific types of payload; intuitively, if a process expects to receive an `int`, it can handle a `nat` payload.
- **Selection**: The supertype of a selection process offers a superset of the internal choices and can send more generic types of payload; intuitively, if a process sends a `nat`, the payload is compatible with receivers expecting a more generic `int` payload.

$$\frac{\forall i \in I. \qquad U_1' \leqslant U_1 \ldots U_n' \leqslant U_n \qquad S_i \leqslant S_i'}{\mathbf{p} \& \{l_i(U_1 \ldots U_n) : S_i\}_{i \in I \cup J} \leqslant \mathbf{p} \& \left\{l_i(U_1' \ldots U_n') : S_i'\right\}_{i \in I}} \; [\textsc{Sub-Bra}]$$

$$\frac{\forall i \in I. \qquad U_1 \leqslant U_1' \ldots U_n \leqslant U_n' \qquad S_i \leqslant S_i'}{\mathbf{p} \oplus \{l_i(U_1 \ldots U_n) : S_i\}_{i \in I} \leqslant \mathbf{p} \oplus \left\{l_i(U_1' \ldots U_n') : S_i'\right\}_{i \in I \cup J}} \; [\textsc{Sub-Sel}]$$

We also introduce subsumption in [Ty-Sub] to incorporate the subtyping relation into the typing judgement.

$$\frac{\Gamma \vdash P : S \quad S \leqslant S'}{\Gamma \vdash P : S'} \; [\textsc{Ty-Sub}]$$

This allows us to construct a derivation to show that the binary session

$$\mathcal{M} = \textbf{Client} :: P_{\text{Client}} \mid \textbf{Server} :: P_{\text{Server}}$$

is well-typed, assuming P_{Client} and P_{Server} are typed S_{Client} and S_{Server} respectively.

$$\frac{\dfrac{\vdots}{\cdot \vdash P_{\text{Client}} : S_{\text{Client}}} \qquad \dfrac{\dfrac{\vdots}{\cdot \vdash P_{\text{Server}} : S_{\text{Server}}} \qquad \dfrac{\vdots}{S_{\text{Server}} \leqslant S_{\text{Client}}}}{\cdot \vdash P_{\text{Server}} : S_{\text{Client}}} \; [\textsc{Ty-Sub}]}{\vdash \textbf{Client} :: P_{\text{Client}} \mid \textbf{Server} :: P_{\text{Server}}} \; [\textsc{MTy}]$$

2.1.3 Multiparty Session Types

Whilst binary session types provide communication guarantees between exactly 2 participants, distributed systems generally involve more parties in practice. This is equally relevant in interactive web applications, as motivated by the *Battleships* game example in [10] where the server coordinates interactions between two players.

Whilst there is a natural syntactical extension to our session calculus for describing multiparty sessions[3] as

$$\mathcal{M} ::= \quad \mathbf{p_1} :: P_1 \mid \mathbf{p_2} :: P_2 \mid \ldots \mid \mathbf{p_n} :: P_n$$

[3] We also adopt the shorthand $\mathcal{M} ::= \prod_{i=1}^{n} \mathbf{p_i} :: P_i$ as used in the literature.

the same cannot be said for the binary session typing discipline, particularly with respect to duality. The same notion of duality does not extend to the decomposition of multiparty interactions into multiple binary sessions: [14] and [16] both present counterexamples of well-typed binary sessions that, when composed to represent a multiparty session, results in communication errors thus violating guarantees of well-typed sessions.

Honda et al. presents *multiparty session types* in [7] to extend the binary session typing discipline for sessions involving more than 2 participants, whilst redefining the notion of compatibility in this multiparty context. Multiparty session types are defined in terms a *global type*, which provides a bird's eye view of the communication protocol describing the interactions between pairs of participants. Figure 4 defines the syntax of global types inspired by [16] and adapted to be compatible with our session calculus presented in §2.1.2.

$$\begin{array}{lll}
G ::= & & \text{Global Types} \\
& \mathbf{end} & \text{Termination} \\
| & \mathbf{p} \to \mathbf{q} : \{l_i(U_1 \ldots U_n).\ G_i\}_{i \in I} & \text{Message Exchange} \\
| & \mu\mathbf{t}.\ G & \text{Recursive Type} \\
| & \mathbf{t} & \text{Type Variable}
\end{array}$$

Figure 4: Syntax of Global Types

To check the conformance of a participant's process against the protocol specification described by the global type, we *project* the global type onto the participant to obtain a session type that only preserves the interactions described by the global type that pertain to the participant. Projection is defined by the $\upharpoonright$ operator, more commonly seen in literature in its infix form as $G \upharpoonright \mathbf{p}$ describing the projection of global type G for participant $\mathbf{p}$. Intuitively, the projected local type of a participant describes the protocol from the viewpoint of the participant.

More formally, projection can be interpreted as a *partial function* $\upharpoonright :: G \times \mathbf{p} \rightharpoonup S$, as the projection for a participant may be undefined for an ill-formed global type; [14] presents examples of where this is the case, and [16] presents the formal definition of projection.

The notion of compatibility in multiparty session types is still captured by [MTy], but adapted to consider the local projections for all participants as supposed to dual types in the binary case.

$$\frac{\forall i \in I.\quad \cdot \vdash P_i : G \upharpoonright \mathbf{p_i} \qquad \mathrm{pt}(G) \subseteq \{\mathbf{p_i} \mid i \in I\}}{\vdash \prod_{i \in I} \mathbf{p_i} :: P_i \ : G}\ [\textsc{MTy}]$$

For a multiparty session $\mathcal{M} = \prod_{i \in I} \mathbf{p_i} :: P_i$ to be well-typed by a global type G, we require:

1. All participant processes $\mathbf{p_i} :: P_i$ to be well-typed with respect to their corresponding well-defined projection $G \upharpoonright \mathbf{p_i}$, and
2. G does not describe interactions with participants not defined in $\mathcal{M}$

Well-typed multiparty session enjoys the following communication guarantees as outlined in [2]:

- **Communication safety**: The types of sent and expected messages will always match.
- **Protocol fidelity**: The exchange of messages between processes will abide by the global protocol.
- **Progress**: Messages sent by a process will be eventually received, and a process waiting for a message will eventually receive one; this also means there will not be any sent but unreceived messages.

This motivates an elegant, decentralised solution for checking protocol conformance in practice: once the global type for the protocol is defined, local processes can verify their implementation against their corresponding projection in isolation, independent of each other.

2.2 The Scribble Protocol Language

Whilst session type theory represents the type language for concurrent processes, it also forms the theoretical basis of proposals introduced to implement session types for real-world application development: the Scribble language is one such proposal.

Scribble [15] is a platform-independent description language for the specification of message-passing protocols. The language describes the behaviour of communicating processes at a high level of abstraction: more importantly, the description is independent from implementation details in the same way that the type signature of a function declaration is decoupled from the corresponding function definition.

A Scribble protocol specification describes an agreement of how participating systems, referred to as *roles*, interact. The protocol stipulates the sequence of structured messages exchanged between roles; each message is labelled with a name and the type of payload carried by the message.

We present an example of a Scribble protocol in Figure 5 adapted from [8]. The protocol specifies an arithmetic web service offered by a server to a client, represented by roles S and C respectively. The client is permitted to either:

- Send two `int`s attached to an `Add` message, where the server will respond with an `int` in a message labelled `Res`, and the protocol recurses; or,

- Send a `Quit` message, where the server will respond with a `Terminate` message and the protocol ends.

The platform-independent nature of Scribble can be observed from the `type` declaration on Line 1: the developer has the freedom to specify message payload formats and data types from the target language of the implementation - in this case, aliasing the built-in Java integer as `int` throughout the protocol.

```
 1  type <java> "java.lang.Integer" from "rt.jar" as int;
 2
 3  global protocol Adder(role C, role S) {
 4      choice at C {
 5          Add(int, int)   from C to S;
 6          Res(int)        from S to C;
 7          do Adder(C, S);
 8      } or {
 9          Quit()       from C to S;
10          Terminate() from S to C;
11      }
12  }
```

Figure 5: Adder Protocol in Scribble

The simplicity of the protocol specification language reflects the design goals for Scribble, as outlined in [15], to be easy to read, write and maintain, even for developers who are not accustomed to the formalities in protocol specification. Moreover, we clearly observe the parallels between the Scribble language and multiparty session type (MPST) theory, from the homomorphic mapping between Scribble roles and MPST participants to the syntactic similarities between the specification in Figure 5 and the global type below written in the calculus.

$$
\begin{aligned}
G = \mu \mathbf{t}.\mathbf{C} \to \mathbf{S} : \{ \quad & \mathrm{Add}(\mathtt{int}, \mathtt{int}) : \mathbf{S} \to \mathbf{C} : \{\mathrm{Res}(\mathtt{int}) : \mathbf{t}\}, \\
& \mathrm{Quit}() : \mathbf{S} \to \mathbf{C} : \{\mathrm{Terminate}() : \mathbf{end}\} \\
\}&
\end{aligned}
$$

2.2.1 Endpoint Finite State Machines

The protocol specification language is a component of the broader Scribble project in [15]; through the project, Honda et al. also aims to facilitate the development of endpoint applications that conform to user-specified protocols.

A Scribble global protocol can be projected to a role to obtain the local specification of said protocol from the role's viewpoint. This is analogous to the standard algorithmic projections in multiparty session type theory: the projected local specification, or local protocol, only preserves the interactions in which the target role is involved. Any user-defined type declarations in the global protocol will also be preserved. This allows local roles, also referred to as *endpoints*, to verify their implementation against their local protocol for conformance, independent of other endpoints. The communication safety guarantees from MPST theory also apply here:

if the implementation for each endpoint is verified against its local protocol, the multiparty system as a whole will conform to the global protocol.

To facilitate the verification process, a local protocol is converted into an *endpoint finite state machine* (EFSM). An EFSM encodes the control flow of the local protocol, where an initial state is defined and each transition from some state to a successor state corresponds to a valid IO action (i.e. sending to or receiving from another role) permitted at the endpoint at that state. Figure 6 presents the EFSM formalism adopted from [8].

$$
\begin{array}{rlr}
\text{EFSM} ::= & \mathbb{R} \times \mathbb{L} \times \mathbb{T} \times \Sigma \times \mathbb{S} \times \delta & \text{Endpoint FSM} \\
\mathbb{R} ::= & r,\ r',\ \ldots & \text{Role Identifiers} \\
\mathbb{L} ::= & l,\ l',\ \ldots & \text{Message Label Identifiers} \\
\mathbb{T} ::= & \texttt{int},\ \texttt{bool},\ T,\ T',\ \ldots & \text{Payload Format Types} \\
\Sigma ::= & & \text{Actions} \\
 & r!l(\tilde{T}) \quad \text{where } \tilde{T} \subseteq \mathbb{T} & \text{Output} \\
\mid & r?l(\tilde{T}) \quad \text{where } \tilde{T} \subseteq \mathbb{T} & \text{Input} \\
\mathbb{S} ::= & S,\ S', \ldots & \text{State Identifiers} \\
\delta ::= & \mathbb{S} \times \Sigma \rightharpoonup \mathbb{S} & \text{State Transition Function}
\end{array}
$$

Figure 6: Syntax for Endpoint FSM

We highlight the semantics of the *state transition function*. Its domain defines the set of permitted actions for each state, and the range defines the successor state. Crucially, this is a *partial function* because the EFSM restricts what can be done (in terms of send and receive actions, and what label/payload types can be sent and received).

[8] also identifies properties of EFSMs for well-formed protocol specifications: **(1)** there is exactly one initial state, **(2)** there is at most one terminal state, and **(3)** every non-terminal state is either an input state or output state. There are further restrictions on input and output states, namely these states must only send to or receive from the same role. We formalise these properties in Figure 7.

2.3 Code Generation

The EFSM describes the local session type and provides guidance to developers for verifying that their endpoint implementation conforms to the communication protocol. However, a direct encoding of the local session type into the target language of

$$
\begin{aligned}
\texttt{initial}_{EFSM}(S) &\iff \nexists S' \in \mathbb{S}, \alpha \in \Sigma.\ \delta(S', \alpha) = S \\
\texttt{terminal}_{EFSM}(S) &\iff \delta(S) = \emptyset \\
\texttt{output}_{EFSM}(S) &\iff \delta(S) = \{\alpha \in \Sigma \mid \exists l \in \mathbb{L}, \tilde{T} \subseteq \mathbb{T}.\ \alpha = r!l(\tilde{T})\} \text{ for some } r \in \mathbb{R} \\
\texttt{input}_{EFSM}(S) &\iff \delta(S) = \{\alpha \in \Sigma \mid \exists l \in \mathbb{L}, \tilde{T} \subseteq \mathbb{T}.\ \alpha = r?l(\tilde{T})\} \text{ for some } r \in \mathbb{R}
\end{aligned}
$$

Figure 7: Types of EFSM States

the implementation is usually not feasible as the EFSM assumes IO objects as first-class citizens and communication channels are linear resources, features that are left to be desired in mainstream programming languages (such as Java and Python) used by the developers' implementations. We note that languages with native session type support [13] do exist, but their usage largely remains for research purposes as supposed to real-world application development.

Code generation is a common approach for verifying implementations written in the aforementioned mainstream languages against the EFSM. Approaches in the literature differ by how they leverage features in the target language (such as the type system), but generally define some interpretation of the EFSM in the target language and generate APIs which the developer can use to implement a target application that guarantees the following two properties:

- **Behavioural typing**: The execution trace of messages sent and received by the application is accepted by the EFSM.
- **Channel linearity**: Each transition in the EFSM represents a channel resource. When the application transitions from some state S to some successor state S', it must no longer be able to access a reference (e.g. have an alias) to S.

We outline the different existing approaches and summarise how they verify the aforementioned properties in §2.3.4.

2.3.1 Runtime Monitors

Neykova and Yoshida targeted the MPST methodology for Python programs in [12] and proposed to generate *runtime monitors* from the EFSM. These monitors expose APIs for sending and receiving messages, which is used by the developer in their implementation. The runtime monitor is an abstraction between the developer's implementation and the actual communication channel, and "executes" the EFSM internally to ensure protocol conformance. When the developer sends a message (with some label and payload) using the API, the runtime monitor checks whether this send action conforms to the current EFSM state, and if so, performs the send and advances to the successor state. Likewise, when the developer invokes a receive, the runtime monitor verifies that this is permitted at the current EFSM state before returning the received payload.

We observe that this approach complements the dynamic typing nature of the Python language, which makes it sensible to perform behavioural typing at runtime. As the send and receive IO primitives are made available to the developer, there are no "instances" of channel resources created, so the developer cannot explicitly hold a reference to some state in the EFSM (let alone keep aliases), so channel linearity is trivially guaranteed here.

2.3.2 Type-Level Encoding

King et al. presented an approach in [10] for integrating session types into web development using the PureScript language, which takes advantage of its expressive type system to provide static guarantees on implementation conformance with respect to the protocol. We outline the main components of their EFSM encoding:

Actions as type classes The semantics of the state transition function in the EFSM formalism express that a tuple of state-action *uniquely defines* a successor state. These semantics can be expressed by *multi-parameter type classes (MPTC) with functional dependencies*: `class Send r s t a | s -> t r a` defines `Send` as a MPTC parameterised by recipient `r`, current state `s`, successor state `t` and payload type `a`, and `s -> t r a` expresses the functional dependency that, for an instance of this type class, the current state uniquely determines the successor state, the recipient and payload type. These type classes are independent of the EFSM.

Transitions as instances of type classes By encoding states as data types, valid EFSM transitions are encoded as *instances* of the type classes. If `S1` is an output state, sending an `Int` to `Svr` with successor S2, we would encode `instance SendS1 :: Send Svr S1 S2 Int`. Because of the functional dependency, the developer cannot instantiate an invalid transition (e.g. `Send Svr S1 S3 Bool`, since `S1` uniquely determines the other type parameters.

This proposal is relevant to the problem we are tackling, and we appreciate that the intricacy of the library design in the communication combinators to conceal the channel resources is something we can build upon in our solution. We also observe the challenges of applying session types to the front-end environment, as shown by the careful choice made in [10] to use the *Concur UI* framework which builds UI elements sequentially to model sequential sessions; not doing so would require binding channel resources into event listeners on UI elements, which makes linearity violation possible (e.g. by binding a send transition to a button click event, the user might click the button twice, thus reusing the channel).

2.3.3 Hybrid Session Verification

Whilst runtime monitors and type-level encoding achieve communication safety guarantees via dynamic and static approaches respectively, Hu and Yoshida implemented a workflow in [8] that performs hybrid verification of communication protocol conformance for Java applications. We highlight the two main components below:

Static session typing States are represented as classes; supported transitions on each state are represented as instance methods, parameterised by the role, label and payload involved in the message exchange. A send method takes the payload to send as a parameter, whilst a receive method is a blocking call that requires the caller to allocate a `Buf<T>` wrapper on the stack (where `T` is the expected payload type), then the receive method populates the payload into the wrapper and returns upon receiving from the channel. These instance methods return a new instance of the successor state class.

Runtime linear channel usage checks Each state keeps track of its usage in a private boolean flag and throws an exception when the instance method is called twice, which signifies channel reuse and a linearity violation. Similarly, the `SessionEndpoint` class keeps track of whether the connection is open or close, and throws an exception when program execution exits the scope of the session endpoint and a terminal state has yet to be reached, signifying a protocol violation as the session hasn't completed yet but is now out of scope.

We find that this proposal strikes a good balance between maximising static communication safety guarantees whilst providing an intuitive set of APIs for developers to efficiently write their applications. For example, the encoding of transitions as instance methods that return new instances of the successor state exposes the channel resource, and since Java does not build support for linear resources and does not monitor aliasing of variables, linear channel usage is monitored dynamically. However, this complements the imperative style of application code written in Java, and takes advantage of the type system to statically enforce valid transitions in the code.

2.3.4 Comparison

We compare how these existing code generation approaches provide communication safety guarantees in Table 2.

Language	Behavioural typing	Channel linearity
Python [12]	Dynamically enforced - runtime monitor represent the EFSM and only execute supported transitions.	Trivially guaranteed - channel resources not exposed, developer uses `send()` and `receive()` primitives.
PureScript [10]	Type-level encoding - EFSM states are encoded as types, IO actions are encoded as multi-parameter type classes (to express the semantics of the state transition function) and EFSM transitions are encoded as instances of said type classes.	Statically guaranteed - channels are not exposed in the provided combinators to prevent reuse, and session constructor requires a `Session` continuation parameterised by initial and terminal state to prevent incomplete sessions.
Java [8]	Statically guaranteed - states are encoded as classes; transitions are encoded as instance methods on the state class and return a new instance of the successor state class.	Checked at runtime - states keep `used` boolean flag to detect and prevent reuse, Session API implements `AutoCloseable` interface to be used in resource try-catch block to prevent unused.

Table 2: Comparison between existing MPST code generation approaches.

Chapter 3

Project Plan

We aim to develop a multiparty session type-safe development workflow for building interactive full-stack TypeScript applications that conform to a communication protocol. This involves encoding session types using the TypeScript language and implementing a code generation workflow to generate the encodings.

3.1 Delivery

At the point of writing, we have analysed the existing code generation approaches presented in [8; 15; 12] and assessed their applicability with respect to verifying the communication aspects of interactive web applications. We have also explored approaches for encoding session types into TypeScript, motivated by existing works along with similar proposals specific to web development in [10; 5].

We plan to deliver our code generation implementation incrementally, aiming to get a working version of the end-to-end workflow first, then iteratively add support for more complex session type primitives. This minimises the risks involved in the project by ensuring we have a functional deliverable at the early stages of the project. We plan to deliver this basic end-to-end workflow by the end of February.

The subsequent months will involve adding support for other primitives (i.e. selection, choice and recursion) for multiparty sessions. We plan to measure our progress by writing example protocols for these milestones and checking that our implementation supports those protocols by the end of each month.

We will also try to connect with web developers in the community to experiment with our implementation to get feedback on ways to make it more applicable and compatible with industry practices, such that we can apply their feedback on the next iterations of our deliverables.

If time permits, we may explore possible extensions of applying supporting *Explicit Connection Actions* presented in [9] in our workflow to support more protocols.

3.2 Timetable

We attach a preliminary timetable (Table 3) for guidance.

Month	Milestones	Deadlines
Jan	• Explore approaches for TypeScript encoding • Investigate methods for code generation from EFSM	• 24th: Interim Report • 25th: PLACES 2020
Feb	• Develop API generation end-to-end toolchain • Support binary sessions with simple send/receive	• 14th: Project Review
Mar	• Support multiparty sessions with simple send/receive	• 16th-20th: Examinations
Apr	• Support binary sessions with selection, choice and recursion • Example: calculator service	• 6th-17th: Easter break
May	• Support multiparty sessions with selection, choice and recursion • Example: Tic Tac Toe game	• 15th: Health check-up
Jun	• Complete report write-up	• 15th: Final Report • 26th: Final Archive

Table 3: Project Timetable.

Chapter 4

Evaluation Plan

4.1 Correctness

We will qualitatively argue how our TypeScript encodings of session types achieve behavioural typing and preserves channel linearity.

If time permits, we will explore possible formalisms of our implementation as a calculus upon which to prove correctness. This may be related to the formal language presented in [1], but we will need to extend this to be specific to our TypeScript encodings of session types.

4.2 Productivity

By adopting tools and practices used in industry, we wish to assess the extent to which our implementation boosts developer productivity by providing APIs that guarantee communication protocol conformance in a way that is compatible with existing workflows. Comparing the development workflow of implementing the *Battleships* game using the PureScript workflow in [10] and our TypeScript-based proposal would yield interesting findings.

This can be carried out as a survey to be completed by web developers familiar with both languages. We would include questions that allow us to collect both qualitative and quantitative feedback to make effective comparisons between our approach and existing proposals.

4.3 Performance

Examples of performance metrics include: lines of code to be written by the developer (as a result of using the generated APIs), the size of the transpiled JavaScript assets to be loaded on the server and client browsers. We aim to extract these metrics from our approach and compare them against metrics extracted from a baseline implementation (i.e. without using the MPST methodology) and evaluate the performance overhead (if any) of our MPST-based approach.

Bibliography

[1] Bierman, G., Abadi, M., and Torgersen, M. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming* (2014), R. Jones, Ed., Lecture Notes in Computer Science, Springer, p. 257–281. pages 22

[2] Coppo, M., Dezani-Ciancaglini, M., Padovani, L., and Yoshida, N. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming* (2015), vol. 9104 of *LNCS*, Springer, pp. 146–178. pages 13

[3] Facebook Open Source. React – A JavaScript library for building user interfaces. pages 3

[4] Fette, I., and Melnikov, A. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. pages 3, 5

[5] Fowler, S. Model-view-update-communicate: Session types meet the elm architecture. *arXiv:1910.11108 [cs]* (Oct 2019). arXiv: 1910.11108. pages 20

[6] Honda, K., and Tokoro, M. An object calculus for asynchronous communication. In *ECOOP'91 European Conference on Object-Oriented Programming* (1991), P. America, Ed., Springer Berlin Heidelberg, p. 133–147. pages 6

[7] Honda, K., Yoshida, N., and Carbone, M. Multiparty Asynchronous Session Types. In *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2008), ACM, pp. 273–284. pages 12

[8] Hu, R., and Yoshida, N. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering* (2016), vol. 9633 of *LNCS*, Springer, pp. 401–418. pages 13, 15, 18, 19, 20

[9] Hu, R., and Yoshida, N. *Explicit Connection Actions in Multiparty Session Types*, vol. 10202. Springer Berlin Heidelberg, 2017, p. 116–133. pages 20

[10] King, J., Ng, N., and Yoshida, N. Multiparty session type-safe web development with static linearity. *Electronic Proceedings in Theoretical Computer Science 291* (Apr 2019), 35–46. pages 5, 11, 17, 19, 20, 22

[11] MILNER, R. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999. pages 5

[12] NEYKOVA, R., AND YOSHIDA, N. How to Verify Your Python Conversations. *Behavioural Types: from Theory to Tools* (2017), 77–98. pages 5, 16, 19, 20

[13] XI, H. Applied type system: An approach to practical programming with theorem-proving. *Journal of Functional Programming* (2016), 30. pages 5, 16

[14] YOSHIDA, N. Lecture Notes in CO406 Concurrent Processes, October 2019. pages 6, 7, 9, 12

[15] YOSHIDA, N., HU, R., NEYKOVA, R., AND NG, N. The Scribble Protocol Language. In *8th International Symposium on Trustworthy Global Computing* (2013), vol. 8358 of *LNCS*, Springer, pp. 22–41. pages 13, 14, 20

[16] YOSHIDA, N., AND LORENZO, G. A Very Gentle Introduction to Multiparty Session Types. pages 3, 7, 10, 12