Imperial College London

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

December 31, 2019

Contents

1 Introduction **3**

2 Background **4**
2.1 Session Types 4
2.1.1 Overview 4
2.1.2 Asynchronous π-calculus 5
2.1.3 Binary Session Types 6
2.1.4 Multiparty Session Types 10
2.2 The Scribble Protocol Language 12
2.2.1 Overview 12
2.2.2 Endpoint Finite State Machines 13
2.3 Code Generation 15
2.3.1 Overview 15
2.3.2 Static Session Typing 15
2.3.3 Runtime Monitors 15
2.3.4 Hybrid Session Verification 15
2.3.5 Comparison 16
2.4 TypeScript 17
2.4.1 Overview 17
2.4.2 Gradual Type System 17
2.4.3 Language Features 17

3 Project Plan **18**
3.1 Theory 18
3.2 Implementation 18
3.3 Timetable 18

4 Evaluation Plan **19**
4.1 Theory 19
4.2 Implementation 19

Bibliography **20**

Chapter 1

Introduction

Chapter 2

Background

2.1 Session Types

2.1.1 Overview

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 [2].

The process algebra π-calculus introduced by Milner in [7] 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 [9] 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 [6], whilst Neykova and Yoshida introduced dynamic approaches to check the conformance of Python programs with respect to session types in [8].

2.1.2 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 [3]. Among the many flavours of the calculus which vary depending on the application domain, we outline the variant as presented in [10].

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.

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

Figure 1: Syntax of Asynchronous π-calculus

The operational semantics model the interaction between parallel processes. Whilst [10] 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 [10].

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

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.

2.1.3 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 [12]. Figure 2 defines the syntax formally; we briefly discuss the main components and how it differs from the variant introduced in [10]:

- **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 §2.1.2 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 **p** and **q** where **q** offers a set of labelled branches, if **p** selects a label offered by **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 [10].

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

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

$v ::=$	$\underline{n} \mid \texttt{true} \mid \texttt{false}$	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
$\mathbf{p} ::=$	**Client**, **Server**	Participant
$P, Q ::=$		Processes
	$\mathbf{0}$	Nil Process
$\mid$	$\mathbf{p} \rhd \{l_i(x_1 \ldots x_n) : P_i\}_{i \in I}$	Branching
$\mid$	$\mathbf{p} \lhd l\langle e_1 \ldots e_n \rangle.\ P$	Selection
$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional
$\mid$	$\mu X.\ P$	Recursive Process
$\mid$	X	Process Variable
$l, l' ::=$	"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.

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

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 [10] and can be trivially defined as they leverage the syntactic similarities between session types and our session calculus.

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

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

The definition of stuck processes from §2.1.2 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.

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

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].

$$\frac{\cdot \vdash P : S \quad \cdot \vdash Q : \overline{S}}{\vdash \mathbf{p} :: P \mid \mathbf{q} :: Q} \text{ [MTY]}$$

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:

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

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 [12] 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.

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

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

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

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

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

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

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

$$\dfrac{\dfrac{\vdots}{\cdot \vdash P_{\text{Client}} : S_{\text{Client}}} \qquad \dfrac{\dfrac{\vdots}{\cdot \vdash P_{\text{Server}} : S_{\text{Server}}} \qquad \dfrac{\vdots}{S_{\text{Server}} \leqslant \overline{S_{\text{Client}}}}}{\cdot \vdash P_{\text{Server}} : \overline{S_{\text{Client}}}}\text{[Ty-Sub]}}{\vdash \mathbf{Client} :: P_{\text{Client}} \mid \mathbf{Server} :: P_{\text{Server}}}\text{[MTy]}$$

2.1.4 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 [6] 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: [10] and [12] 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 [4] 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 [12] and adapted to be compatible with our session calculus presented in §2.1.3.

$G ::=$		Global Types
	$\mathbf{end}$	Termination
$\mid$	$\mathbf{p} \to \mathbf{q} : \{l_i(U_1 \dots U_n).\ G_i\}_{i \in I}$	Message Exchange
$\mid$	$\mu \mathbf{t}.\ G$	Recursive Type
$\mid$	$\mathbf{t}$	Type Variable

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; [10] presents examples of where this is the case, and [12] 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.

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

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 [1]:

- **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. Figure 5 illustrates this diagrammatically.

ate figure

Figure 5: Type checking with Multiparty Session Types

2.2 The Scribble Protocol Language

2.2.1 Overview

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 [11] 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 6 adapted from [5]. 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 6: Adder Protocol in Scribble

The simplicity of the protocol specification language reflects the design goals for Scribble, as outlined in [11], 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 6 and the global type below written in the calculus.

$$G = \mu \mathbf{t}.\mathbf{C} \to \mathbf{S} : \{ \begin{array}{l} \text{Add}(\texttt{int},\texttt{int}) : \mathbf{S} \to \mathbf{C} : \{\text{Res}(\texttt{int}) : \mathbf{t}\}, \\ \text{Quit}() : \mathbf{S} \to \mathbf{C} : \{\text{Terminate}() : \mathbf{end}\} \end{array} \}$$

2.2.2 Endpoint Finite State Machines

The protocol specification language is a component of the broader Scribble project in [11]; 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.

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

Figure 7: Syntax for Endpoint FSM

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

Figure 8: Types of EFSM States

2.3 Code Generation

2.3.1 Overview

- Motivate the practical implementation of session types in programming
- What needs to be verified - I/O behaviour (introduce the term behaviour typing) and linear channel usage

2.3.2 Static Session Typing

- PureScript implementation that addresses the 2 criteria statically
  - Relevance for web development
  - Static behavioural typing
  - Static channel linearity by construction
  - Pros - type dependencies (expressing dependent types)
  - Cons
- Other initiatives – creating language with first-class channel primitives (e.g. SILL)

PureScript - [6]

2.3.3 Runtime Monitors

- Python dynamic runtime monitors that addresses the 2 criteria dynamically
  - Channel abstraction
  - How invitation/conversation messages are handled

Python - [8]

2.3.4 Hybrid Session Verification

- How the aforementioned 2 approaches motivated hybrid session verification
- Workflow using Scribble toolchain
- Static behavioural typing, dynamic channel usage runtime checks
- Pros - Abstract I/O interfaces
- Pros - Input futures for ‘complex communication patterns’ motivated by SMTP example
- Cons - “practical compromise” as outlined in paper

Java - [5]

2.3.5 Comparison

- Comparison table between the 3 approaches for how they deal with behavioural typing

2.4 TypeScript

2.4.1 Overview

- Background
- Aims
- Example code snippet

2.4.2 Gradual Type System

- Introduce static (Java), dynamic (Python/JS)
- Use pros and cons of the former two systems to motivate the concept of a gradual type system
- Shwo example of unsoundness

2.4.3 Language Features

- Outline features that are either (a) relevant to implementing web applications, (b) differ from the features prevalent in traditional OO (Java) or functional (PureScript) languages

Promises

Relevant for modelling 'state-specific input futures' from the hybrid Java paper?

Discriminated Union

Useful for dealing with the switch case approach for branch receives?

Index Types

With index types, you can get the compiler to check code that uses dynamic property names.

Conditional Types

Compare this with the type dependencies achieved in the PureScript paper?

Chapter 3

Project Plan

3.1 Theory

3.2 Implementation

3.3 Timetable

Chapter 4

Evaluation Plan

4.1 Theory

4.2 Implementation

Bibliography

[1] 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 12

[2] FETTE, I., AND MELNIKOV, A. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. pages 4

[3] 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 5

[4] 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 11

[5] 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 12, 15

[6] 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 4, 10, 15

[7] MILNER, R. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999. pages 4

[8] NEYKOVA, R., AND YOSHIDA, N. How to Verify Your Python Conversations. *Behavioural Types: from Theory to Tools* (2017), 77–98. pages 4, 15

[9] XI, H. Applied type system: An approach to practical programming with theorem-proving. *Journal of Functional Programming* (2016), 30. pages 4

[10] YOSHIDA, N. Lecture Notes in CO406 Concurrent Processes, October 2019. pages 5, 6, 8, 11

[11] 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 12, 13

[12] YOSHIDA, N., AND LORENZO, G. A Very Gentle Introduction to Multiparty Session Types. pages 6, 9, 11