University of Glasgow | School of Computing Science

Session types in the wild

A. Laura Voinea

First Supervisor: Dr. Simon Gay
Second Supervisor: Dr. Wim Vanderbauwhede

3nd Year PhD Report

June 20, 2018

Contents

1 Introduction

Nowadays, distributed systems are ubiquitous and communication is an important feature and reason for its success. Communication-centred programming has proven to be one of the most successful attempts to replace shared memory for building concurrent, distributed systems. Communication is easier to reason about and scales well as opposed to shared memory, making it a more suitable approach for systems where scalability is a must, as in the case of multi-core programming, service-oriented applications or cloud computing[1].

Communication is usually standardised via protocols that specify the possible interactions between the communicating parties in a specific order. Mainstream programming languages fail to adequately support the development of communication-centred software. Thus, implementations of communication behaviours are often based on informal protocol specification, and so informal verification. As a result they are prone to errors such as communication mismatch, when the message sent by one party is not expected by the other party, or deadlock, when parties are waiting for a message from each other causing the system to block.[1].

Session types [15, 30, 16] are types for communication protocols, describing types of the messages being sent as well as the order of communication.Programmers can express a protocol specification as a session type, which can guarantee, within the scope where the session type applies, that communications will always match and the system will never deadlock.

The main goal of the ABCD project[1] is to improve the practice of software development for concurrent and distributed systems through the use of session types. This is to be accomplished through built-in language support for protocol codification in existing languages such as Java or Python, in new languages such as Links[2], inter-language interoperability via session types, and through adapting interactive development environments and modelling techniques to support session types. Logical and automata foundations of session types will be further developed to express a wider class of behaviour and, as need arises, to support the former.

1.1 Thesis Statement

Data types and typing theories have proved to be essential for sequential computation. The session control and discipline that session types provide are similarly important for concurrent, distributed programming. To further incorporate session typing in practical settings, we must be able to express subtle and complex protocols and make safety guarantees. While providing strong guarantees, linearity is often an obstacle for expressivity. Specifying protocols without the full cost of linearity, without sacrificing safety would be highly beneficial.

1.2 Publications

1. Ornela Dardha, Simon J Gay, Dimitrios Kouzapas, Roly Perera, A Laura Voinea, and Florian Weber. Mungo and stmungo: Tools for typechecking protocols in java.

Behavioural Types: from Theory to Tools, page 309, 2017

2. A Laura Voinea and Simon J Gay. Benefits of session types for software development. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 26–29. ACM, 2016

2 Literature review

Session types [15, 30, 16] describe a protocol as a type abstraction, guaranteeing privacy, communication safety and session fidelity. Privacy, since the session channel is owned only by the communicating parties. Communication safety is the requirement that the exchanged data have the expected type. Lastly, session fidelity is a typical property of sessions and is the requirement that the session channel has the expected structure.

Session types are defined as a sequence of input and output operations, explicitly indicating the types of messages being transmitted. Beside this, they permit choice, internal and external, branch and select. A fundamental notion of session types is that of duality. In order to achieve communication safety, a binary session channel is split by giving rise to two opposite endpoints, each of which is owned by one of the interacting participants. These endpoints are required to have dual behaviour and thus have dual types.

Session types are an active area of research aimed at extending their theoretical foundations, as well as developing new tools, and techniques. For example, session types have been augmented with subtyping polymorphism to enable protocols to describe richer behaviours[13]. They have been extended to ensure the progress property and deadlock freedom [11]. Session types have also been extended to multiparty session types to support communication instances with more than two participants while still guaranteeing the absence of deadlock [14]. In other work [7], global session types have been used to detect choreographies that can be realised in the context of web services.

Session types have been successfully applied in many different settings, with implementations varying from language primitives to compiler plugin to libraries or some external tool. Conformance to session types is checked either statically, dynamically or a combination of the two. Static checking ensures that any error, be it sending the wrong message, not completing a session, or duplicating a channel endpoint, will be reported before the program compiles. For dynamic checking conformance to session types is checked at runtime. Session types are compiled into communicating finite-state machines, and messages are verified against these. Lastly, in hybrid checking, sending messages in the right order is checked statically, and linearity is checked dynamically. The language Links [22], designed at the University of Edinburgh, offers support for binary session types as language primitives. Many mainstream languages have various implementations of session types. For C, it is Multiparty Session C [23, 24], where session communication happens using a runtime library, and type-checking is done via a clang plugin. For Haskell there are several options [21, 29, 25], all for binary sessions with full static checking. For OCaml, FuSe [26] implements binary session types, verifying message ordering statically and linearity violations dynamically. For Java, several tools are available. CO2 Middleware [6, 5],based on the theory of timed session types, offers middleware for Java applications, and dynamically monitors conformance to timing constraints. The Scribble Endpoint Generation tool [17] generates endpoint APIs from Scribble [32] protocols. It is multiparty, hybrid approach with message ordering checked statically, and linearity checked dynamically. The Mungo/StMungo [18, 10] toolset is another alternative, providing an external implementation of multiparty session types. This last tool, on which some of my research is carried out, will be detailed below.

2.1 Mungo

A typestate defines the valid sequence of operations that can be performed on an instance of a certain type by associating state information with variables of that type. This state information can then be used at compile-time to determine the operations that can be invoked with valid results on an instance of a type. Mungo[18, 10] is a Java front-end tool, developed at the University of Glasgow, used to statically typecheck Java programs augmented with typestate Mungo implements two main components. First, a Java-like syntax to define typestate specifications for classes, and second, a typechecker that checks whether objects that have typestate specifications are used correctly. Typestates are specified in separate files and are associated with Java classes by means of a Java annotation. This allows programs that have been checked by Mungo to be compiled and run using standard Java tools. If a class has a typestate specification, the Mungo typechecker analyses each object of that class in the program and extracts the method call behaviour (sequences of method calls) through the object's life. Finally, it checks the extracted information against the sequences of method calls allowed by the typestate specification.

Mungo supports typechecking for a subset of Java. The programmer can define both classes that follow a typestate specification and classes that do not. The typechecking procedure follows objects (instances of the former classes) through argument passing and return values. Moreover, the typechecking procedure for the fields of a class follows the typestate specification of the class to infer a typestate usage for the fields. For this reason fields that follow a typestate specification are only allowed to be defined in a class that also follows a typestate specification.

3 Work Undertaken

Over the past year the focus of my research has shifted to a more theoretical one. As such, some of the goals stated last year have been met, while others have been superseded by new ones. An overview of which I present here.

3.1 Modular linearity

Existing work typically assumes linear type systems as necessary for session types. In order to maintain session fidelity and ensure that all communication actions in a session type occur, session type systems typically require that endpoints are used linearly: each endpoint must be used exactly once. This work sets out to investigate whether linear types are necessary to provide resource control/ownership for session types; what are the characteristics necessary to assure session fidelity; what other semantic properties should/might be guaranteed; how can more general approaches to resource control/ownership be integrated with session types.

The *pi*-calculus with session types has been chosen as a suitable starting point for its simplicity and expressivity. An overview of the syntax, semantics and typing rules is to be found in the appendix.

To begin to explore these questions this work has taken two directions. The first looks at general approaches to deal with control/ownership for session types. The second looks at analysing properties of the type system, especially linearity and how it is used to guarantee session safety, separately from the format of the typing rules. Both are described below.

3.1.1 Capabilities in π-calculus with sessions

We present a simple type system based on π-calculus with session types and enriched with capabilities. Our approach builds on work done to deal with memory management, explicit region deallocation in the Capability Calculus [9]; changing the type of a mutable object whenever the contents of the object is changed [3]; or aliasing data structures that point to linear objects [12]. The work presented below is particularly inspired by the work done by Fähndrich and DeLine [12]. The formulation allows aliasing while ensuring through capabilities that all communication actions in a session type occur.

$$
\begin{array}{llcll}
\text{Ordinary types} & T & ::= & int \mid Bool & \\
 & & & & \\
\text{Session types} & S & ::= & !S.U & \text{output} \\
 & & \mid & ?S.U & \text{input} \\
 & & \mid & \oplus\{\ell_i : S_i\}_{i \in I} & \text{choice} \\
 & & \mid & \&\{\ell_i : S_i\}_{i \in I} & \text{branch} \\
 & & \mid & \mathsf{end} & \text{terminated session} \\
 & & & & \\
\text{Cababilities} & C & ::= & \emptyset \mid \{\rho \to S\} \otimes C \mid \epsilon \otimes C & \\
 & & & & \\
\text{Types} & \tau & ::= & tr(\rho) & \text{tracked type} \\
 & & \mid & T & \text{ordinary types} \\
 & & \mid & S & \text{session types} \\
 & & \mid & C & \text{capabilities} \\
 & & & & \\
\text{Type Environments} & \Gamma & ::= & \emptyset \mid \Gamma, x : T & \\
 & \Delta & ::= & \emptyset \mid \rho, \Delta \mid \epsilon, \Delta &
\end{array}
$$

Figure 1: Types

The type language distinguishes several kinds of types. Ordinary types, ranged over by T, are integers and booleans. The syntax of session types is standard. Let S, U range over types. A type can be end, the type of the terminated channel where no communication can take place any longer. $!S.U$ and $?S.U$ are types for sending or receiving a value of type S with continuation of type U. $\oplus\{\ell_i : S_i\}_{i \in I}$ and $\&\{\ell_i : S_i\}_{i \in I}$ are sets of labelled types indicating, respectively, internal and external choice. The labels are all different and the order of the labelled types does not matter. Capabilities, ranged over by C map a channel name to a session type S. When communication starts, a new capability is added to the set, when the communication ends, it is removed. The types τ range over ordinary types, session types, capabilities, and tracked types. Tracked types are singleton types, $tr(\rho)$, where the key ρ is a static name for the channel being tracked. We separate the handle to a linear object, $tr(\rho)$, from the capability to access that type.

The conjunction of capabilities is formed via $C_1 \otimes C_2$, expressing that keys in C_1 are disjoint from keys in C_2.

$$\begin{array}{rcll}
P, Q & ::= & \mathbf{0} & \text{(Inaction)} \\
& | & x!\langle v \rangle.P & \text{(Send)} \\
& | & x?(y).P & \text{(Receive)} \\
& | & x \triangleleft \ell_j.P & \text{(Selection)} \\
& | & x \triangleright \{\ell_i : P_i\}_{i \in I} & \text{(Branching)} \\
& | & P \mid Q & \text{(Composition)} \\
& | & (\nu\, xy)P & \text{(Restriction)} \\
& | & \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q & \text{(Conditional)} \\
v & ::= & x & \text{(Variable)} \\
& | & \mathsf{true} \mid \mathsf{false} & \text{(Booleans)}
\end{array}$$

Figure 2: Processes

The syntax of processes is standard. Let P, Q range over processes, x, y over variables, v over values, and l over labels. Process **0** is the terminated process. The send process $x!\langle v \rangle.P$ sends a value v on channel x and proceeds as process P; the receive process $x?(y).P$ receives a value on channel x, stores it in variable y and proceeds as P. The internal choice process $x \triangleleft \ell_j.P$selects label ℓ_j on channel x and proceeds as P. The branching process $x \triangleright \{\ell_i : P_i\}_{i \in I}$ offers a range of labelled alternatives on channel x, followed by their respective process continuation. The order of labelled processes is not important and the labels are all different. Process $\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q$ proceeds as either P or Q depending on the value of boolean v. Process $\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q$ is the parallel composition of processes P and Q. Process $(\nu\, xy)P$ restricts variables x, y with scope P. It states that variables x and y are bound with scope P, and most importantly, are bound together, by representing two endpoints of the same channel. When occurring under the same restriction, x and y are called co-variables.

Typing judgments have the form $\Delta; \Gamma \vdash P; C$. Γ maps program variables to nonlinear types T only. Linearity is enforced via capabilities, rather than via environment splitting as is the case in the standard π-calculus with sessions. A process is either correctly typed or not; we do not assign types to processes.

$$\frac{}{\Delta;\Gamma, x : T \vdash x : T}\ \text{(TVar)}$$

$$\frac{\Delta;\Gamma \quad v \in T}{\Delta;\Gamma \vdash v : T, C}\ \text{(TVal)}$$

$$\frac{}{\Delta;\Gamma, x : tr(\rho_x) \vdash x : tr(\rho_x); \{\rho_x \mapsto S\}}\ \text{(TVar)}$$

$$\frac{C \text{ completed}}{\Delta;\Gamma \vdash \mathbf{0}; C}\ \text{(TInact)}$$

$$\frac{\Delta;\Gamma \vdash P; C_P \quad \Delta;\Gamma \vdash Q; C_Q}{\Delta;\Gamma \vdash P \,|\, Q; C_P \otimes C_Q}\ \text{(TPar)}$$

$$\frac{\Delta;\Gamma, x : tr(\rho_x), y : tr(\rho_y) \vdash P; C \otimes \{\rho_x \mapsto S\} \otimes \{\rho_y \mapsto \overline{S}\}}{\Delta;\Gamma \vdash (\nu\, xy)P; C_P}\ \text{(TRes)}$$

$$\frac{\Delta;\Gamma \vdash x : Bool \quad \Delta;\Gamma \vdash P; C \quad \Delta;\Gamma \vdash Q; C}{\Delta;\Gamma \vdash \mathsf{if}\, x\, \mathsf{then}\, P\, \mathsf{else}\, Q; C}\ \text{(TCond)}$$

$$\frac{\Delta, \rho_y;\Gamma, x : tr(\rho_x), y : tr(\rho_y) \vdash P; C \otimes \{\rho_x \mapsto U\} \otimes \{\rho_y \mapsto S\}}{\Delta;\Gamma, x : tr(\rho_x) \vdash x?(y).P; C \otimes \{\rho_x \mapsto \forall[\rho_y].?S.U\}}\ \text{(TRcv)}$$

$$\frac{\Delta;\Gamma, x : tr(\rho_x), y : T \vdash P; C \otimes \{\rho_x \mapsto U\}}{\Delta;\Gamma, x : tr(\rho_x) \vdash x?(y).P; C \otimes \{\rho_x \mapsto ?T.U\}}\ \text{(TRcvVal)}$$

$$\frac{\Delta, \rho_v;\Gamma, x : tr(\rho_x) \vdash P; C \otimes \{\rho_x \mapsto U\} \quad \Delta;\Gamma \vdash v : tr(\rho_v); \{\rho_v \mapsto S\}}{\Delta;\Gamma, x : tr(\rho_x) \vdash x!\langle v\rangle.P; C \otimes \{\rho_x \mapsto \forall[\rho_v].!S.U\} \otimes \{\rho_v \mapsto S\}}\ \text{(TSnd)}$$

$$\frac{\Delta;\Gamma, x : tr(\rho_x) \vdash P; C \otimes \{\rho_x \mapsto U\} \quad \Delta;\Gamma \vdash v : T}{\Delta;\Gamma, x : tr(\rho_x) \vdash x!\langle v\rangle.P; C \otimes \{\rho_x \mapsto !T.U\}}\ \text{(TSndVal)}$$

$$\frac{\Delta;\Gamma, x : tr(\rho_x) \vdash P; C \otimes \{\rho_x \mapsto S_j\}, j \in I}{\Delta;\Gamma, x : tr(\rho_x) \vdash x \triangleleft \ell_j.P; C \otimes \{\rho_x \mapsto \oplus\{\ell_i : S_i\}_{i \in I}\}}\ \text{(TSel)}$$

$$\frac{\Delta;\Gamma, x : tr(\rho_x) \vdash P_i; C \otimes \{\rho_x \mapsto S_i\}, \forall i \in I}{\Delta;\Gamma, x : tr(\rho_x) \vdash x \triangleright \{\ell_i : P_i\}_{i \in I}; C \otimes \{\rho_x \mapsto \&\{\ell_i : S_i\}_{i \in I}\}}\ \text{(TBr)}$$

Figure 3: Typing rules

Rule (TVar) states that a variable x is of type T, if this is the type assumed in the typing context. Rule (TVal) states that a value v, being either true or false, is of type Bool. (TInact) states that the terminated process **0** is always well-typed. (TPar) types the parallel composition of two processes, using the split operator for typing contexts ∘ which ensures that each linearly-typed channel x, is used linearly, i.e., in $P|Q$, x occurs either in P or in Q but never in both. Rule (TRes) states that $(\nu xy)P$ is well typed if P is well typed and the co-variables have dual types, namely T and $\overline{T}$. Rule (TCond) states that the conditional statement is well typed if its guard is typed by a boolean type and the branches are well typed under the same typing context. Rules (TRcv) and (TSnd) type the receiving and the sending of a value. Rule (TBr) types an external choice on channel x, checking that each branch continuation P_i follows the respective continuation type of x. Dually, rule (TSel) types an internal choice communicated on channel x, checking that the chosen label is among the ones offered by the receiver and that the continuation proceeds as expected by the type of x.

The operational semantics of the language is standard, and can be seen in the appendix.

Proofs that this system still offers the same guarantees as the standard *pi*-calculus with session, namely proofs of progress and type preservation, are under way.

An implementation of this work is planned for later in the year. It will be implemented within the Mungo tool[18], developed at Glasgow University. This will allow for more flexible aliasing within Mungo.

3.1.2 A semantic approach to π-calculus with sessions

To explore what characteristics are necessary to assure session fidelity or what other semantic properties might be guaranteed; we have taken a semantic approach in the style of [4]. This is achieved through defining logical relations that explain the meaning of the type in terms of the operational semantics of the language. Using this model of types, I prove each typing rule as a lemma.

Logical relations have been used in the works of Pérez et al. in [27] and in [28] for proving properties about session types in an intuitionistic linear logic setting. However, this approach remains largely unexplored in the context of session types. In particular, we want to analyse properties of the type system, especially linearity and how it is used to guarantee session safety, separately from the format of the typing rules.

This work has been started fairly recently, with definitions being under construction.

3.2 A Paxos implementation

Paxos is a protocol for solving consensus in a network of unreliable processes. It ensures that a single value among the proposed values can be chosen. It assumes an asynchronous, non-Byzantine model.[19]

Two major advantages of Paxos are that it is provably correct in asynchronous networks that eventually become synchronous and it does not block if a majority of participants are available. Furthermore, it has provably minimal message delays in the best case. Despite its reputation of being difficult to understand & implement it is widely used in systems such as Google's Chubby[8] or Yahoo's ZooKeeper. The protocol comes with three roles and a two-phase approach. A proposer responsible for initiating the protocol, that handles client requests and proposes values to be chosen. An acceptor that responds to messages from proposers by either rejecting them or agreeing in principle and making a promise about the proposals it will accept in the future. An a listener or learner, who wants to know which value was chosen. Each Paxos server can act as any or all 3 roles.[20] Some representations of the algorithm have been attempted using Scribble, combined with StMungo and Mungo to give a working Java code. However in trying to represent it some shortcomings of the toolset became apparent: representing broadcasting, representing quorum/a majority, representing express the dynamic aspects such as processes failing, restarting, expressing multiple instances of the protocol

Since, a successful implementation of Paxos has been achieved using the session type system for unreliable broadcast communication devised by Gutkovas, Kouzapas and Gay. The example has been written up as part of a paper describing the type system and will be submitted to the Logical Methods in Computer Science journal[1].

I will first briefly introduce the language I have used, followed by the implementation itself.

3.2.1 Implementation language

Syntax. Assume the following disjoint countable sets of names: $\mathcal{C}$ is the set of *shared channels* ranged over by $a, b, c, \ldots$; $\mathcal{S}$ is the set of *session channels* ranged over by $s, s', \ldots$, where each channel has two distinct endpoints s and $\check{s}$ (we write κ to denote either s or $\check{s}$); $\mathcal{V}$ is the set of *variables* ranged over by $x, y, z, \ldots$; and Lab is the set of labels ranged over by $\ell, \ell', \ldots$. We write k to denote either κ or x or $\check{x}$, where $\check{x}$ is used to distinguish a variable used as a $\check{s}$-endpoint.

Let $\mathcal{E}$ be a non-empty set of *expressions* ranged over by $e, e', \ldots$. Assume a binary operation $\odot$ on $\mathcal{E}$ called *aggregation*, and element $\mathbf{1} \in \mathcal{E}$ called *unit*. Let $\mathcal{F} \subseteq \mathcal{E}$ be a non-empty set of *conditions* ranged over by φ, and assume a truth predicate $\Vdash \subseteq \mathcal{F}$. Elements of $\mathcal{E}$ may contain variables. Assume a substitution function $e\{e'/x\}$.

The syntax of processes is then defined as:

$$\begin{array}{rclrclrclr}
P & ::= & \mathbf{0} & \textsf{(Inaction)} & \mid & a!(\check{x}).P & \textsf{(Request)} & \mid & a?(x).P & \textsf{(Accept)} \\
& \mid & k!\langle e\rangle.P & \textsf{(Send)} & \mid & k?(x).P & \textsf{(Receive)} & \mid & k \triangleleft \ell.P & \textsf{(Selection)} \\
& \mid & k \triangleright \{\ell_i : P_i\}_{i\in I} & \textsf{(Branching)} & \mid & \multicolumn{4}{l}{\mathsf{if}\,\varphi\,\mathsf{then}\,P\,\mathsf{else}\,P} & \textsf{(Conditional)}
\end{array}$$

(Inaction) is the inactive term. (Request) and (Accept) express the terms that are ready to initiate a fresh session on a shared channel a via a request/accept interaction, respectively. (Send) defines a prefix ready to send an expression on k, whereas term (Receive) defines a prefix ready to receive a message on k and substitute it on x. Session selection is defined by term (Selection) where the prefix sends a label ℓ over session k. Dually, session branching is defined by term (Branching) where the prefix receives a label from a predefined set of label $\{\ell_i\}_{i\in I}$.

To introduce networks, assume a countable set of *networks* $\mathcal{N}$ ranged over by N. Let symbol

[1] https://lmcs.episciences.org

m, called *message*, range over by expressions e and labels ℓ, with $\tilde{m}$ a message vector. Also, let h denote a message tagged with a natural number c, (c, e).

$$
\begin{array}{rclllll}
B & ::= & \varepsilon & \mid & B|s[c,\tilde{m}] & \mid & B|\check{s}[c,\tilde{h}] \\
N, M & ::= & [P \triangleright R|B] \quad \text{(Node)} & \mid & N|M \quad \text{(Parallel)} & \mid & (\nu\, n)N \quad \text{(Restriction)}
\end{array}
$$

Term, B, is a parallel composition of session buffers, that are used to store messages and to count session state via natural number c. Buffer terms are used to model a form of asynchrony that preserves the order of received messages. The purpose of counter c is to keep track of the session state in the presence of a lossy communication and to synchronise the interaction between session prefixes. Message loss in an unreliable setting leads to participants that are not synchronised with the overall protocol.

Buffer terms on s-endpoints store messages m, while Buffer terms on $\check{s}$-endpoints store expression messages tagged with a session counter, $h = (c, e)$. The session counter in h distinguishes the session state at which the expression e was be received. Network (Node) consists of a process P, a recovery process R that may take over if P cannot proceed in a session, and the necessary buffer terms used for asynchronous session communication. A process may participate in several sessions, and therefore, more than one buffer term may be present in a node. The type system ensures that there is no more than one buffer term on the same session in each network node. We write $[P \triangleright R]$ for node $[P \triangleright R|\varepsilon]$.

A network is a parallel composition of nodes — term (Parallel). We may write $\prod_{i\in I} N_i$ for the parallel composition of $N_1|\cdots|N_n$ for (possibly empty) $I = \{1, \ldots, n\}$. Network (Restriction) binds both session and shared channels.

Definition 1 (Session Type) *Let $\mathcal{B}$ be a set of* base types *ranged over by β. Session types are inductively defined by the following grammar:*

$$
T \;::=\; !\beta.T \mid ?\beta.T \mid \oplus\{\ell_i : T_i\}_{i\in I} \mid \&\{\ell_i : T_i\}_{i\in I} \mid \mathsf{end} \mid t \mid \mu t.T
$$

We say that two types T_1 and T_2 are dual if $\overline{T}_1 = T_2$. Note $\overline{\overline{T}} = T$ for any T.

$$
\begin{array}{rclrclrclrcl}
\overline{\mathsf{end}} & = & \mathsf{end} & \overline{!\beta.T} & = & ?\beta.\overline{T} & \overline{\oplus\{\ell_i : T_i\}_{i\in I}} & = & \&\{\ell_i : \overline{T}_i\}_{i\in I} & \overline{t} & = & t \\
 & & & \overline{?\beta.T} & = & !\beta.\overline{T} & \overline{\&\{\ell_i : T_i\}_{i\in I}} & = & \oplus\{\ell_i : \overline{T}_i\}_{i\in I} & \overline{\mu t.T} & = & \mu t.\overline{T}
\end{array}
$$

Definition 2 (Typing Context) *We define* Γ, Δ, *and* Θ *typing contexts:*

$$
\begin{array}{l}
\Gamma \;::=\; \emptyset \mid \Gamma, a : T \mid \Gamma, x : \beta \qquad \Delta \;::=\; \emptyset \mid \Delta, k : T \mid \Delta, s : c \\
\Theta \;::=\; \emptyset \mid \Theta, \kappa : M \mid \Theta, s : c
\end{array}
$$

Shared context Γ tracks shared names used in a process, while context Δ tracks linear (session) names or session variables used in a process. Context Θ is also linear and tracks the types of buffer terms. We denote by Γ, Γ' the concatenation of contexts Γ and Γ', and similarly for Δ, Δ' and Θ, Θ'.

3.2.2 Implementation

The session type representation of Paxos is used to check that implementations correctly follow the protocol, rather than correctness of the protocol itself. Session types can also help to identify subtle interactions such as branching or dropping sessions. Furthermore, a session type representation allows the basic algorithm to be easily extended while still providing formal guarantees.

Paxos agents implement various roles: i) a *proposer* initiates the paxos rounds, in which it proposes a value to the acceptors; ii) an *acceptor* will accept a proposal if it is from the latest round i.e. with the highest round number. A value accepted form a majority of acceptors(quorum) signifies the reach of consensus and protocol termination; and iii) a *learner* is an agent that wants to know which value has been chosen. The Paxos setting assumes lossy communication. Agents may crash and recover. Proposers record their highest round to stable storage and begin a new round with a higher round number than previously usedIf eventually a majority of the acceptor agents run for long enough without failing, consensus is guaranteed.

We implement the most basic protocol of the Paxos family, as introduced in [20]. The protocol proceeds over several rounds. A round has two phases, Prepare and Accept. Each run of the protocol decides on a single consensus value.

The Paxos communication interaction is described by session type PaxosType:

$$!\mathtt{prep}.?\mathtt{prom}.\oplus\left\{\begin{array}{ll}\mathsf{restart}: & \mathsf{end},\\ \mathsf{accept}: & !(\mathtt{rnd},\mathtt{value}).?\mathtt{rnd}.\oplus\{\mathsf{restart}:\mathsf{end},\mathsf{chosen}:\mathsf{end}\}\end{array}\right\}$$

A Paxos agent is described by network node $\mathsf{PaxosNode}_{n,v} = [\mathsf{Paxos}_{n,v} \triangleright \mathsf{Paxos}_{n,v}]$ with n being the (fresh) number of the current protocol round, v being the *consensus value* the agent currently holds and process $\mathsf{Paxos}_{x,y}$ defined as:

$$\begin{array}{lcl}
\mathsf{def} & & \\
\quad\mathsf{Proposer}(x,y) & \overset{\mathsf{def}}{=} & a!(s).\breve{s}!\langle x\rangle.\breve{s}?(\{(n,v)_i\}_{i\in I}).\\
& & \mathsf{if}\ |\{(n,v)_i)\}_{i\in I}| \leq \frac{M}{2}\ \mathsf{then}\ \breve{s}\triangleleft\mathsf{restart}.\mathsf{Paxos}\langle x+1,y\rangle\\
& & \mathsf{else\ if}\ \mathsf{max}(\{n_i\}_{i\in I}) > x\ \mathsf{then}\ \breve{s}\triangleleft\mathsf{restart}.\mathsf{Paxos}\langle x+1,y\rangle\\
& & \mathsf{else}\ \breve{s}\triangleleft\mathsf{accept}.\\
& & \quad\breve{s}!\langle x, v_h = \mathsf{max}(\{v\,|\,(n,v)\in\{(n,v)_i\}_{i\in I}\})\rangle.\breve{s}?(\{n_i\}_{i\in I}).\\
& & \quad\mathsf{if}\ |\{n_i\}_{i\in I}| \leq \frac{M}{2}\ \mathsf{then}\ \breve{s}\triangleleft\mathsf{restart}.\mathsf{Paxos}\langle x+1,y\rangle\\
& & \quad\mathsf{else}\ \breve{s}\triangleleft\mathsf{chosen}.\mathsf{Paxos}\langle x,v_h\rangle\\
\quad\mathsf{Acceptor}(x,y) & \overset{\mathsf{def}}{=} & a?(s).s?(x').\ s!\langle x,y\rangle.(\mathsf{Acc}\langle s,x,y\rangle\ +\ a?(s').s'?(x'').\\
& & \mathsf{if}\ (x'' > x')\ \mathsf{then}\ s'!\langle x,y\rangle.\mathsf{Acc}\langle s',x,y\rangle\ \mathsf{else}\ \mathsf{Acc}\langle s,x,y\rangle)\\
\quad\mathsf{Acc}(w,x,y) & \overset{\mathsf{def}}{=} & w\triangleright\left\{\begin{array}{ll}\mathsf{restart}: & \mathsf{Paxos}\langle x,y\rangle,\\ \mathsf{accept}: & w?(x',y').w!\langle x'\rangle.w\triangleright\left\{\begin{array}{ll}\mathsf{restart}: & \mathsf{Paxos}\langle x,y\rangle,\\ \mathsf{chosen}: & \mathsf{Paxos}\langle x',y'\rangle\end{array}\right\}\end{array}\right\}\\
\quad\mathsf{Paxos}(x,y) & \overset{\mathsf{def}}{=} & \mathsf{Proposer}\langle x,y\rangle + \mathsf{Acceptor}\langle x,y\rangle\\
\mathsf{in}\ \mathsf{Paxos}\langle n,v\rangle & &
\end{array}$$

Unlike [20], our implementation allows a agent to non-deterministically behave either as a proposer, (definition Proposer (x, y)), or an acceptor (definition Acceptor (x, y)).

A proposer interacts within the same round with with multiple acceptors. The communication behaviour of an Acceptor is described by the session type PaxosType, whereas the communication behaviour of the proposer is described by the dual type $\overline{\mathsf{PaxosType}}$.

During a round a Paxos agent may restart in which case it terminates its current sessions and proceeds to the initial Paxos network, $\mathsf{Paxos}_{x,y}$. Note that each time an initial Paxos agent enters a new protocol round it establishes a new session. Recovery by a Paxos agent is equivalent to a Paxos agent restart.

If a Paxos agent decides to behave as a proposer, it first requests session communication and enters the Prepare phase. All Paxos agents that accept a session request behave as acceptors. The proposer then broadcasts a *prepare*,type prep, request with a fresh round (or session) number n.

All the acceptors that received the *prepare* message reply with a *promise*, type prom, not to respond to a prepare message with a lower round number. The promise message contains the current round number and the current value of the acceptor. All the promises are gathered in a set by the proposer, which then checks whether the majority of acceptors have replied, i.e. $|\{(n,v)_i)\}_{i \in I}| \leq \frac{M}{2}$, with M the number of expected connected acceptors.

If the check fails the proposer sends a restart label to all the acceptors, and restarts its own computation with an increment on its round number, as in $\mathsf{Paxos}\langle x+1, y\rangle$. All acceptors that receive label restart restart. If majority is achieved, the proposer checks whether any of the acceptors has promised to reply on higher round numbers than x, in which case the proposer increments its round number and restarts all agents within the session via label restart.

If both of these checks are passed, the protocol enters the Accept phase. The proposer selects a value to submit to the acceptors by inspecting the promises received and selecting the highest value, i.e. $v_h = \mathsf{max}(\{v \,|\, (n,v) \in \{(n,v)_i\}_{i \in I}\})$. It then broadcasts an *accept* message; selects label accept followed by the current round number together with the chosen highest value. The acceptors reply with a message of type rnd, containing their current round number. These messages are gathered by the proposer and checked for majority, in which case, consensus has been reached and the acceptors will be informed via label chosen. All informed agents restart by updating their round number and consensus value. Otherwise, if lack of majority is detected, the proposer increments its round number and restarts all agents within the session via label restart.

On the acceptor side we use the non-deterministic construct to capture the case of session initiations from multiple proposers. In such a case, the session with the lowest round number is dropped. A dropped session by an acceptor will have an impact to the majority check by the corresponding proposer of the session, thus checking for majority is crucial for reaching consensus.

Following the semantics of our framework, messages can be arbitrarily lost, thus triggering the recovery procedure for the corresponding Paxos agent. Lost messages are taken into account by the logic of the Paxos protocol and have an impact on the execution of the algorithm. For example, lost messages lead to failure to reply by the majority of acceptors, which implies failure to reach consensus within a round, thus restarting a new protocol round.

We can type the $\mathsf{PaxosNode}_{n,v}$ node as: $\Gamma, a : \mathsf{PaxosType}; \emptyset \vdash \mathsf{PaxosNode}_{n,v}$. Shared channel a uses type PaxosType, thus all establish sessions follow the behaviour described by the PaxosType session type. We can then define a well-typed network, N, that runs the Paxos protocol: $N = \prod_{i \in I} \mathsf{PaxosNode}_{n_i,v_i}$ with $\Gamma, a : \mathsf{PaxosType}; \emptyset \vdash N$. Typing is possible due to the typing of network node $\mathsf{PaxosNode}_{n,v}$ and multiple applications of rule (TPar).

An implementation of this work is planned for later in the year. It would be interesting to see whether any of the typing can be expressed/checked in Mungo, even if not all aspects are there.

3.3 Other Activities

As part of the third year of PhD, various additional activities have been undertaken, such as attending conferences, ABCD group meetings of various sizes, seminars and talks(e.g. The Scottish programming language seminar series, FATA seminars) which improved my knowledge of the field and gave me some insight of the exciting research ongoing. Notably, I have attended the Oregon Programming Languages Summer School[2] which was a great opportunity to learn more about programming languages, from foundational work on semantics and type theory to advanced program verification techniques; and had a big impact on the direction my research has taken since.

4 Future Work

Gantt chart	2018							2019								
	06	07	08	09	10	11	12	01	02	03	04	05	06	07	08	09
A. Modular linearity with Capabilities	■	■	■													
B. Semantic approach to modular linearity	■	■	■	■	■											
C. Paxos & Mungo						■	■	■	■							
D. Thesis write-up							■	■	■	■						

The last six months will be used for any task that overruns its allotted time.

A Modular linearity with Capabilities Over the coming months my focus will be on finishing and publishing this work.

B Semantic approach to modular linearity I have already begun research on this, so this will be carried out in the next five months.

C Paxos & Mungo The implementation of Paxos may need some refinement as the language in which is implemented might itself be refined. This however is expected

[2] https://www.cs.uoregon.edu/research/summerschool/summer17/

to be a minor task. The bulk of the work remaining is exploring the extent to which this can be translated into the Mungo setup and implementing any extensions necessary to support it.

D Thesis write-up In the final four months outlined here I will focus exclusively on the write up of the thesis. As writing material relevant to the final thesis is a significant part of my day to day activities I do not foresee there being any issue in producing the final thesis in this timescale. I expect the thesis will structured as following:

(a) Introduction
  i. Motivation and Objectives
  ii. Contributions
  iii. Publications

(b) Background theory
  i. Introduction

(c) Modular linearity with Capabilities
  i. Introduction

(d) Modular linearity, a semantic approach
  i. Introduction

(e) Mungo and Paxos
  i. Introduction

(f) Conclusion
  i. Summary of Thesis Achievements
  ii. Future Work

A π-calculus with sessions

A.1 Syntax

Definition 3 (π-calculus with sessions)

$$
\begin{array}{rcll}
P, Q & ::= & \mathbf{0} & \text{(Inaction)} \\
 & | & x!\langle v \rangle.P & \text{(Send)} \\
 & | & x?(y).P & \text{(Receive)} \\
 & | & x \triangleleft \ell_j.P & \text{(Selection)} \\
 & | & x \triangleright \{\ell_i : P_i\}_{i \in I} & \text{(Branching)} \\
 & | & P \mid Q & \text{(Composition)} \\
 & | & (\nu\, xy)P & \text{(Restriction)} \\
 & | & \text{if}\, v\, \text{then}\, P\, \text{else}\, Q & \text{(Conditional)} \\
v & ::= & x & \text{(Variable)} \\
 & | & \text{true} \mid \text{false} & \text{(Booleans)}
\end{array}
$$

Let P, Q range over processes, x, y over variables, v over values, and l over labels. Process **0** is the terminated process. The send process $x!\langle v \rangle.P$ sends a value v on channel x and proceeds as process P; the receive process $x?(y).P$ receives a value on channel x, stores it in variable y and proceeds as P. The internal choice process $x \triangleleft \ell_j.P$selects label ℓ_j on channel x and proceeds as P. The branching process $x \triangleright \{\ell_i : P_i\}_{i \in I}$ offers a range of labelled alternatives on channel x, followed by their respective process continuation. The order of labelled processes is not important and the labels are all different. Process $\text{if}\, v\, \text{then}\, P\, \text{else}\, Q$ proceeds as either P or Q depending on the value of boolean v. Process $\text{if}\, v\, \text{then}\, P\, \text{else}\, Q$ is the parallel composition of processes P and Q. Process $(\nu\, xy)P$ restricts variables x, y with scope P. It states that variables x and y are bound with scope P, and most importantly, are bound together, by representing two endpoints of the same channel. When occurring under the same restriction, x and y are called co-variables.

A.2 Semantics

Definition 4 (Structural congruence for the π-calculus with sessions)

$$
\begin{array}{rcl}
P \mid Q & \equiv & Q \mid P \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\
P \mid \mathbf{0} & \equiv & P \\
(\nu\, xy)\mathbf{0} & \equiv & \mathbf{0} \\
(\nu\, xy)(\nu\, zw)P & \equiv & (\nu\, zw)(\nu\, xy)P \\
(\nu\, xy)P \mid Q & \equiv & (\nu\, xy)(P \mid Q)(x, y \notin \text{fv}(Q))
\end{array}
$$

Definition 5 (Reduction for the π-calculus with sessions)

$$(\nu\, xy)(x!\langle v\rangle.P \mid y?(z).Q) \longrightarrow (\nu\, xy)(P \mid Q[v/z]) \;\; \text{(RCom)}$$

$$(\nu\, xy)(x \triangleleft \ell_j.P \mid y \triangleright \{\ell_i : P_i\}_{i\in I}) \longrightarrow (\nu\, xy)(P \mid P_j) j \in I \;\; \text{(RSel)}$$

$$\text{if true then } P \text{ else } Q \longrightarrow P \;\; \text{(RTrue)}$$

$$\text{if false then } P \text{ else } Q \longrightarrow Q \;\; \text{(RFalse)}$$

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q \equiv Q'}{P \longrightarrow Q} \;\; \text{(RCong)}$$

$$\frac{P \longrightarrow Q}{(\nu\, xy)P \longrightarrow (\nu\, xy)Q} \;\; \text{(RRes)}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \;\; \text{(RPar)}$$

A.3 Session Types

Definition 6 (Syntax of Session Types)

$$T \;\; ::= \;\; !T.U \;\mid\; ?T.U \;\mid\; \oplus\{\ell_i : T_i\}_{i\in I} \;\mid\; \&\{\ell_i : T_i\}_{i\in I} \;\mid\; \text{end} \;\mid\; Bool$$

Where $!T.U$,$?T.U$,$\oplus\{\ell_i : T_i\}_{i\in I}$,$\&\{\ell_i : T_i\}_{i\in I}$ are linear; and end,$Bool$ are unrestricted. We define predicates un(S) and lin(S) as follows:

- un(S) if and only if S = end, or $S = Bool$
- lin(S) if and only if $S = !T.U$, or $S = \oplus\{\ell_i : T_i\}_{i\in I}$, or $S = \&\{\ell_i : T_i\}_{i\in I}$, or $S = ?T.U$.

Definition 7 (Type duality)

$$\overline{\text{end}} = \text{end} \quad \begin{array}{rcl} \overline{!T.U} & = & ?T.\overline{U} \\ \overline{?T.U} & = & !T.\overline{U} \end{array} \quad \begin{array}{rcl} \overline{\oplus\{\ell_i : T_i\}_{i\in I}} & = & \&\{\ell_i : \overline{T}_i\}_{i\in I} \\ \overline{\&\{\ell_i : T_i\}_{i\in I}} & = & \oplus\{\ell_i : \overline{T}_i\}_{i\in I} \end{array} \quad \overline{Bool} \;=\; Bool$$

Definition 8 (Typing Context) *The syntax of typing contexts is defined as follows:*

$\Gamma \;\; ::= \;\; \emptyset \;\mid\; \Gamma, x : T$

We consider the typing context Γ to be a partial function from variables to types.

Γ is unlimited, $un(\Gamma)$ if it contains no linear types, $lin(T)$.

By Γ, Γ' we denote the concatenation of contexts Γ and Γ'. Γ and Γ' must have disjoint domains. The definition of context split follows.

Definition 9 (Context split)

$$\frac{}{\emptyset = \emptyset \circ \emptyset} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad un(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad lin(T)}{\Gamma, x : T = \Gamma_1, x : T \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad lin(T)}{\Gamma, x : T = \Gamma_1 \circ \Gamma_2, x : T}$$

The context split operator, $\circ$, adds a linear type $\mathsf{lin}(T)$ to either Γ_1 or Γ_2, when $\Gamma_1 \circ \Gamma_2$ is defined. When $\mathsf{lin}(T)$ is added to Γ_1 it is not present in Γ_2 and vice versa, when it is added to Γ_2 it is not present in Γ_1. If $\mathrm{un}(T)$, then it is possible to add this type to both Γ_1 and Γ_2.

Let $\mathrm{dom}(\Gamma)$ denote the set of variables x such that $x : T$ is in Γ, and let $\mathcal{U}(\Gamma)$ denote the typing context containing exactly the entries $x : T$ in Γ such that $\mathrm{un}(T)$.

Lemma 1 (Properties of context split) *Let $\Gamma = \Gamma_1 \circ \Gamma_2$. Then:*

1. $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$
2. *if $x : \mathsf{lin}(T) \in \Gamma$ then either $x : \mathsf{lin}(T) \in \Gamma_1$ and $x \notin \mathrm{dom}(\Gamma_2)$, or $x : \mathsf{lin}(T) \in \Gamma_2$ and $x \notin \mathrm{dom}(\Gamma_1)$*
3. $\Gamma = \Gamma_2 \circ \Gamma_1$
4. *$\Gamma_1 = \Delta_1 \circ \Delta_2$ then $\Delta = \Delta_2 \circ \Gamma_2$ and $\Gamma = \Delta_1 \circ \Delta$.*

We say that a process is **prefixed at** a variable x, if it is of the form $x!\langle v\rangle.P, x?(y).P, x \triangleleft \ell.P, x \triangleright \{\ell_i : P_i\}_{i \in I}$. We call a **redex** a process of the form $(x!\langle\tilde{v}\rangle.P \,|\, x?(\tilde{y}).Q)$, or of the form $(\nu\, xy)(x \triangleleft \ell_j.P \,|\, y \triangleright \{\ell_i : P_i\}_{i \in I})$ for $j \in I$.

Definition 10 (Well-Formedness) *A process is well-formed if for each of its structural congruent processes $(\nu\, \tilde{xy})(P \,|\, Q)$, the following conditions hold:*

- *if P is of the form* if v then P else Q, *then v is either* true *or* false, *and*
- *if P and Q are processes prefixed at the same variable, then they are of the same nature (input, output, branch, selection), and*
- *if P is prefixed in x_i and Q is prefixed in y_i where $x_i \in \tilde{x}$, $y_i \in \tilde{y}$, then $P|Q$ is a redex.*

A.4 Typing Rules

Typing judgements for values have the form $\Gamma \vdash v : T$, stating that a value v has type T in the typing context Γ, and for processes $\Gamma \vdash P$, stating that a process P is well typed in the typing context Γ.

Definition 11 (Typing Rules)

$$\frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}\ (\mathsf{TVar}) \qquad \frac{\mathsf{un}(\Gamma) \quad v = true/false}{\Gamma \vdash v : Bool}\ (\mathsf{TVal})$$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ (\mathsf{TInact}) \qquad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \,|\, Q}\ (\mathsf{TPar})$$

$$\frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu\, xy)P}\ (\mathsf{TRes}) \qquad \frac{\Gamma_1 \vdash v : Bool \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q}\ (\mathsf{TCond})$$

$$\frac{\Gamma, x : U, y : T \vdash P}{\Gamma, x : ?T.U \vdash x?(y).P}\ (\mathsf{TRcv}) \qquad \frac{\Gamma_1, x : U \vdash P \quad \Gamma_2 \vdash v : T}{\Gamma_1 \circ \Gamma_2, x : !T.U \vdash x!\langle v \rangle.P}\ (\mathsf{TSnd})$$

$$\frac{\Gamma, x : T_j \vdash P \quad j \in I}{\Gamma, x : \oplus\{\ell_i : T_i\}_{i \in I} \vdash x \triangleleft \ell_j.P}\ (\mathsf{TSel}) \qquad \frac{\Gamma, x : T_i \vdash P_i \quad i \in I}{\Gamma, x : \&\{\ell_i : T_i\}_{i \in I} \vdash x \triangleright \{\ell_i : P_i\}_{i \in I}}\ (\mathsf{TBr})$$

Rule (TVar) states that a variable x is of type T, if this is the type assumed in the typing context. Rule (TVal) states that a value v, being either true or false, is of type Bool. (TInact) states that the terminated process **0** is always well-typed. Notice that in all the previous rules, the typing context ÎŞ is an unrestricted one. (TPar) types the parallel composition of two processes, using the split operator for typing contexts $\circ$ which ensures that each linearly-typed channel x, is used linearly, i.e., in $P \,|\, Q$, x occurs either in P or in Q but never in both. Rule (TRes) states that $(\nu\, xy)P$ is well typed if P is well typed and the co-variables have dual types, namely T and $\overline{T}$. Rule (TCond) states that the conditional statement is well typed if its guard is typed by a boolean type and the branches are well typed under the same typing context. Rules (TRcv) and (TSnd) type the receiving and the sending of a value. Rule (TBr) types an external choice on channel x, checking that each branch continuation P_i follows the respective continuation type of x. Dually, rule (TSel) types an internal choice communicated on channel x, checking that the chosen label is among the ones offered by the receiver and that the continuation proceeds as expected by the type of x.

A.5 Main Results

Weakening allows introduction of new unrestricted channels in a typing con- text. It holds only for unrestricted channels, for linear ones it would be unsound, since when a linear channel is in a typing context, this means that it should be used in the process it types. The weakening lemma is useful when we need to relax the typing assumptions for a process and include new typing assumptions of variables not free in the process.

Lemma 2 (Unrestricted Weakening) *If* $\Gamma \vdash P$ *and* T *is not linear then* $\Gamma, x : T \vdash P$.

Strengthening is somehow the opposite operation of weakening, since it allows us to remove unrestricted channels from the typing context that are not free in the process being typed. This operation is mostly used after a context split is performed.

Lemma 3 (Strengthening for expressions) *If* $\Gamma, x : T \vdash v : U$ *and* $x \notin fv(v)$ *then* $\mathsf{un}(T)$ *and* $\Gamma \vdash v : U$.

Lemma 4 (Strengthening) *If* $\Gamma, x : T \vdash P$ *and* $x \notin fv(P)$ *then* $\mathsf{un}(T)$ *and* $\Gamma \vdash P$.

The substitution lemma that follows is important in proving the main results at the end of the section.

Lemma 5 (Substitution) *Let* $\Gamma_1 \vdash v : Z$ *and* $\Gamma_2, z : Z \vdash P$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$ *then* $\Gamma \vdash P[v/z]$.

Another important property is the following one, stating the type preservation of a process under structural congruence.

Lemma 6 (Type Preservation under ≡ for π-calculus with sessions) *Let* $\Gamma \vdash P$ *and* $P \equiv P'$, *then* $\Gamma \vdash P'$.

Theorem 1 (Type preservation) *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$ *then* $\Gamma \vdash Q$

Theorem 2 (Progress) *If* $\vdash P$, *then* P *is well-formed.*

We are ready now to present the main result of the session type system. The following theorem states that a well-typed closed process does not reduce to an ill-formed one.

Theorem 3 (Type safety) *If* $\vdash P$, *and* P *reduces to* Q *in zero or more steps, then* Q *is well formed.*

References

[1] A basis for concurrency and distribution. http://groups.inf.ed.ac.uk/abcd/.

[2] Links: Linking theory to practice for the web. http://groups.inf.ed.ac.uk/links/.

[3] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, December 2007.

[4] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton, NJ, USA, 2004. AAI3136691.

[5] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 161–177. Springer, 2015.

[6] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. A contract-oriented middleware. In *International Workshop on Formal Aspects of Component Software*, pages 86–104. Springer, 2015.

[7] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*

[8] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[9] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. ACM.

[10] Ornela Dardha, Simon J Gay, Dimitrios Kouzapas, Roly Perera, A Laura Voinea, and Florian Weber. Mungo and stmungo: Tools for typechecking protocols in java. *Behavioural Types: from Theory to Tools*, page 309, 2017.

[11] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In Gilles Barthe and Cédric Fournet, editors, *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[12] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. *PLDI, ACM SIGPLAN Notices*, 37(5):13–24, 2002.

[13] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[14] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM Press, 2008.

[15] Kohei Honda. Types for dyadic interaction. In *CONCUR '93*, volume 715 of *Springer LNCS*, pages 509–523, 1993.

[16] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98*, volume 1381 of *Springer LNCS*, pages 122–138, 1998.

[17] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint api generation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 401–418. Springer, 2016.

[18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP '16*, pages 146–159. ACM Press, 2016.

[19] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[20] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[21] Sam Lindley and J Garrett Morris. Embedding session types in haskell. In *Proceedings of the 9th International Symposium on Haskell*, pages 133–145. ACM, 2016.

[22] Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools*, page 265, 2017.

[23] Nicholas Ng and Nobuko Yoshida. High performance parallel design based on session programming. *MEng thesis, Department of Computing, Imperial College London*, 2010.

[24] Nicholas Ng, Nobuko Yoshida, Xin Yu Niu, and Kuen Hung Tsoi. Session types: towards safe and fast reconfigurable programming. *ACM SIGARCH Computer Architecture News*, 40(5):22–27, 2012.

[25] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *ACM SIGPLAN Notices*, volume 51, pages 568–581. ACM, 2016.

[26] Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27, 2017.

[27] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 539–558, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[28] Jorge A. PÃľrez, LuÃŋs Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254 – 302, 2014.

[29] Matthew Sackman and Susan Eisenbach. Session types in haskell: Updating message passing for the 21st century. 2008.

[30] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. *PARLE, Springer LNCS*, 817:398–413, 1994.

[31] A Laura Voinea and Simon J Gay. Benefits of session types for software development. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 26–29. ACM, 2016.

[32] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC '13*, volume 8358 of *Springer LNCS*, pages 22–41, 2013.