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 19, 2018

Contents

1 Introduction **2**

1.1 Thesis Statement 2

1.2 Publications 2

2 Literature review **3**

2.1 Mungo 3

3 Work Undertaken **4**

3.1 Modular linearity 4

3.1.1 Capabilities in π-calculus with sessions 5

4 Modular linearity **5**

4.0.1 A semantic approach to π-calculus with sessions 7

4.1 A Paxos implementation 7

4.2 Other Activities 11

5 Future Work **11**

5.1 Thesis Outline 11

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, 24, 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. Empirical studies to assess methodologies and tools are to be carried out, with results being used to improve language and tool design and implementation.

1.1 Thesis Statement

Session types are a useful addition to the syntax and semantics of modern languages.

Programming with session types and the constraints that this comes with i.e. linearity can be understood and used by real world programmers.

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, 24, 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[11]. They have been extended to ensure the progress property and deadlock freedom [8]. 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 [4], global session types have been used to detect choreographies that can be realized in the context of web services.

Session types have been successfully applied in functional programming languages [25], object-oriented languages like Java [17, 12], low-level programming languages like C in [23], dynamically-typed languages like Python [22] or Erlang [21], or in the operating systems context with the Sing# language [9].

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] 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.

Mungo uses the JastAdd framework[1] [13]. The JastAdd framework provides a Java parser which was used for the implementation of the Mungo typechecker. The JastAdd suite was also used to implement a parser for the Java-like typestate specification language.

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.

Our approach builds on work done to deal with memory management, explicit region deallocation in the Capability Calculus [6]; 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 [10]. The work presented below is inspired by the work done by FÃďhndrich and DeLine [10]

[1] http://jastadd.org/web/extendj/

3.1.1 Capabilities in π-calculus with sessions

4 Modular linearity

We present a simple type system based on *pi*-calculus with session types and enriched with capabilities. The formulation allows aliasing while ensuring through capabilities that all communication actions in a session type occur.

Ordinary types	T	$::= int \mid Bool$	
Session types	S	$::= !S.U$	output
		$\mid ?S.U$	input
		$\mid \oplus\{\ell_i : S_i\}_{i\in I}$	choice
		$\mid \&\{\ell_i : S_i\}_{i\in I}$	branch
		$\mid$ end	terminated session
Cabapilities	C	$::= . \mid \{\rho \to S\} \otimes C \mid \epsilon \otimes C$	
Types	τ	$::= tr(\rho)$	tracked type
		$\mid T$	ordinary types
		$\mid S$	session types
		$\mid C$	capabilities
Type Environments	Γ	$::=$	
	Δ	$::= T$	ordinary types

Figure 1: Syntax

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.

Typing judgments have the form $\Delta;\Gamma \vdash P;C$. Γ maps program variables to nonlinear types T only. Linearity is enforced via capabilities, not via environment splitting. Such a judgement means that the process P uses channels as specified by the types in ÎŞ. A process is either correctly typed or not; we do not assign types to processes.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2: Typing rules

Proofs for progress and type preservation are in progress. 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.

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.

4.0.1 A semantic approach to π-calculus with sessions

4.1 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[5] 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[2].

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

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

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

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:

$$\begin{array}{rcllllll} P & ::= & \mathbf{0} & \text{(Inaction)} & | & a!(\check{x}).P \ \text{(Request)} & | & a?(x).P \quad \text{(Accept)} \\ & | & k!\langle e\rangle.P & \text{(Send)} & | & k?(x).P \ \text{(Receive)} & | & k \triangleleft \ell.P \quad \text{(Selection)} \\ & | & k \triangleright \{\ell_i : P_i\}_{i\in I} & \text{(Branching)} & | & \text{if } \varphi \text{ then } P \text{ else } P & & \text{(Conditional)} \end{array}$$

Variable and name binders are standard, so is $\mathrm{fn}(P)$. (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 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).

We extend the $\mathrm{fn}(N)$ function to networks. 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. Session Type] Let $\mathcal{B}$ be a set of *base types* ranged over by β. Session types are inductively defined by the following grammar:

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

The duality operator also follows the standard binary session type definition [**?**].

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

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.

The following definitions are needed to define the typing system. Message Types $M ::= \varepsilon \mid {}_{!}.\beta M \mid {}_{\oplus}\ell.M$

Message types represent the sequence types of the values, ${}_{!}\beta.M$ or labels, ${}_{\oplus}\ell.M$ in a session buffer term. Typing Context] We define Γ, Δ, and Θ typing contexts: $\Gamma ::= \emptyset \mid \Gamma, a : T \mid \Gamma, x : \beta \qquad \Delta ::= ($
$\Theta ::= \emptyset \mid \Theta, \kappa : M \mid \Theta, s : c$

Shared context Γ tracks shared names used in a process, while context Δ tracks linear (session) names or sesion 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'.

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. We assume the extension of our calculus in Appendix **??** that also supports tuples of values as messages. For simplicity, the learner and acceptor roles are combined.

The Paxos communication interaction is described by session type PaxosType:

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

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:

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

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 $\overline{\text{PaxosType}}$.

During a round a Paxos agent may restart in which case it terminates its current sessions and proceeds to the initial Paxos network, $\text{Paxos}_{x,y}$. 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 $\textsf{Paxos}\langle x+1, y\rangle$. 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. $v_h = \max(\{v \,|\, (n,v) \in \{(n,v)_i\}_{i\in I}\})$. 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).

4.2 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[3] 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.

5 Future Work

5.1 Thesis Outline

1. Introduction

    1.1. Motivation and Objectives

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

1.2. Contributions
1.3. Publications

2. Background theory

2.1. Introduction

3. Modular linearity with Capabilities

3.1. Introduction

4. Modular linearity, a semantic approach

4.1. Introduction

5. Mungo and Paxos

5.1. Introduction

6. Conclusion

6.1. Summary of Thesis Achievements
6.2. Future Work

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, pages 177–190. ACM Press, 2006.

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

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

[12] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL '10*, pages 299–312. ACM Press, 2010.

[13] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Springer LNCS*, pages 166–200, 2011.

[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, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP '08*, volume 5142 of *Springer LNCS*, pages 516–541, 2008.

[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] Dimitris Mostrous and VascoT. Vasconcelos. Session typing for a featherweight erlang. In *Coordination Models and Languages*, volume 6721 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[22] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: local verification of global protocols. In *RV '13*, volume 8174 of *Springer LNCS*, pages 358–363, 2013.

[23] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *TOOLS '12*, pages 202–218, 2012.

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

[25] Vasco T. Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. In *CONCUR '04*, volume 3170 of *Springer LNCS*, pages 497–511. Springer, 2004.

[26] 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.