



University of Glasgow | School of
Computing Science

Honours Individual Project Dissertation

ENCODING SESSION TYPES INTO LINEAR TYPES IN π -CALCULUS

John Bell
March 27, 2019

Abstract

π -calculus, and the encoding from session types to linear types in π -calculus, can both be difficult to understand for people first learning of them, and the encoding can be hard to use even with experience. This project attempted to remedy this with a web app that would both teach new users about the concepts of pi calculus, and help those using the encoding by allowing them to encode types and processes automatically. The web app was found to be moderately helpful in teaching pi calculus, but with room for improvement.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: John Bell Date: 25 March 2019

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Aims	1
1.3	Outline	1
2	Background	3
2.1	π -Calculus	3
2.2	Session Types	3
2.2.1	Typechecking	5
2.2.2	Semantics	8
2.3	Linear Types	9
2.3.1	Typechecking	10
2.3.2	Semantics	11
2.4	Encoding	12
3	Design	15
3.1	Syntax	15
3.2	Typing Extensions	16
3.3	Web Interface	17
4	Implementation	23
4.1	Architecture	23
4.2	Web Interface	23
4.3	Encoding	24
4.4	Typechecking	26
4.5	Semantics	29
5	Evaluation	31
5.1	User Study	31
6	Conclusion	37
6.1	Related Works	37
6.2	Future Work	38
	Appendices	40
A	Appendices	40
A.1	User Study Survey	40
A.2	User Study Ethics Checklist	43
A.3	User Study Introduction Script	44
A.4	User Study Debriefing Script	45
	Bibliography	46

1 | Introduction

1.1 Motivation

π -Calculus, and the *encoding* from session types to linear types in π -calculus, are both useful tools in theoretical computing science. Due to how abstract they are, many people find them complicated and confusing. To mitigate this, we have created a teaching tool, allowing users to learn about, and get experience with, π -calculus, session types, linear types and the encoding. As well as theoretical uses, π -calculus also has practical applications, and in helping users understand the theoretical concepts, we also help them make use of these practical applications.

The tool was created not just to help people better understand π -calculus but also to act as a utility for those working with the encoding. The encoding of session-typed π -calculus into linear-typed π -calculus is cumbersome and complex, especially with larger processes. By creating a tool which can perform the encoding automatically, we can remove much of the effort of working with it.

As the encoding turns session-typed π -calculus into linear-typed π -calculus, having a tool to perform the encoding would also allow people to use tools designed for π -calculus with linear types on programs in π -calculus with session types. This would however require some additional development into creating an intermediate link between this tool and existing tools, due to differences between them.

1.2 Aims

The goal of this project was to create a tool which can be used both to teach π -calculus, session types, linear types and the encoding, and also to aid those already familiar with the concepts in using the encoding. As such, the primary aims for the tool were:

- Explanations of the concepts behind π -calculus, session types, linear types and the encoding.
- An environment to write code in both session- or linear-typed π -calculus.
- The ability to execute the π -calculus code you have written.
- The ability to typecheck the π -calculus code you have written.
- The ability to encode session-typed π -calculus programs you have written into linear-typed π -calculus.

For the purposes of teaching, the tool is intended to be used in conjunction with another means of learning π -calculus, but it can also be used on its own.

1.3 Outline

This work details the creation of a web app designed as a learning tool for π -calculus and the encoding from session types to linear types. This includes the decisions made in design to aid in

facilitating understanding of these concepts, and the technical development of the tool. It also considers how effective the tool will be in its purpose, and an analysis of the tool as a product. The remainder of this work is structured as follows:

Chapter 2 - Background This chapter describes the theory behind the app, detailing π -calculus, session and linear types, and the encoding from the former to the latter.

Chapter 3 - Design This chapter exhibits the thought behind the design decisions that went into the app and their intended effects on its effectiveness.

Chapter 4 - Implementation This chapter details the technologies and methods employed to create the app and the reasoning behind them.

Chapter 5 - Evaluation This chapter discusses an effort to measure how useful the app is in its intended purpose as a teaching tool.

Chapter 6 - Conclusion This chapter reflects on the app and how it compares to similar tools and how it may be further developed.

2 | Background

2.1 π -Calculus

π -Calculus is a process calculus, created by Milner et al. (1992), as an extension of CCS (Calculus of Concurrent Systems) by Milner (1980). *Process calculi* are methods of modelling concurrent computation, represented as processes passing messages between each other. What makes π -calculus different from other process calculi is that it allows those messages to be the channels on which communication occurs. This means that while other process calculi describe systems with a fixed network configuration, π -calculus can represent systems where it may change, e.g. one member of the network informing another member about the location of a third member, allowing those two to communicate. In this work, we consider two typing disciplines to π -calculus: Session Types, described in Section 2.2 and Linear Types, described in Section 2.3.

2.2 Session Types

Session types are a type system designed for process calculi used to add structure to communication. A session has a defined protocol, denoting an ordered set of interactions which communication must follow. (Honda et al. 1998) Session types also define two types and corresponding processes which are used to create choice in processes, branch and select. Session types can be applied to π -calculus to add structure to its communications. Session types also provide π -calculus with privacy to communication, as the session is known only to the processes communicating through it. In this work, we denote sessions through co-names as presented in Vasconcelos (2012). However, following the example of Dardha et al. (2017), we use co-names only for sessions. This allows us to more easily express the concept of duality of session types. Sessions specify the structure of the communication symmetrically for each endpoint of the session, for instance, when one endpoint is set to send a particular type, the other is set to receive that same type. This symmetry in the types of sessions is the duality of session types, where each session type is designated another as its dual. Figure 2.1 defines the syntax of session-typed π -calculus, and Figure 2.2 presents each session type's dual.

$T ::= S$	(Session Type)	$S ::= \text{end}$	(Termination)
$\#T$	(Standard Channel)	$?T.S$	(Receive)
Unit	(Unit Type)	$!T.S$	(Send)
\dots	(Other Constructs)	$\&\{l_i : S_i\}_{i \in I}$	(Branch)
		$\oplus \{l_i : S_i\}_{i \in I}$	(Select)
$P, Q ::= x! \langle v \rangle . P$	(Output)	$\mathbf{0}$	(Inaction)
$x?(w).P$	(Input)	$P \mid Q$	(Composition)
$x \triangleleft l_j . P$	(Selection)	$(\nu xy) P$	(Session Restriction)
$x \triangleright \{l_i : P_i\}$	(Branching)	$(\nu x) P$	(Channel Restriction)
$v ::= x$	(Name)	$*$	(Unit Value)

Figure 2.1: The standard syntax of session-typed π -calculus. Presented above the line is the syntax of types, and below the line processes and values. Input and Output are the two constituent parts of a basic communication; a value sent by an output is received by an input composed in parallel. Selection and Branching represent choice; Branching offers a range of possible ways to continue, and Selection chooses one of those. This figure has been adapted from Dardha et al. (2017).

$$\begin{aligned}
\overline{\text{end}} &\triangleq \text{end} \\
\overline{!T.S} &\triangleq ?T.\overline{S} \\
\overline{?T.S} &\triangleq !T.\overline{S} \\
\overline{\oplus \{l_i : S_i\}_{i \in I}} &\triangleq \&\{l_i : \overline{S_i}\}_{i \in I} \\
\overline{\&\{l_i : S_i\}_{i \in I}} &\triangleq \oplus \{l_i : \overline{S_i}\}_{i \in I}
\end{aligned}$$

Figure 2.2: Duality of session types. For a session endpoint of any particular type, the type of the other endpoint in the session is the dual of that type. This figure has been adapted from Dardha et al. (2017).

Throughout this work, we will illustrate various concepts by means of an example process that we will call the maths server. This process, presented in Figure 2.3, represents a server offering three mathematical services: addition of two integers, equality checking of two integers, or the negation of one integer; and a simple client, requesting the equality checking service on the integers 3 and 5. For this, we will consider integers and booleans to be predefined, as well as the operations used on them. We also present the types of the session endpoints used in the process. We will name these processes and types for ease of later reference.

$$\begin{aligned}
server &\triangleq x \triangleright \{plus : x?(v_1).x?(v_2).x!\langle v_1 + v_2 \rangle.0, \\
&\quad equal : x?(v_1).x?(v_2).x!\langle v_1 == v_2 \rangle.0, \\
&\quad neg : x?(v).x!\langle v_1 * -1 \rangle.0\} \\
client &\triangleq y \triangleleft equal.y!\langle 3 \rangle.y!\langle 5 \rangle.y?(eq).0 \\
\\
sv &\triangleq \&\{plus : ?Int.?Int.!Int.end, \\
&\quad equal : ?Int.?Int.!Bool.end, \\
&\quad neg : ?Int.!Int.end\} \\
cl &\triangleq \oplus\{plus : !Int.!Int.?Int.end, \\
&\quad equal : !Int.!Int.?Bool.end, \\
&\quad neg : !Int.?Int.end\} \\
\\
&\quad (v \ xy)(server \ | \ client)
\end{aligned}$$

Figure 2.3: The maths server process that will be used as an example throughout this work. For convenience, each part of the process has been given a name, as have the types of the session endpoints. Here, sv is the name of the type of x , and cl the type of y . This process has been adapted from Dardha et al. (2017).

2.2.1 Typechecking

We typecheck π -calculus code to ensure that the structure we have given to the code is correct. We do so by using a partial function from names to types known as a typing context, denoted as Γ . The typing context for a process should contain the types of all the variables in that process. A typing context containing the type of a variable is denoted $\Gamma \vdash v : T$, and a process being well-typed under a context is denoted $\Gamma \vdash P$. To handle the linearity of session-types, we have predicates for linear (lin) and unrestricted (un) types and contexts (Vasconcelos 2012), and an operator \circ known as the context split. These are defined in Figure 2.4.

$$\begin{aligned}
\text{lin}(T) &\quad \text{if } T \text{ is a session type and } T \neq \text{end} \\
\text{un}(T) &\quad \text{otherwise} \\
\text{lin}(\Gamma) &\quad \text{if } \Gamma \vdash x : T \text{ where } \text{lin}(T) \\
\text{un}(\Gamma) &\quad \text{otherwise} \\
\\
\frac{}{\emptyset = \emptyset \circ \emptyset} &\qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \text{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)} \\
\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \text{lin}(S)}{\Gamma, x : S = (\Gamma_1, x : S) \circ \Gamma_2} &\qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad \text{lin}(S)}{\Gamma, x : S = \Gamma_1 \circ (\Gamma_2, x : S)}
\end{aligned}$$

Figure 2.4: The rules for the lin and un predicates and the context split operator. These are used to ensure the privacy of session types, as the rules mean that each endpoint can appear in only one of the split contexts. This figure has been adapted from Dardha et al. (2017).

To typecheck a process or value under a particular context, typing rules are applied. These typing rules have the structure that if some premise is true then the conclusion, that some process

or value is well-typed under this context, is true. The premise of a typing rule typically contains the conclusion of another typing rule. Typing rules are applied to the premise of the previous rule, until all rules reach premises that are otherwise proven, typically that the typing context in this rule is unrestricted. The typing rules for session-typed π -calculus are presented in Figure 2.5. Figure 2.6 demonstrates how these rules are used, in a full typechecking proof of the maths server process.

$$\begin{array}{c}
\text{(T-Var)} \\
\frac{\text{un}(\Gamma)}{\Gamma, x : T \vdash x : T} \\
\\
\text{(T-Val)} \\
\frac{\text{un}(\Gamma)}{\Gamma \vdash * : \text{Unit}} \\
\\
\text{(T-Inact)} \\
\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \\
\\
\text{(T-Par)} \\
\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \\
\\
\text{(T-Res)} \\
\frac{\Gamma, x : S, y : \bar{S} \vdash P}{\Gamma \vdash (v xy) P} \\
\\
\text{(T-StdRes)} \\
\frac{\Gamma, x : T \vdash P \quad T \text{ is not an S}}{\Gamma \vdash (v x) P} \\
\\
\text{(T-In)} \\
\frac{\Gamma_1 \vdash x : ?T.S \quad \Gamma_2, x : S, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(w).P} \\
\\
\text{(T-StdIn)} \\
\frac{\Gamma_1 \vdash x : \#T \quad \Gamma_2, x : \#T, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?(w).P} \\
\\
\text{(T-Out)} \\
\frac{\Gamma_1 \vdash x : !T.S \quad \Gamma_2 \vdash v : T \quad \Gamma_3, x : S \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v \rangle.P} \\
\\
\text{(T-StdOut)} \\
\frac{\Gamma_1 \vdash x : \#T \quad \Gamma_2 \vdash v : T \quad \Gamma_3, x : \#T \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v \rangle.P} \\
\\
\text{(T-Sel)} \\
\frac{\Gamma_1 \vdash x : \oplus\{l_i : T_i\}_{i \in I} \quad \Gamma_2, x : T_j \vdash P \quad \exists j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P} \\
\\
\text{(T-Brch)} \\
\frac{\Gamma_1 \vdash x : \&\{l_i : T_i\}_{i \in I} \quad \Gamma_2, x : T_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}}
\end{array}$$

Figure 2.5: The rules used to typecheck session-typed π -calculus. The typing rules for processes all follow a rough pattern that the current typing context is split into contexts checking that the current operation is well-typed, and a context augmented with the results of the current operation checking that the continuation process is well-typed. This figure has been adapted from Dardha et al. (2017).

$$\begin{array}{c}
\frac{\frac{\frac{\frac{\frac{\frac{\text{(I)} \quad \text{(II)} \quad \text{(III)} \quad \text{(IV)} \quad \text{(V)} \quad \text{(VI)}}{\Gamma_1 \vdash x \triangleright \{plus : P_1, equal : P_2, neg : P_3\}}}{\Gamma_2 \vdash y \triangleleft equal.Q}}{\Gamma, x : sv, y : cl \vdash (server \mid client)}}{\Gamma \vdash (vxy) server \mid client}}{\text{(I)}} \\
\frac{\text{un}(\Gamma_{1a} \setminus x : \&\{plus : ?I.?I.!I.end, equal : ?I.?I.!B.end, neg : ?I.!I.end\})}{\Gamma_{1a} \vdash x : \&\{plus : ?I.?I.!I.end, equal : ?I.?I.!B.end, neg : ?I.!I.end\}} \text{(I)} \\
\frac{\frac{\frac{\frac{\frac{\text{un}(\Gamma_{1c} \setminus x : \dots)}{\Gamma_{1c} \vdash x : \dots}}{\Gamma_{1e} \vdash x : \dots}}{\Gamma_{1g} \vdash x : \dots}}{\Gamma_{1f}, x : !I.end, v_2 : I \vdash x! \langle v_1 + v_2 \rangle.0}}{\Gamma_{1d}, x : ?I.!I.end, v_1 : I \vdash x?(v_2).x! \langle v_1 + v_2 \rangle.0}}{\Gamma_{1b}, x : ?I.?I.!I.end \vdash x?(v_1).x?(v_2).x! \langle v_1 + v_2 \rangle.0}} \text{(II)} \\
\frac{\frac{\frac{\frac{\frac{\frac{\text{un}(\Gamma_{1i} \setminus x : \dots)}{\Gamma_{1i} \vdash x : \dots}}{\Gamma_{1k} \vdash x : \dots}}{\Gamma_{1m} \vdash x : \dots}}{\Gamma_{1l}, x : !B.end, v_2 : I \vdash x! \langle v_1 == v_2 \rangle.0}}{\Gamma_{1j}, x : ?I.!B.end, v_1 : I \vdash x?(v_2).x! \langle v_1 == v_2 \rangle.0}}{\Gamma_{1b}, x : ?I.?I.!B.end \vdash x?(v_1).x?(v_2).x! \langle v_1 == v_2 \rangle.0}} \text{(III)} \\
\frac{\frac{\frac{\frac{\frac{\frac{\text{un}(\Gamma_{1o} \setminus x : \dots)}{\Gamma_{1o} \vdash x : \dots}}{\Gamma_{1q} \vdash x : \dots}}{\Gamma_{1p}, x : !I.end, v_1 : I \vdash x! \langle v * -1 \rangle.0}}{\Gamma_{1r}, x : end \vdash 0}}{\Gamma_{1b}, x : ?I.!I.end \vdash x?(v).x! \langle v * -1 \rangle.0}} \text{(IV)} \\
\frac{\text{un}(\Gamma_{2a} \setminus y : \oplus\{plus : !I.!I.?I.end, equal : !I.!I.?B.end, neg : !I.?I.end\})}{\Gamma_{2a} \vdash y : \oplus\{plus : !I.!I.?I.end, equal : !I.!I.?B.end, neg : !I.?I.end\}} \text{(V)} \\
\frac{\frac{\frac{\frac{\frac{\frac{\frac{\text{un}(\Gamma_{2c} \setminus y : \dots)}{\Gamma_{2c} \vdash y : \dots}}{3 : I}}{\Gamma_{2e} \vdash y : \dots}}{\Gamma_{2g} \vdash y : \dots}}{\Gamma_{2f}, y : ?B.end \vdash y?(eq).0}}{\Gamma_{2d}, y : !I.?B.end \vdash y! \langle 5 \rangle.y?(eq).0}}{\Gamma_{2b}, y : !I.!I.?B.end \vdash y! \langle 3 \rangle.y! \langle 5 \rangle.y?(eq).0}} \text{(VI)} \\
\Gamma = \Gamma_1 \circ \Gamma_2, \quad \Gamma_1 = \Gamma_{1a} \circ \Gamma_{1b}, \quad \Gamma_2 = \Gamma_{2a} \circ \Gamma_{2b} \\
\Gamma_{1b}, x : ?I.?I.!I.end = \Gamma_{1c} \circ \Gamma_{1d}, \quad \Gamma_{1d}, x : ?I.!I.end, v_1 : I = \Gamma_{1e} \circ \Gamma_{1f}, \\
\Gamma_{1f}, x : !I.end, v_2 : I = \Gamma_{1g} \circ \Gamma_{1h} \\
\Gamma_{1b}, x : ?I.?I.!B.end = \Gamma_{1i} \circ \Gamma_{1j}, \quad \Gamma_{1j}, x : ?I.!B.end, v_1 : I = \Gamma_{1k} \circ \Gamma_{1l}, \\
\Gamma_{1l}, x : !B.end, v_2 : I = \Gamma_{1m} \circ \Gamma_{1n} \\
\Gamma_{1b}, x : ?I.!I.end = \Gamma_{1o} \circ \Gamma_{1p}, \quad \Gamma_{1p}, x : !I.end, v_1 : I = \Gamma_{1q} \circ \Gamma_{1r} \\
\Gamma_{2b}, y : !I.!I.?B.end = \Gamma_{2c} \circ \Gamma_{2d}, \quad \Gamma_{2d}, y : !I.?B.end = \Gamma_{2e} \circ \Gamma_{2f}, \\
\Gamma_{2f}, y : ?B.end = \Gamma_{2g} \circ \Gamma_{2h}
\end{array}$$

Figure 2.6: A full type derivation for the maths server process. Various parts are presented separately, and types `Int` and `Bool` are shortened to `I` and `B`, for spacing. Typing judgements of literal values and expressions are considered self-evident here and do not need to be proven with a typing rule.

2.2.2 Semantics

The operational semantics of π -calculus is represented as a binary relation called *reduction*. Each reduction can be thought of as a single step of execution, and typically replaces processes with their continuations. Execution finishes successfully when all processes are reduced to the termination process, or fails when no further reductions can be made while some processes are not terminated. As most reductions represent a communication between two processes, parallel composition is essential to reduction. Reduction typically results in the replacement of bounded variables. The reduction rules for session-typed π -calculus are presented in Figure 2.7. The reduction rule (R-Struct) uses structural congruence between processes. The structural congruence relation is defined as the smallest congruence relation satisfying the axioms presented in Figure 2.8. Figure 2.9 shows how the reduction rules are applied to the maths server process.

$$\begin{array}{ll}
(v\ xy)(x!\langle v\rangle.P \mid y?(w).Q) \rightarrow (v\ xy)(P \mid Q[v/z]) & \text{(R-Com)} \\
x!\langle v\rangle.P \mid x?(w).Q \rightarrow P \mid Q[v/w] & \text{(R-StndCom)} \\
(v\ xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i \in I}) \rightarrow (v\ xy)(P \mid P_j) & \text{(R-Case)} \\
P \rightarrow Q \implies (v\ x)P \rightarrow (v\ x)Q & \text{(R-StndRes)} \\
P \rightarrow Q \implies (v\ xy)P \rightarrow (v\ xy)Q & \text{(R-Res)} \\
P \rightarrow Q \implies P \mid R \rightarrow Q \mid R & \text{(R-Par)} \\
P \equiv P', P \rightarrow Q, Q' \equiv Q \implies P' \rightarrow Q' & \text{(R-Struct)}
\end{array}$$

Figure 2.7: A list of the reduction rules used to execute processes in session-typed π -calculus. $[v/w]$ represents the replacement of w with v . The first three of these rules are the reductions themselves, and the others are properties of the reductions. R-Com and R-Case can only occur under a session restriction to ensure the privacy of sessions. This figure has been adapted from Dardha et al. (2017).

$$\begin{array}{l}
P \mid Q \equiv Q \mid P \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
P \mid \mathbf{0} \equiv P \\
(v\ xy)\mathbf{0} \equiv \mathbf{0} \\
(v\ x)\mathbf{0} \equiv \mathbf{0} \\
(v\ xy)(v\ vw)P \equiv (v\ vw)(v\ xy)P \\
(v\ x)P \mid Q \equiv (v\ x)(P \mid Q) \\
(v\ xy)P \mid Q \equiv (v\ xy)(P \mid Q)
\end{array}$$

Figure 2.8: The axioms which the structural congruence relation must satisfy. These state the commutativity and associativity of parallel composition and that the inaction is the neutral element of parallel composition, as well as that restrictions on an inaction are unnecessary, order of restrictions does not matter, and restrictions can be extended to include other processes. The two axioms stating the extension of restrictions are only true when this extension does not cause the capture of names, i.e. x and y are not free names in Q . This figure has been adapted from Dardha et al. (2017).

$$\begin{aligned}
(v xy) (server \mid client) &\triangleq (v xy) (x \triangleright \{ \dots equal : P_e, \dots \} \mid y \triangleleft equal.y! \langle 3 \rangle.y! \langle 5 \rangle.y?(eq).0) \\
&\rightarrow (v xy) (x?(v_1).x?(v_2).x! \langle v_1 == v_2 \rangle.0 \mid y! \langle 3 \rangle.y! \langle 5 \rangle.y?(eq).0) \\
&\rightarrow (v xy) (x?(v_2).x! \langle 3 == v_2 \rangle.0[3/v_1] \mid y! \langle 5 \rangle.y?(eq).0) \\
&\rightarrow (v xy) (x! \langle 3 == 5 \rangle.0[3/v_1][5/v_2] \mid y?(eq).0) \\
&\rightarrow (v xy) (0[3/v_1][5/v_2] \mid 0[False/eq]) \\
&\equiv 0
\end{aligned}$$

Figure 2.9: The reduction rules being applied to the example process to show how it is executed. The rules being applied are, in order, R-Case, R-Com, R-Com and R-Com. Structural congruence is then used to simplify the fully reduced process to a single inaction.

2.3 Linear Types

Linear types are an extension of π -calculus, created by Kobayashi et al. (1999). They extend the idea of polarised channels (Odersky 1995; Pierce and Sangiorgi 1996), channels which can only communicate in one direction, by adding the limitation that they can only do such a communication once, after which they cannot be used in any form of communication. This can be used to preserve privacy of communication and prevent interference from other processes. We also use the variant type (Sangiorgi 1998), a type which allows a value to be one of multiple different type options. This is used alongside the Case process to add choice in processes. Figure 2.10 presents the syntax of linear-typed π -calculus, and Figure 2.11 describes the duality of linear types.

$\tau ::= l_o[\tilde{\tau}]$	(Linear Output)	$\#[\tilde{\tau}]$	(Standard Connection)
$l_i[\tilde{\tau}]$	(Linear Input)	$\langle l_i _ \tau_i \rangle_{i \in I}$	(Variant Type)
$l_{\#}[\tilde{\tau}]$	(Unit Type)	Unit	(Unit Type)
$\emptyset[]$	(Other Constructs)	...	(Other Constructs)

$P, Q ::= x! \langle \tilde{v} \rangle.P$	(Output)	0	(Inaction)
$x?(\tilde{w}).P$	(Input)	$P \mid Q$	(Composition)
$(v x)P$	(Restriction)	$\text{case } v \text{ of } \{ l_i _ (x_i) \triangleright P_i \}_{i \in I}$	(Case)

$v ::= x$	(Name)	*	(Unit Value)
l_v	(Variant Value)		

Figure 2.10: The standard syntax of linear-typed π -calculus. Presented above the line is the syntax of types, below the line processes and values. As in session types, Input and Output represent basic communication, but here, choice is handled through the Case. Unlike Branching and Selection, Case handles choice within a single process; there does not need to be two processes composed in parallel for a choice to be made through Case. $\tilde{\tau}$ and \tilde{v} are used to denote a sequence of types and a sequence of values respectively. This figure has been adapted from Dardha et al. (2017).

$$\begin{aligned}\overline{l_i[\tilde{\tau}]} &\triangleq l_o[\tilde{\tau}] \\ \overline{l_o[\tilde{\tau}]} &\triangleq l_i[\tilde{\tau}] \\ \overline{\emptyset[]} &\triangleq \emptyset[]\end{aligned}$$

Figure 2.11: Duality of linear types. A process possessing one capability of a linear channel can only communicate with a process possessing the dual of that capability of the same linear channel. This figure has been adapted from Dardha et al. (2017).

2.3.1 Typechecking

Typechecking in linear types is similar to in session types. The lin and un predicates are still used with largely similar definitions, but instead of context split, there is instead a context combination operator \uplus . This operator and the rules for lin and un in linear types are defined in Figure 2.12. The typing rules for linear-typed π -calculus are presented in Figure 2.13.

$$\begin{aligned}\text{lin}(\tau) &\text{ if } \tau = l_\alpha[\tilde{\tau}] \text{ or } (\tau = \langle l_{i_j}\tau_i \rangle_{i \in I} \text{ and for some } j \in I, \text{lin}(\tau_j)) \\ \text{un}(\tau) &\text{ otherwise} \\ \text{lin}(\Gamma) &\text{ if } \Gamma \vdash x : \tau \text{ where } \text{lin}(\tau) \\ \text{un}(\Gamma) &\text{ otherwise}\end{aligned}$$

$$\begin{aligned}l_i[\tilde{\tau}] \uplus l_o[\tilde{\tau}] &\triangleq l_\#[\tilde{\tau}] \\ \tau \uplus \tau &\triangleq \tau && \text{if } \text{un}(\tau) \\ \tau \uplus \tau' &\triangleq \text{undef} && \text{otherwise}\end{aligned}$$

$$(\Gamma_1 \uplus \Gamma_2)(x) \triangleq \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2 \text{ are defined} \\ \Gamma_1(x) & \text{if } \Gamma_1(x), \text{ but not } \Gamma_2 \text{ is defined} \\ \Gamma_2(x) & \text{if } \Gamma_2(x), \text{ but not } \Gamma_1 \text{ is defined} \\ \text{undef} & \text{otherwise} \end{cases}$$

Figure 2.12: The rules for the lin and un predicates and the context combination operator. These are used to ensure the privacy of linear channels. This figure has been adapted from Dardha et al. (2017).

$$\begin{array}{c}
\text{(T}\pi\text{-Var)} \\
\frac{\text{un}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \\
\\
\text{(T}\pi\text{-Val)} \\
\frac{\text{un}(\Gamma)}{\Gamma \vdash * : \text{Unit}} \\
\\
\text{(T}\pi\text{-LVal)} \\
\frac{\Gamma \vdash v : \tau_j \quad j \in I}{\Gamma \vdash l_{j_} v : \langle l_{i_} \tau_i \rangle_{i \in I}} \\
\\
\text{(T}\pi\text{-Inact)} \quad \text{(T}\pi\text{-Par)} \quad \text{(T}\pi\text{-Res1)} \quad \text{(T}\pi\text{-Res2)} \\
\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \quad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q} \quad \frac{\Gamma, x : l_{\#}[\tilde{\tau}] \vdash P}{\Gamma \vdash (v x) P} \quad \frac{\Gamma, x : \text{empty}[] \vdash P}{\Gamma \vdash (v x) P} \\
\\
\text{(T}\pi\text{-Inp)} \quad \text{(T}\pi\text{-StdInp)} \\
\frac{\Gamma_1 \vdash x : l_i[\tilde{\tau}] \quad \Gamma_2, \tilde{y} : \tilde{\tau} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x?(\tilde{w}). P} \quad \frac{\Gamma_1 \vdash x : \#[\tilde{\tau}] \quad \Gamma_2, x : \#[\tilde{\tau}], \tilde{y} : \tilde{\tau} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x?(\tilde{w}). P} \\
\\
\text{(T}\pi\text{-Out)} \quad \text{(T}\pi\text{-StdOut)} \\
\frac{\Gamma_1 \vdash x : l_o[\tilde{\tau}] \quad \tilde{\Gamma}_2 \vdash \tilde{v} : \tilde{\tau} \quad \Gamma_3 \vdash P}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash x! \langle \tilde{v} \rangle . P} \quad \frac{\Gamma_1 \vdash x : \#[\tilde{\tau}] \quad \tilde{\Gamma}_2 \vdash \tilde{v} : \tilde{\tau} \quad \Gamma_3, x : \#[\tilde{\tau}] \vdash P}{\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash x! \langle \tilde{v} \rangle . P} \\
\\
\text{(T}\pi\text{-Case)} \\
\frac{\Gamma_1 \vdash v : \langle l_{i_} \tau_i \rangle_{i \in I} \quad \Gamma_2, x_i : \tau_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \uplus \Gamma_2 \vdash \text{case } v \text{ of } \{ l_{i_} (x_i) \triangleright P_i \}_{i \in I}}
\end{array}$$

Figure 2.13: A list of the rules used to typecheck processes in linear-typed π -calculus. Most of these are similar to those in session-typed π -calculus, with the exception of $\text{T}\pi\text{-LVal}$ and $\text{T}\pi\text{-Case}$. This figure has been adapted from Dardha et al. (2017).

2.3.2 Semantics

The operational semantics of linear-typed π -calculus is similar to those of session-typed π -calculus. The reduction rules for linear-typed π -calculus are presented in Figure 2.14. A notable difference is that rule $(\text{R}\pi\text{-Case})$ is a reduction containing no parallel composition of processes, instead reducing only a single process, something which does not occur in session-typed π -calculus.

$$\begin{array}{l}
x! \langle \tilde{v} \rangle . P \mid x?(\tilde{w}) . Q \rightarrow P \mid Q[\tilde{v}/\tilde{w}] \quad (\text{R}\pi\text{-Com}) \\
\text{case } l_{j_} v \text{ of } \{ l_{i_} (x_i) \triangleright P_i \}_{i \in I} \rightarrow P_j[v/x_j] \quad j \in I \quad (\text{R}\pi\text{-Case}) \\
P \rightarrow Q \implies (v x) P \rightarrow (v x) Q \quad (\text{R}\pi\text{-Res}) \\
P \rightarrow Q \implies P \mid R \rightarrow Q \mid R \quad (\text{R}\pi\text{-Par}) \\
P \equiv P', P \rightarrow Q, Q' \equiv Q \implies P' \rightarrow Q' \quad (\text{R}\pi\text{-Struct})
\end{array}$$

Figure 2.14: A list of the reduction rules used to execute processes in linear-typed π -calculus. $[v/w]$ represents the replacement of w with v . The first two of these rules are the reductions themselves, and the others are properties of the reductions. This figure has been adapted from Dardha et al. (2017).

2.4 Encoding

The encoding from session-typed π -calculus to linear-typed π -calculus was created by Dardha et al. (2012; 2017), and uses the *continuation-passing principle*. While communication can occur over a pair of session endpoints multiple times, communication over a linear channel can occur only once. To get around this, alongside every communication between processes, the encoding creates a new linear channel and sends one of the capabilities of this new channel alongside the original payload. This gives each of the communicating processes access to the new channel, on which communication continues. The encoding uses a renaming function, f , to replace the session co-names in processes with the continuation channel that would be in use at that point of the process. The rules for encoding session-types and session-typed processes are presented in Figure 2.15.

$\llbracket \text{end} \rrbracket \triangleq \emptyset []$	(E-End)
$\llbracket ?T.S \rrbracket \triangleq l_i[\llbracket T \rrbracket, \llbracket S \rrbracket]$	(E-Inp)
$\llbracket !T.S \rrbracket \triangleq l_o[\llbracket T \rrbracket, \llbracket \bar{S} \rrbracket]$	(E-Out)
$\llbracket \&\{l_i : S_i\}_{i \in I} \rrbracket \triangleq l_i[\langle l_{i-} \llbracket S \rrbracket \rangle_{i \in I}]$	(E-Branch)
$\llbracket \oplus\{l_i : S_i\}_{i \in I} \rrbracket \triangleq l_o[\langle l_{i-} \llbracket \bar{S} \rrbracket \rangle_{i \in I}]$	(E-Select)
$\llbracket x \rrbracket_f \triangleq f_x$	(E-Name)
$\llbracket * \rrbracket_f \triangleq *$	(E-Star)
$\llbracket 0 \rrbracket_f \triangleq 0$	(E-Inaction)
$\llbracket P \mid Q \rrbracket_f \triangleq \llbracket P \rrbracket_f \mid \llbracket Q \rrbracket_f$	(E-Composition)
$\llbracket (v \ x y) P \rrbracket_f \triangleq (v \ c) \llbracket P \rrbracket_{f, \{x, y \rightarrow c\}}$	(E-Restriction)
$\llbracket (v \ z) P \rrbracket_f \triangleq (v \ z) \llbracket P \rrbracket_f$	(E-StdRestriction)
$\llbracket x! \langle v \rangle . P \rrbracket_f \triangleq (v \ c) f_x! \langle \llbracket v \rrbracket_f, c \rangle . \llbracket P \rrbracket_{f, \{x \rightarrow c\}}$	(E-Output)
$\llbracket x?(w) . P \rrbracket_f \triangleq f_x?(w, c) . \llbracket P \rrbracket_{f, \{x \rightarrow c\}}$	(E-Input)
$\llbracket x \triangleleft l_j . P \rrbracket_f \triangleq (v \ c) f_x! \langle l_{j-} c \rangle . \llbracket P \rrbracket_{f, \{x \rightarrow c\}}$	(E-Selection)
$\llbracket x \triangleright \{l_i : P_i\}_{i \in I} \rrbracket_f \triangleq f_x?(y) . \text{case } y \text{ of } \{l_{i-}(c) > \llbracket P_i \rrbracket_{f, \{x \rightarrow c\}}\}_{i \in I}$	(E-Branching)

Figure 2.15: A list of the rules used to encode session-typed π -calculus into linear-typed π -calculus. f is the encoding function, a renaming function used to replace session endpoints with continuation channels. This figure has been adapted from Dardha et al. (2017).

The encoding of the session type end is simply a channel with no capabilities, as no communication can occur on either of those types. The encoding of input and output are similar to each other: The type is encoded into a linear channel corresponding to its action (input or output) whose two payloads are the encoding of the original payload and the encoding of the continuation type. However, in output, the dual of the continuation type is encoded instead. This is because the linear channel must send the capability that the recipient process will be using, not the capability that the sending process will be using.

To illustrate, the type $! \text{Int} . ! \text{Int} . \text{end}$ would become $l_o[\text{Int}, l_i[\text{Int}, \emptyset []]]$. Meanwhile, the dual of this type, $? \text{Int} . ? \text{Int} . \text{end}$ becomes $l_i[\text{Int}, l_i[\text{Int}, \emptyset []]]$. As the sending process sends

the first integer, the process which receives this integer must now be prepared to receive another, thus the continuation channel that the sending process sends must be capable of receiving an integer. This means that the duality of session types, when encoded, becomes duality in the capability of only the outermost linear channel.

The encoding of branch and select are that they become a linear input and linear output channel respectively, whose payload is a variant type containing the continuation types of the branch and select. The duals of the continuation types of select are used for the same reason as in output.

Session restriction is encoded into a channel restriction containing a linear connection, with the encoding function for the continuation process updated such that both of the session co-names are replaced with the name of the linear connection. The idea here is that each endpoint is replaced with one of the capabilities of the connection.

The encoding of output processes is the core of the continuation-passing principle. It inserts a channel restriction in front of the output process, containing a linear connection. This linear connection is the continuation channel. One of its capabilities is sent along the encoding function's current replacement for the session endpoint and the other is kept for the continuation process. The encoding function is updated to replace the session endpoint with the new continuation channel. The encoding of input processes is simpler: it adds to the input process another payload, the continuation channel, which the encoding function is updated with as the replacement for the session endpoint.

The encoding of selection replaces the select process with a channel restriction, creating the continuation channel, and an output process sending a variant value, comprised of the label being selected and the continuation channel. The encoding function is updated as always. The encoding of branching replaces the branch process with a receive process which receives a variant value, and a case process using the received variant value as its choice variable. The encoding function is updated with the continuation channel, embedded in the variant value, to replace the session endpoint and this updated encoding function is used to encode each of the continuation processes.

To demonstrate how these encoding rules work in practice, Figure 2.16 presents the encoding being used to produce a linear-typed version of the maths server process.

$$\begin{aligned}
\llbracket (v\ xy)(server \mid client) \rrbracket_{\emptyset} &= (v\ c) \llbracket (server \mid client) \rrbracket_{\{x, y \mapsto c\}} \\
&= (v\ c) (\llbracket server \rrbracket_{\{x \mapsto c\}} \mid \llbracket client \rrbracket_{\{y \mapsto c\}}) \\
\llbracket server \rrbracket_{\{x \mapsto c\}} &= c?(s).\text{casesof}\{ \\
&\quad plus_c' \triangleright c'?(v_1, c'').c''?(v_2, c''').(v\ c''')c''! \langle v_1 + v_2, c'''' \rangle.0, \\
&\quad equal_c' \triangleright c'?(v_1, c'').c''?(v_2, c''').(v\ c''')c''! \langle v_1 == v_2, c'''' \rangle.0, \\
&\quad neg_c' \triangleright c'?(v_1, c'').(v\ c''')c''! \langle v_1 + v_2, c'''' \rangle.0, \\
\llbracket client \rrbracket_{\{y \mapsto c\}} &= (v\ c')c! \langle equal_c' \rangle.(v\ c'')c'! \langle 3, c'' \rangle.(v\ c''')c''! \langle 5, c'''' \rangle.c''''?(eq, c''''').0 \\
\llbracket sv \rrbracket &= l_i[\langle plus_l_i[\text{Int}, l_i[\text{Int}, l_o[\text{Int}, \emptyset[]]], \\
&\quad equal_l_i[\text{Int}, l_i[\text{Int}, l_o[\text{Bool}, \emptyset[]]], \\
&\quad neg_l_i[\text{Int}, l_o[\text{Bool}, \emptyset[]]] \rangle] \\
\llbracket cl \rrbracket &= l_o[\langle plus_l_i[\text{Int}, l_i[\text{Int}, l_o[\text{Int}, \emptyset[]]], \\
&\quad equal_l_i[\text{Int}, l_i[\text{Int}, l_o[\text{Bool}, \emptyset[]]], \\
&\quad neg_l_i[\text{Int}, l_o[\text{Bool}, \emptyset[]]] \rangle]
\end{aligned}$$

Figure 2.16: The encoding of the maths server process. This demonstrates how the continuation-passing principle works. For each communication, one process creates a channel and sends it alongside the original payload, and both processes use that channel for the next communication. This figure has been adapted from Dardha et al. (2017).

The encoding is useful because it eliminates the need for various proofs. As the encoding allows session-typed π -calculus to be expressed in terms of standard π -calculus constructs, properties which have already been proven for these constructs can now be applied to session-typed π -calculus without proving them again separately. Additionally, types are no longer separated into two syntactic categories, as they were in session-typed π -calculus, as shown in Figure 2.1. Having two syntactic categories produced the need for separate proofs for each category, which the encoding also eliminates. This gives us the benefits of session-typed π -calculus for substantially less theoretical complexity than if the encoding did not exist.

However, while the theoretical complexity is reduced, the practical complexity increases. Performing the encoding of a process by hand is difficult and cumbersome. And as the original process becomes larger, its encoding also grows at a higher rate, due to the inserted channel restrictions, making the encoding even more arduous. As mentioned previously in Section 1.1, this is part of the reasoning behind our project. By creating a tool which can perform the encoding of a process automatically, we can remove this practical complexity as well.

3 | Design

3.1 Syntax

In the web app's syntax, π -calculus code is structured as an optional list of declarations followed by a process. These declarations are mostly type declarations of variables but can also contain variable assignments, process naming and type naming. Process and type naming were included to help simplify code, allowing complicated processes or types to be written only once and referred back to when needed.

The web app's π -calculus syntax differs in some ways from the standard π -calculus syntax. The most obvious difference is the use of English words in processes. Where the standard syntax would use symbols such as $!$ and $?$, the web app instead uses the words **send** and **receive**. Types still use the same symbols as in the standard syntax, or the closest approximation available on standard keyboards. This change was made to processes to try to make it more readable, while types were kept the same to distinguish them from processes at a glance, and also to add a link back to the standard syntax. Figure 3.1 shows all of these such differences.

$$\begin{aligned}
 x?(w).P &\mapsto \text{receive}(x, w : T).P \\
 x!\langle v \rangle.P &\mapsto \text{send}(x, v).P \\
 x \triangleleft \{l_i : P_i\}_{i \in I} &\mapsto \text{branch}(x)\{l_i : P_i\}_{i \in I} \\
 x \triangleright l_j.P &\mapsto \text{select}(x, l_j).P \\
 (v x)P &\mapsto (\text{new } x : T) (P) \\
 0 &\mapsto \text{stop} \\
 \\
 \oplus \{l_i : T_i\}_{i \in I} &\mapsto +\{l_i : T_i\}_{i \in I} \\
 l_i[\tilde{\tau}] &\mapsto \text{li}[\tilde{\tau}] \\
 l_o[\tilde{\tau}] &\mapsto \text{lo}[\tilde{\tau}] \\
 l_{\#}[\tilde{\tau}] &\mapsto \text{l\#}[\tilde{\tau}] \\
 \emptyset[] &\mapsto \text{empty}[]
 \end{aligned}$$

Figure 3.1: A list of all the changes made for the web app's syntax. These changes were made in the hopes of being easier to understand for beginners. $\emptyset[]$ was initially changed to $/[]$, but this was found to be hard to read, particularly at the end of a highly-nested type, so **empty** $[]$ was used instead to be more readable.

As can be seen in the modified syntax for input processes and restrictions, the other main difference is the presence of type annotations. Type annotations are used to indicate the type of bounded variables. For example, in the standard π -calculus, an input process would resemble $x?(v).P$, which gives no information on what v is expected to be. In the web app, this would resemble **receive** $(x, v : \text{sInt}).P$ from which we can see that v is an integer value. This helps make it more clear what variables are intended for, and thus what a process does.

3.2 Typing Extensions

The web app's π -calculus possesses, as an extension on the standard π -calculus, integers, strings and booleans as predefined types. These have possible values of any positive or negative integer, any sequence of alphanumeric characters prepended and appended with quotation marks, and `True` or `False` respectively. It also contains expressions on these basic types, such as integer arithmetic or boolean algebra operations. These types and expressions were included to help the user more easily reason about how π -calculus works and how it can be useful, as expressing all values as channels can be confusingly abstract. Presented in Figure 3.2 are the typing rules for these types and expressions.

$\frac{\text{un}(\Gamma) \quad n \in \mathbb{Z}}{\Gamma \vdash n : \text{sInt}}$	$\frac{\text{un}(\Gamma)}{\Gamma \vdash \text{True} : \text{sBool}}$	$\frac{\text{un}(\Gamma)}{\Gamma \vdash \text{False} : \text{sBool}}$
$\frac{\text{un}(\Gamma) \quad l = \text{"... "}}{\Gamma \vdash l : \text{sString}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : T, y : T}{\Gamma \vdash (x == y) : \text{sBool}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : T, y : T}{\Gamma \vdash (x != y) : \text{sBool}}$
$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x + y) : \text{sInt}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x - y) : \text{sInt}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x * y) : \text{sInt}}$
$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x / y) : \text{sInt}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x \% y) : \text{sInt}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x > y) : \text{sBool}}$
$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x >= y) : \text{sBool}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x < y) : \text{sBool}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sInt}, y : \text{sInt}}{\Gamma \vdash (x <= y) : \text{sBool}}$
$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sBool}}{\Gamma \vdash (\text{NOT } x) : \text{sBool}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sBool}, y : \text{sBool}}{\Gamma \vdash (x \text{ AND } y) : \text{sBool}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sBool}, y : \text{sBool}}{\Gamma \vdash (x \text{ OR } y) : \text{sBool}}$
$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sBool}, y : \text{sBool}}{\Gamma \vdash (x \text{ XOR } y) : \text{sBool}}$	$\frac{\text{un}(\Gamma) \quad \Gamma \vdash x : \text{sString}, y : \text{sString}}{\Gamma \vdash (x ++ y) : \text{sString}}$	

Figure 3.2: A list of the rules used to typecheck basic values and expressions in the web app's π -calculus. These rules shown are the rules used in session-typed π -calculus. The rules for linear-typed π -calculus, named $T\pi$ -__ instead of T -__, differ only in the types $sUnit$, $sInt$, etc. instead being $lUnit$, $lInt$, etc.. In T -String, the ellipsis represents any arbitrary sequence of alphanumeric characters.

3.3 Web Interface

The primary concept for the web app is that it allows people to write code in session- or linear-typed π -calculus, encode the former into the latter, and execute both, to help improve understanding of these concepts. So, the interface needs to have areas to enter code, and areas to display output from performing these operations. As session-typed and linear-typed π -calculus are different, they should have separate sections in the interface. So, the overall layout of the interface contains four main sections: input and output sections each for session-typed and linear-typed π -calculus. As the input sections are the sections the user would be interacting with first, these are placed along the top to emphasise them over the output sections along the bottom. Buttons to perform the operations on the user's input are placed in a strip in between the input and output sections. As the encoding occurs from session types to linear types, the sections relating to session-typed π -calculus are placed on the left, and linear-typed on the right. Figure 3.3 shows the interface of the web app's main page, and Figure 3.4 shows the maths server example entered into the interface.

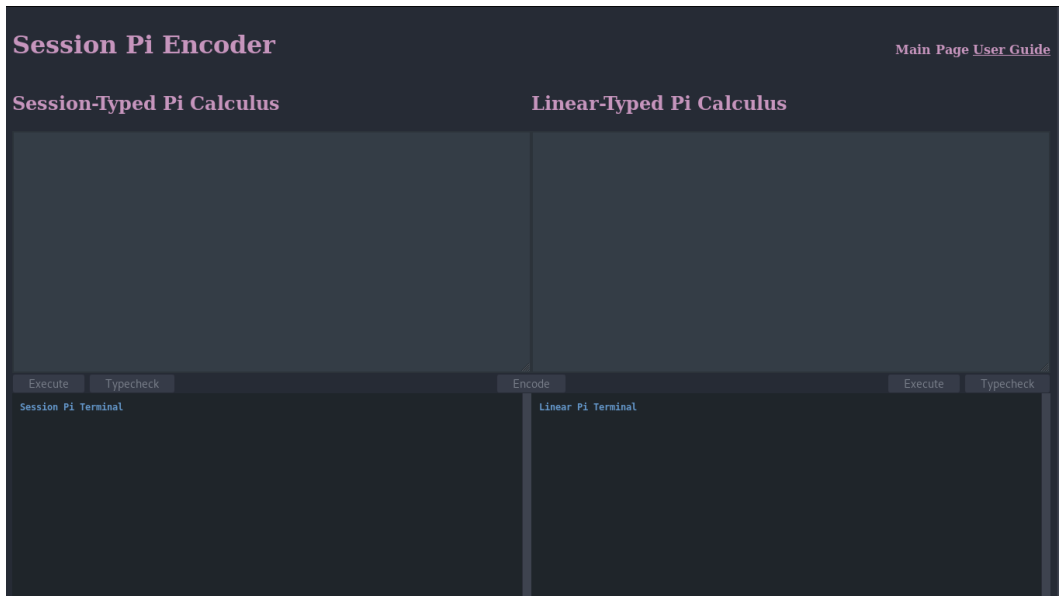


Figure 3.3: The web app's main page, as it appears when first accessing the web app. As described above, the top-left and top-right areas are input areas to type π -calculus code for session types and linear types respectively, and the bottom-left and bottom-right are the output areas for session types and linear types respectively. As there is no input in either of the input areas, the buttons are all disabled, and become enabled when there is text present in the appropriate input area.

Session-Typed Pi Calculus

```

type sv := &{ plus : ?sInt.?sInt.!sInt.end,
             equal : ?sInt.?sInt.!sBool.end,
             neg : ?sInt.!sInt.end },
type cl := +{ plus : !sInt.!sInt.?sInt.end,
             equal : !sInt.!sInt.?sBool.end,
             neg : !sInt.?sInt.end },
server(a : sv) := branch(a){ plus : receive(a, v1 : sInt).receive(a, v2 : sInt).send(a, (v1+v2))
                             equal : receive(a, v1 : sInt).receive(a, v2 : sInt).send(a, (v1==v2))
                             neg : receive(a, v : sInt).send(a, (-1*v)).stop },
client(b : cl) := select(b, equal).send(b, 3).send(b, 5).receive(b, eq : sBool).stop

(new x y : sv) ((server(x) | client(y)))

```

Figure 3.4: The maths server example, entered into the session-typed π -calculus input area. The input areas have horizontal scrolling, as wrapping the text onto the next line was found to be difficult to read.

The interface has 5 buttons that perform operations on the user's π -calculus code: An execute button for each of session-typed and linear-typed π -calculus, a typecheck button for each, and an encode button. The execute and encode buttons also typecheck the code alongside their own functionality. When code is typechecked, the interface produces a success message detailing the typing rules used. When code is executed, the interface displays a success message with a list of the reductions made as part of execution. Encoding from session-typed to linear-typed π -calculus produces a brief success message, and automatically inserts the encoded π -calculus into the input area for linear-typed π -calculus. Each of these can instead produce error messages, if some part of the operation is unsuccessful due to the user input containing incorrect or invalid π -calculus. Figures 3.5, 3.6, 3.7, 3.8 and 3.9 show respectively the outputs of typechecking the maths server example in session types, executing it in session types, encoding it into linear types, typechecking the encoding, and executing the encoding.

```

Session Pi Terminal
Typechecking successful. Rules used:
T-Res
[x : &{plus:?sInt.?sInt.!sInt.end, equal:?sInt.?sInt.!sBool.end, neg:?sInt.!sInt.end},
 y : +{plus:!sInt.!sInt.?sInt.end, equal:!sInt.!sInt.?sBool.end, neg:!sInt.?sInt.end}],
T-Par
(L: T-Var (x), T-Brch
  {plus : T-Var (x), T-In, T-Var (x), T-In, T-Var (x), T-Add (T-Var (v1), T-Var (v2))},
T-Out, T-Inact;
  equal : T-Var (x), T-In, T-Var (x), T-In, T-Var (x), T-Equal (T-Var (v1), T-Var (v2)),
T-Out, T-Inact;
  neg : T-Var (x), T-In, T-Var (x), T-Mult (T-Int, T-Var (v)), T-Out, T-Inact}
|
R: T-Var (y), T-SEL, T-Var (y), T-Int, T-Out, T-Var (y), T-Int, T-Out, T-Var (y), T-In,
T-Inact)

```

Figure 3.5: Typechecking in session-typed π -calculus. This is the typechecking output for the maths server example. T-Par structures the rest of the typing rules similarly to how the parallel composition structures its processes, to help readability of the output. Similarly, expressions contain the typing rules of their operands in parentheses after the typing rule for the expression itself. T-Res shows the types of the session endpoints it is restricting, and T-Var shows the name of the variable being checked.

```

Session Pi Terminal
Typechecking successful. Rules used:
T-Res
[x : &{plus:?sInt.?sInt.!sInt.end, equal:?sInt.?sInt.!sBool.end, neg:?sInt.!sInt.end},
 y : +{plus:!sInt.!sInt.?sInt.end, equal:!sInt.!sInt.?sBool.end, neg:!sInt.?sInt.end}],
T-Par
(L: T-Var (x), T-Brch
  {plus : T-Var (x), T-In, T-Var (x), T-In, T-Var (x), T-Add (T-Var (v1), T-Var (v2))},
T-Out, T-Inact;
  equal : T-Var (x), T-In, T-Var (x), T-In, T-Var (x), T-Equal (T-Var (v1), T-Var (v2)),
T-Out, T-Inact;
  neg : T-Var (x), T-In, T-Var (x), T-Mult (T-Int, T-Var (v)), T-Out, T-Inact}
|
R: T-Var (y), T-SEL, T-Var (y), T-Int, T-Out, T-Var (y), T-Int, T-Out, T-Var (y), T-In,
T-Inact)
Execution successful. Actions performed:
Selecting equal on y out of {plus, equal, neg} from x. (R-Case)
Sending 3 from y to x, replacing v1. (R-Com)
Sending 5 from y to x, replacing v2. (R-Com)
Sending (3==5) = False from x to y, replacing eq. (R-Com)

```

Figure 3.6: Execution of session-typed π -calculus. This is the execution output for the maths server example. Each message shows what message is being passed, and what bound variables are being replaced. Note also that the final message shows both the expression present in the output process and the value obtained from it that is the actual payload.

Session-Typed Pi Calculus	Linear-Typed Pi Calculus
<pre> type sv := &{ plus : ?sInt.?sInt.!sInt.end, equal : ?sInt.?sInt.!sBool.end, neg : ?sInt.!sInt.end }, type cl := +{ plus : !sInt.!sInt.?sInt.end, equal : !sInt.!sInt.?sBool.end, neg : !sInt.?sInt.end }, server(a : sv) := branch(a){ plus : receive(a, v1 : sInt).receive(a, v2 : sInt).send(a, (v1+v2) equal : receive(a, v : sInt).receive(a, v2 : sInt).send(a, (v1*v2) neg : receive(a, v : sInt).send(a, (-1*v)).stop }, client(b : cl) := select(b, equal).send(b, 3).send(b, 5).receive(b, eq : sBool).stop (new x y : sv) ((server(x) client(y))) </pre>	<pre> type sv' := li{< plus li{!Int, li{!Int, lo{!Int, empty[]}}}, equal li{!Int, li{!Int, lo{!Bool, empty[]}}}, neg li{!Int, lo{!Int, empty[]}}}, type cl' := lo{< plus li{!Int, li{!Int, lo{!Int, empty[]}}}, equal li{!Int, li{!Int, lo{!Bool, empty[]}}}, neg li{!Int, lo{!Int, empty[]}}}, server'(c : sv') := receive(c, c' : < plus li{!Int, li{!Int, lo{!Int, empty[]}}}, equal li{!Int, li{!Int, lo{!Bool, empty[]}}}, neg li{!Int, lo{!Int, empty[]}}}, case c' of { plus c'' : li{!Int, li{!Int, lo{!Int, empty[]}}}, receive(c'', v1 : !Int, c''' : li{!Int, lo{!Int, empty[]}}).receive(c''', v2 : !Int, c'''' : li{!Int, lo{!Bool, empty[]}}), equal c'' : li{!Int, li{!Int, lo{!Bool, empty[]}}}, receive(c'', v1 : !Int, c''' : li{!Int, lo{!Bool, empty[]}}).receive(c''', v2 : !Int, c'''' : li{!Int, lo{!Int, empty[]}}), neg c'' : li{!Int, lo{!Int, empty[]}}}, receive(c'', v : !Int, c''' : lo{!Int, empty[]}).(new c'''' : empty[]) (send(c''', (-1* client'(c : cl') := (new c' : !li{!Int, li{!Int, lo{!Bool, empty[]}}}) { </pre>

Figure 3.7: Encoding from session-typed π -calculus into linear-typed π -calculus. This is the encoding of the maths server example. The encoded version is clearly much longer and more complicated, only part of it able to be shown in the input area at one time.

```

Linear Pi Terminal
Typechecking successful. Rules used:
T $\pi$ -Res1
[c : li[]],
T $\pi$ -Par
(L: T $\pi$ -Var (c), T $\pi$ -Inp, T $\pi$ -Var (c'), T $\pi$ -Case
  {plus : T $\pi$ -Var (c'), T $\pi$ -Inp, T $\pi$ -Var (c''), T $\pi$ -Inp, T $\pi$ -Res2
  [c'''' : empty[]],
  T $\pi$ -Var (c'''), T $\pi$ -Add (T $\pi$ -Var (v1), T $\pi$ -Var (v2)), T $\pi$ -Var (c'''''), T $\pi$ -Out, T $\pi$ -Inact;
  equal : T $\pi$ -Var (c'), T $\pi$ -Inp, T $\pi$ -Var (c''), T $\pi$ -Inp, T $\pi$ -Res2
  [c'''' : empty[]],
  T $\pi$ -Var (c'''), T $\pi$ -Equal (T $\pi$ -Var (v1), T $\pi$ -Var (v2)), T $\pi$ -Var (c'''''), T $\pi$ -Out, T $\pi$ -Inact;
  neg : T $\pi$ -Var (c'), T $\pi$ -Inp, T $\pi$ -Res2
  [c'''' : empty[]],
  T $\pi$ -Var (c'''), T $\pi$ -Mult (T $\pi$ -Int, T $\pi$ -Var (v)), T $\pi$ -Var (c'''''), T $\pi$ -Out, T $\pi$ -Inact}
|
R: T $\pi$ -Res1
[c' : lo{!Int, li{!Int, lo{!Bool, empty[]}}}],
T $\pi$ -Var (c), T $\pi$ -LVal (equal_c'), T $\pi$ -Out, T $\pi$ -Res1
[c'' : lo{!Int, lo{!Bool, empty[]}}],
T $\pi$ -Var (c'), T $\pi$ -Int, T $\pi$ -Var (c''), T $\pi$ -Out, T $\pi$ -Res1
[c''' : li{!Bool, empty[]}],
T $\pi$ -Var (c''), T $\pi$ -Int, T $\pi$ -Var (c'''), T $\pi$ -Out, T $\pi$ -Var (c'''), T $\pi$ -Inp, T $\pi$ -Inact)

```

Figure 3.8: Typechecking in linear-typed π -calculus. This is the typechecking output for the encoding of the maths server example. Much like the encoding itself, this is clearly much longer and more complicated than the typechecking output of the maths server example in session types. This is due to the inserted restrictions for continuation channels.


```

 $\pi$ -Par
(L:  $\pi$ -Var (c),  $\pi$ -Inp,  $\pi$ -Var (c'),  $\pi$ -Case
 {plus :  $\pi$ -Var (c'),  $\pi$ -Inp,  $\pi$ -Var (c''),  $\pi$ -Inp,  $\pi$ -Res2
 [c'''' : empty[]],
  $\pi$ -Var (c'''),  $\pi$ -Add ( $\pi$ -Var (v1),  $\pi$ -Var (v2)),  $\pi$ -Var (c'''''),  $\pi$ -Out,  $\pi$ -Inact;
 equal :  $\pi$ -Var (c'),  $\pi$ -Inp,  $\pi$ -Var (c''),  $\pi$ -Inp,  $\pi$ -Res2
 [c'''' : empty[]],
  $\pi$ -Var (c'''),  $\pi$ -Equal ( $\pi$ -Var (v1),  $\pi$ -Var (v2)),  $\pi$ -Var (c'''''),  $\pi$ -Out,  $\pi$ -Inact;
 neg :  $\pi$ -Var (c'),  $\pi$ -Inp,  $\pi$ -Res2
 [c'''' : empty[]],
  $\pi$ -Var (c''),  $\pi$ -Mult ( $\pi$ -Int,  $\pi$ -Var (v)),  $\pi$ -Var (c'''''),  $\pi$ -Out,  $\pi$ -Inact}
|
R:  $\pi$ -Res1
 [c' : lo[lInt, li[lInt, lo[lBool, empty[]]]]],
  $\pi$ -Var (c),  $\pi$ -LVal (equal_c'),  $\pi$ -Out,  $\pi$ -Res1
 [c'' : lo[lInt, lo[lBool, empty[]]]],
  $\pi$ -Var (c'),  $\pi$ -Int,  $\pi$ -Var (c''),  $\pi$ -Out,  $\pi$ -Res1
 [c'''' : li[lBool, empty[]]],
  $\pi$ -Var (c''),  $\pi$ -Int,  $\pi$ -Var (c'''),  $\pi$ -Out,  $\pi$ -Var (c'''),  $\pi$ -Inp,  $\pi$ -Inact)
Execution successful. Actions performed:
Sending equal_c' over c, replacing c'. (R $\pi$ -Com)
Selecting case equal out of cases plus, equal, neg, replacing c'' with c'. (R $\pi$ -Case)
Sending 3, c'' over c', replacing v1, c'''. (R $\pi$ -Com)
Sending 5, c''' over c'', replacing v2, c'''''. (R $\pi$ -Com)
Sending (3==5) = False, c'''''' over c''', replacing eq, c'''''. (R $\pi$ -Com)

```

Figure 3.9: Execution of linear-typed π -calculus. This is the execution output for the encoding of the maths server example. Unlike the encoding itself and the typechecking output, semantics in linear types is not strongly complicated by the encoding, with only one extra reduction compared to the maths server example in session types.

As the web app is intended as a learning tool, it contains a user guide. The user guide details the syntax and explains the functionalities of π -calculus, to teach the user how to write valid and meaningful π -calculus code. It also contains sections detailing the typing rules, the encoding and the reduction rules used in execution, to give the user insight into how the web app performs these actions. There are also some minor notes on using the web app, such as things that produce unexpected results due to the parser being unable to interpret it as intended. Figure 3.10 shows the beginning of the user guide.

Session Pi Encoder [Main Page User Guide](#)

Syntax Guide

Declarations

Your code should be structured as a list of declarations, followed by a pi calculus process. This section describes the valid declarations that can be made. In this section, P represents an arbitrary process.

- **Type Declaration** - `type var varType e.g. type alpha sInt` - This is used to declare to type of a variable.
- **Variable Assignment** - `var = value e.g. alpha = 3` - This is used to assign a value to a variable which can take a value (e.g. integers, strings or booleans).
- **Type Declaration and Assignment** - `type var varType = val e.g. type alpha sInt = 3` - This combines the above two into one declaration.
- **Type Naming** - `type typeName := fullType e.g. type msg := sString` - This assigns a name to a particular type, so that that name may be used later in place of that type.
- **Process Naming** - `processName(dummyVar : dummyType) := fullProcess e.g. client(a : !sInt, ?sInt, end) := P` - This assigns a name to a particular process, so that that name may be used later in place of that process. The declaration includes a dummy variable of a specified type, which is replaced with a variable passed into the process when called later, similarly to an argument to a function.

Session-typed Pi Calculus

Processes

In the following section, P, Q and R represent arbitrary processes, x and y represent session endpoints, z represents a standard channel and w and v represent arbitrary values.

- **Parallel Composition** - `(process | process) e.g. (P | 0)` - This allows two processes to run in parallel, allowing them to communicate with each other.
- **Output** - `send(channel, payload).process e.g. send(x, v).P` - This process sends a value as a payload from channel to the corresponding session endpoint, or to itself if channel is a standard channel. This process is the counterpart to Input.
- **Input** - `receive(channel, payload : payloadType).process e.g. receive(x, w : sInt).P` - This process receives a value as a payload of type payloadType on channel that has been sent by the corresponding session endpoint, or by itself if channel is a standard channel. This process is the counterpart to Output.

Figure 3.10: The web app's user guide. The user guide begins with a syntax guide before explaining the concepts behind the actions of the web app, as the syntax is more important to being able to use the web app. The sections for typechecking, semantics and encoding use figures similar to those in this work.

The interface makes use of colour to draw attention to certain aspects of the app. In the main page, colours are used to draw attention to the labels of each section, making it clear what each section is, and are used in the output to give the user information on what has happened at a glance, e.g. when an action is performed successfully, the success message is highlighted in orange, while the details are plain, or if an error occurs, the error message is red. In the user guide, headers and sub-headers are coloured differently from each other to make the structure of the guide clear.

4 | Implementation

4.1 Architecture

The project is written in Python (Python Software Foundation), making use of Flask as a framework for the web app aspects of the project, and ANTLR for the language recognition aspects.

Flask is a web development micro-framework written in Python. By micro-framework, the developers of Flask mean that "Flask aims to keep the core simple but extensible". (Pocoo Team) Flask was chosen as the framework for the project for this reason. The web app is lightweight, not requiring any significant back-end technologies found in most web apps such as databases or user authentication. As many web app frameworks contain these features by default while Flask does not, Flask was an ideal candidate for the project.

ANTLR, short for **AN**other **T**ool for **L**anguage **R**ecognition, is a parser generator. (Parr 2014) Given a grammar for a language, ANTLR can automatically generate parsers that transform something written in that language into a syntax tree, and templates for programs that traverse that tree, performing some action at each node. This allows coders to easily create programming languages and other tools involving files fitting a custom-defined structure. ANTLR was chosen for the project as it allowed us to focus on the implementation of what should be done with the π -calculus code once it has been parsed, rather than the implementation of correctly parsing it.

4.2 Web Interface

The implementation of the web interface is primarily in the file `SPEncoder.py`. This file contains the code defining the Flask app, and connects the Flask app to the ANTLR code detailed in the following sections. It handles the main page and the user guide, which simply render HTML files `SPEncoder.html` and `SPGuide.html` respectively.

Whenever a button on the interface is pressed, the web app's JavaScript, defined in `SPEncoder.js`, gets the contents of the appropriate text area, and sends a request containing those contents to a certain URL, depending on which button is pressed. These URLs are connected to functions in `SPEncoder.py` that instantiate an ANTLR parser to which they give the code supplied by the JavaScript. They then instantiate the ANTLR listeners detailed in the following sections to perform the operation corresponding to the button pressed. The listeners produce an output, or an error message, and `SPEncoder.py` returns them as JSON to `SPEncoder.js`, which inserts them into the appropriate elements. This is coded using AJAX, so that the results appear dynamically.

4.3 Encoding

The encoding is implemented as part of an ANTLR listener, `SPListener.py`. This listener also contains the code to perform the typechecking, as detailed in Section 4.4, which it can do concurrently with the encoding. For a program in session-typed π -calculus code, the encoding of this program is generated by constructing a string of the encoding from information in each of the nodes of the syntax tree.

Before the encoding begins, a separate listener `VariableNameCollector.py` is run to form a list of variable names the user has used in their code. This is done to ensure that none of the new channels generated by the encoding use a pre-existing name. Generated channels only use names of the form `c`, `c'`, `c''`, etc. which users are advised in the user guide not to use, but `VariableNameCollector.py` is used in case users don't follow this advice.

The encoding starts as a string containing a placeholder, to be used as a string builder. Each rule of the grammar has corresponding functions which replace the first placeholder in the string builder with a string representing the encoding of that rule, the string typically containing more placeholders itself. Different placeholders are used to represent processes, types, declarations, etc. A dictionary is used to act as the encoding function.

As the encoding inserts new restrictions and new payloads, these restrictions and the input processes containing these payloads need additional type annotations. These were generated using a dictionary `contChanTypes` to keep track of the type of session endpoints at the current point in the process. When an inserted type annotation needs to be generated, a new parse tree walker is created to traverse the current saved type of the session endpoint, using the current listener instance. This essentially temporarily redirects the tree traversal of the listener to the session endpoint's type, after which it returns to where it was in the process.

Stacks are used in the listener for multiple purposes, such as to keep old copies of the encoding function, and replace the current encoding function with these copies when appropriate. For example, when encoding a branch process with two branches `alpha` and `beta`, a copy of the encoding function is saved to a stack, and then the process `alpha` is encoded. After this, the encoding function is replaced with the saved copy from before `alpha` was encoded, and the process `beta` is encoded with this copy, so that the encoding of `beta` does not use variables created in `alpha`.

Listing 4.1 presents as an example the function `enterInputSesEnc`, used to encode a session-typed output process.

```

1 def enterInputSesEnc(self, ctx):
2     self.checkBranchStack(False)
3     self.checkCompStack(False)
4     if isinstance(self.contChanTypes[ctx.channel.getText()],
5         PiCalcParser.ChannelTypeContext):
6         self.encodedStrBuilder = self.encodedStrBuilder.replace(u"●", "receive(" +
7             ctx.channel.getText() + ", " + ctx.payload.getText() + u" : ▲).●", 1)
8     else:
9         ipStrBuilder = "receive(" + self.encodeName(ctx.channel.getText()) + ", " +
10            ctx.payload.getText() + u" : ★"
11         newChan = self.generateChannelName()
12         newChanType = self.contChanTypes[ctx.channel.getText()].sType()
13         self.contChanTypes[ctx.channel.getText()] = newChanType
14         if isinstance(ctx.plType, PiCalcParser.SessionTypeContext):
15             self.contChanTypes[ctx.payload.getText()] = ctx.plType.sType()
16         elif isinstance(ctx.plType, PiCalcParser.ChannelTypeContext):
17             self.contChanTypes[ctx.payload.getText()] = ctx.plType
18         ipStrBuilder = ipStrBuilder + ", " + newChan + u" : ▲).●"
19         self.encFunc[ctx.channel.getText()] = newChan
20         self.encodedStrBuilder = self.encodedStrBuilder.replace(u"●", ipStrBuilder,
21             1)
22         typeWalker = ParseTreeWalker()
23         typeWalker.walk(self, newChanType)
24         self.encodedStrBuilder = self.encodedStrBuilder.replace(u"★", u"▲", 1)

```

Listing 4.1: `enterInputSesEnc`, the function used to encode a session-typed input process. `ctx` is the ANTLR context of the current instance of the grammar rule. It is the object containing everything found within this instance, such as the ANTLR contexts of instances of other grammar rules that appear within this instance's rule. `self.checkBranchStack` and `self.checkCompStack` check whether the current process is the initial process in the continuation of a branch or an arm of a parallel composition respectively and the encoding function or list of existing variable names should therefore be reset. These functions are both passed a parameter `False` to indicate the current process is not a branch or composition respectively. The `if` statement checks if the input process uses a standard channel, and gives a simpler encoding if so. The `else` clause generates a new channel name, updates `contChanTypes`, adds the payload to `contChanTypes` if it is a channel, adds the new channel to the encoding function and adds the encoded string to the string builder. It then encodes the type of the newly generated type annotation. ● is used as a placeholder for processes, ▲ is used as a placeholder for types and ★ is used so that type annotations are inserted in the correct order.

4.4 Typechecking

Typechecking is implemented in `SPEListener.py`. Given a program in either session- or linear-typed π -calculus, the listener traverses the syntax tree of this program, applying typing rules as appropriate and generating an output string detailing what typing rules it applies.

The listener starts by collecting types from the type declarations at the start of the program, if there are any. Once it has looked at all of these, it now has the initial typing context `gamma`. It then begins traversing the processes. As many of the process typing rules involve the context split or context combination operator and using one of the separated typing contexts in another typing rule, the listener needs some way of saving contexts for later, which is accomplished through the `gammaStack`. For all typing rules, the current typing context is retrieved from the stack, premises of the typing rule which do not involve another typing rule are checked, and then any typing contexts that are used to check another typing rule in the premise are added to the stack so that those can be checked when the listener's traversal reaches them.

`VariableNameCollector.py` is also used in the typechecking, this time to aid the context split or context combination operators. When using these operators, a list of variable names is used to decide which linear types should go into which of the separated typing contexts, those in the list going into the first, all others into the second. In `T-Par` and `T π -Par`, `VariableNameCollector.py` is used to collect all of the variable names used in the left arm of the parallel composition, so that these are placed into the first typing context. For `T π -Par`, `VariableNameCollector.py` can also collect what each channel is used for, i.e. which capability of a linear channel appears in the left arm of the parallel composition.

Listing 4.2 presents as an example the function `enterInputSesSTCh`, the function used to typecheck a session-typed input process. The function `enterInputLinLTCh`, used to typecheck a linear-typed input process, is presented in Listing 4.3. Note the names of these functions, `InputSesSTCh` and `InputLinLTCh`, each contain two references to which typing system is used. This is because the naming convention we used for typechecking in session types is to add `STCh` to the function name, and `LTCh` for linear types, for example `enterOutputSTCh` and `enterOutputLTCh`. However, input processes are defined with two separate grammar rules `InputSes` and `InputLin`, so that an input process's type annotations may contain either session types or linear types. The functions `enterInputSesLTCh` and `enterInputLinSTCh` do exist, but as they are not meaningful, they simply throw errors and cause the typechecking to fail.

```

1 def enterInputSesSTCh(self, ctx):
2     self.gamma = self.gammaStack.pop()
3     trueChan = self.getReplacement(ctx.channel)
4     truePL = self.getReplacement(ctx.payload)
5     (gamma1, gamma2) = self.splitGamma(self.gamma, [trueChan.getText()])
6     chanType = gamma1.get(trueChan.getText())
7     payloadType = ctx.tType()
8     if isinstance(payloadType, PiCalcParser.BasicSesTypeContext):
9         payloadType = payloadType.basicSType()
10    elif isinstance(payloadType, PiCalcParser.SessionTypeContext):
11        payloadType = payloadType.sType()
12    if not isinstance(chanType, PiCalcParser.ChannelTypeContext):
13        if not isinstance(chanType, PiCalcParser.ReceiveContext):
14            self.tcErrorStrBuilder = self.tcErrorStrBuilder = "<span
                class='error'>ERROR: Typechecking rule T-In failed due to " +
                trueChan.getText() + ". Input process on channel not of receive
                type.</span>\n"
15            raise self.typecheckException
16        chanPLType = chanType.tType()
17        if isinstance(chanPLType, PiCalcParser.BasicSesTypeContext):
18            chanPLType = chanPLType.basicSType()
19        elif isinstance(chanPLType, PiCalcParser.SessionTypeContext):
20            chanPLType = chanPLType.sType()
21        if not isinstance(payloadType, type(chanPLType)):
22            self.tcErrorStrBuilder = self.tcErrorStrBuilder = "<span
                class='error'>ERROR: Typechecking rule T-In failed due to " +
                truePL.getText() + ". Input process payload has type annotation that
                does not match channel type.</span>\n"
23            raise self.typecheckException
24        else:
25            augmentations = {trueChan.getText(): chanType.sType(), truePL.getText():
                chanPLType}
26            gamma2 = self.augmentGamma(gamma2, augmentations)
27            self.gammaStack.append(gamma2)
28            self.typeCheckStrBuilder = self.typeCheckStrBuilder.replace(u"□",
                u"□△T-In△□", 1)
29            self.printVariableTypeRule(trueChan, gamma1)
30    else: (...)

```

Listing 4.2: `enterInputSesSTCh`, the function used to typecheck a session-typed input process. `ctx` is as described in Listing 4.1. `self.getReplacement` returns the variable which replaces the channel or payload, should either of those be the dummy variable of a process naming declaration. `self.splitGamma` represents the context split operator. It is passed the typing context to split and a list of variables which if linear should be placed in the first context, with all other linear variables placed in the second. The function then retrieves the types of the channel and the payload, and checks if the channel is a standard channel or a receive session. It then checks that the type of the payload matches the channel type's payload type. The function then prepares the typing context for the continuation process and adds the typing rule into the typechecking output. `self.printVariableTypeRule` inserts into the typechecking output the typing rule for the channel i.e. T-Var. The final `else` clause is the code for rule T-StdIn and is near identical to lines 16–29, so it has been omitted for sake of brevity. \square is used as a placeholder for typing rules, \triangle is used as a placeholder for commas as a separator between the rules.

```

1 def enterInputLinLTCh(self, ctx):
2     self.gamma = self.gammaStack.pop()
3     trueChan = self.getReplacement(ctx.channel)
4     truePLs = [self.getReplacement(pl) for pl in ctx.payloads]
5     (gamma1, gamma2) = self.combineGamma(self.gamma, [(trueChan.getText(),
6         "Input")])
7     chanType = gamma1.get(trueChan.getText())
8     payloadTypes = copy.deepcopy(ctx.plTypes)
9     for i in range(len(payloadTypes)):
10        if isinstance(payloadTypes[i], PiCalcParser.BasicLinTypeContext):
11            payloadTypes[i] = payloadTypes[i].basicLType()
12    if not isinstance(chanType, PiCalcParser.ConnectionContext):
13        if not isinstance(chanType, PiCalcParser.LinearInputContext):
14            self.tcErrorStrBuilder = self.tcErrorStrBuilder + "<span
15                class='error'>ERROR: Typechecking rule  $T\pi$ -Inp failed due to " +
16                trueChan.getText() + ". Input process on channel not of input
17                type.</span>\n"
18            raise self.typecheckException
19        else:
20            augmentations = {}
21            chanPLTypes = chanType.payloads
22            for i in range(len(payloadTypes)):
23                if isinstance(chanPLTypes[i], PiCalcParser.BasicLinTypeContext):
24                    chanPLTypes[i] = chanPLTypes[i].basicLType()
25                if not isinstance(payloadTypes[i], type(chanPLTypes[i])):
26                    self.tcErrorStrBuilder = self.tcErrorStrBuilder + "<span
27                        class='error'>ERROR: Typechecking rule  $T\pi$ -Inp failed due to "
28                        + truePLs[i].getText() + ". Input process payload has type
29                        annotation that does not match channel type.</span>\n"
30                    raise self.typecheckException
31                augmentations[truePLs[i].getText()] = chanPLTypes[i]
32            gamma2 = self.augmentGamma(gamma2, augmentations)
33            self.gammaStack.append(gamma2)
34            self.typeCheckStrBuilder = self.typeCheckStrBuilder.replace(u"□",
35                u"□ $\Delta T\pi$ -Inp $\Delta$ □", 1)
36            self.printVariableTypeRule(trueChan, gamma1)
37    else: (...)

```

Listing 4.3: `enterInputLinLTCh`, the function used to typecheck a linear-typed input process. This function works similarly to `enterInputSesSTCh`, detailed in Listing 4.2. The main differences are the use of `self.combineGamma` instead of `self.splitGamma` and the handling of multiple payloads. The final `else` clause has again been omitted for brevity and similarity to lines 16–29.

4.5 Semantics

Operational semantics of π -calculus are implemented in an ANTLR listener `SPERunner.py`. Given a program in session- or linear-typed π -calculus code, it executes the program by repeatedly checking for reductions between pairs of processes and applying them, until no more reductions can be made. For each reduction it makes, it records for the output what this reduction does.

The listener starts by parsing the declarations, gathering any variables to which values are assigned, then parsing the process, collecting information on what channels exist and what they communicate with from restrictions. Once the parser reaches a parallel composition, it creates a list containing the composed processes. This list begins simply as the two arms of the initial parallel composition, but the listener then refines the list by checking if these are restrictions or nested compositions, and gathering processes from these. Once the final list of composed processes has been found, where all of the processes are some process on which a reduction can occur, the listener starts checking for pairs of processes which can be reduced.

When it has found such a pair, e.g. an output process and an input process, it checks the channels to see if these processes are communicating. If they are, it then performs the reduction, replacing the processes in the list with their continuations, and saving what bound variables have been replaced as a result of communication. After all pairs have been checked, the listener checks whether all processes are terminated. The listener will then finish execution successfully, begin a new cycle of checking for reductions, or finish execution unsuccessfully, depending on whether all processes are terminated, not all processes are terminated and a reduction was made this cycle, or not all processes are terminated and no reduction was made this cycle.

Listing 4.4 presents as an example the implementation of the reduction rules R-Com and R-StdCom.

```

1  if isinstance(self.parProcs[i], PiCalcParser.OutputContext) and
2     isinstance(self.parProcs[j], PiCalcParser.InputSesContext):
3     if (self.getReplacement(i, self.parProcs[i].channel).getText() ==
4         self.chanCounterparts[self.getReplacement(j,
5             self.parProcs[j].channel).getText()] and (self.getReplacement(j,
6             self.parProcs[j].channel).getText() ==
7             self.chanCounterparts[self.getReplacement(i,
8             self.parProcs[i].channel).getText()]):
9         payloadText = self.getReplacement(i, self.parProcs[i].payloads[0]).getText()
10        if isinstance(self.parProcs[i].payloads[0], PiCalcParser.ExprValueContext):
11            evalExpr =
12                self.evaluateExpression(self.parProcs[i].payloads[0].expression(), i)
13            payloadText =
14                self.printExpression(self.parProcs[i].payloads[0].expression(), i) +
15                " = " + evalExpr.getText()
16            self.replacements[j][self.parProcs[j].payload.getText()] = evalExpr
17        else:
18            if isinstance(self.getReplacement(i, self.parProcs[i].payloads[0]),
19                PiCalcParser.NamedValueContext) and self.getReplacement(i,
20                self.parProcs[i].payloads[0]).getText() in self.variableValues:
21                payloadText = payloadText + " = " +
22                    self.variableValues[self.getReplacement(i,
23                        self.parProcs[i].payloads[0]).getText()].getText()
24            self.replacements[j][self.parProcs[j].payload.getText()] =
25                self.getReplacement(i, self.parProcs[i].payloads[0])
26        if self.getReplacement(i, self.parProcs[i].channel).getText() ==
27            self.getReplacement(j, self.parProcs[j].channel).getText():
28            self.executionStrBuilder = self.executionStrBuilder + "Sending " +
29                payloadText + " over " + self.getReplacement(i,
30                self.parProcs[i].channel).getText() + ", replacing " +
31                self.parProcs[j].payload.getText() + ". (R-StndCom)\n"
32        else:
33            self.executionStrBuilder = self.executionStrBuilder + "Sending " +
34                payloadText + " from " + self.getReplacement(i,
35                self.parProcs[i].channel).getText() + " to " +
36                self.getReplacement(j, self.parProcs[j].channel).getText() + ",
37                replacing " + self.parProcs[j].payload.getText() + ". (R-Com)\n"
38        self.parProcs[i] = self.parProcs[i].processSec()
39        self.parProcs[j] = self.parProcs[j].processSec()
40        reductionMade = True

```

Listing 4.4: The implementation of the reduction rules *R-Com* and *R-StndCom*. `self.parProcs` is the list of processes composed in parallel, and `self.chanCounterparts` is a list recording where each channel communicates with, i.e. itself if it is a standard or linear channel, or its co-name if it is a session endpoint. `self.getReplacement` returns what a variable has been replaced by as a result of communication, if it has been replaced. Once the listener has determined that the processes can communicate, it checks if the payload is an expression, and evaluates it if so, and if not, it checks if it is a named variable with an assigned value. In either of these cases, the value of the payload is printed in the output along with the payload itself. The input process's payload variable is replaced with the output process's. The listener then checks if the reduction rule being applied is *R-Com* or *R-StndCom*, and adds the appropriate string to the execution output. The list of processes composed in parallel is updated with the continuation processes, and the listener makes a note that a successful reduction was made.

5 | Evaluation

5.1 User Study

To evaluate how effective the web app is as a teaching tool, we performed a user study. We gave participants access to the web app, and a survey asking them to perform some tasks in the web app and then rate their experience. The full survey can be seen in Appendix A.1. Participants were gathered in the following way:

- One group of 5 participants who were not already familiar with π -calculus.
- One group of 5 participants who had been formally taught π -calculus, in the Theory of Computation course. Of this group:
 - 3 participants had taken Theory of Computation in the academic year 2017-2018, and so had not been formally taught session types.
 - 2 participants had taken Theory of Computation in the academic year 2018-2019, and so had been formally taught session types.

Participants already familiar with π -calculus were used to make sure that the web app was not attempting to teach π -calculus in a way so different from how it is normally taught that it would confuse anyone who had previously learned it in those ways, and so it would still be usable by people already familiar.

The main aims of the user study were the following:

- Aim 1** Find out how well participants felt the user guide explains the concepts of typechecking, semantics and the encoding
- Aim 2** Find out how well participants felt the user guide had taught them about writing π -calculus code.
- Aim 3** Find out how the participants felt the web app had affected their understanding of session types and linear types
- Aim 4** Find out how the participants felt the web app had affected their understanding of π -calculus.
- Aim 5** Find out how the modified syntax might affect the usability of the site for those already experience with π -calculus.

The survey in the user study started by asking participants how familiar they already were with π -calculus and with session types and linear types. This was done so that the results could be considered in the same groups that participants were gathered in, as although we knew beforehand which group each participant was in, the results are stored anonymously. Charts of the responses to these introductory questions can be seen in Figure 5.1 and Figure 5.2.

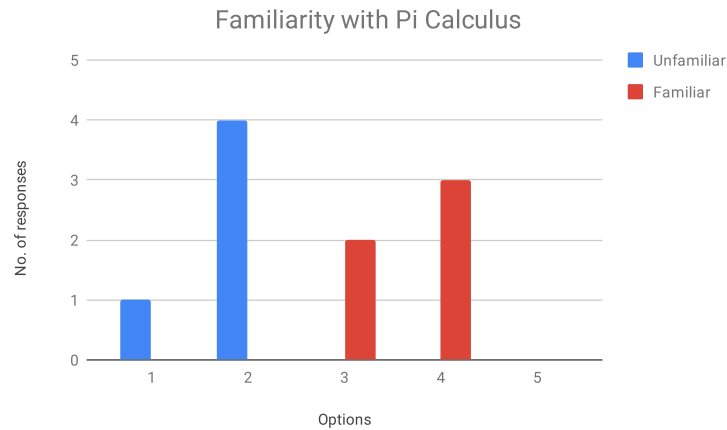


Figure 5.1: The responses to the question asking how familiar participants are with π -calculus. Options ranged from 1 to 5, 1 being labelled "Never heard of π -calculus" and 5 being labelled "Very familiar with π -calculus". As expected, all participants from the group not already familiar with π -calculus answered lower than all participants from the group already familiar.

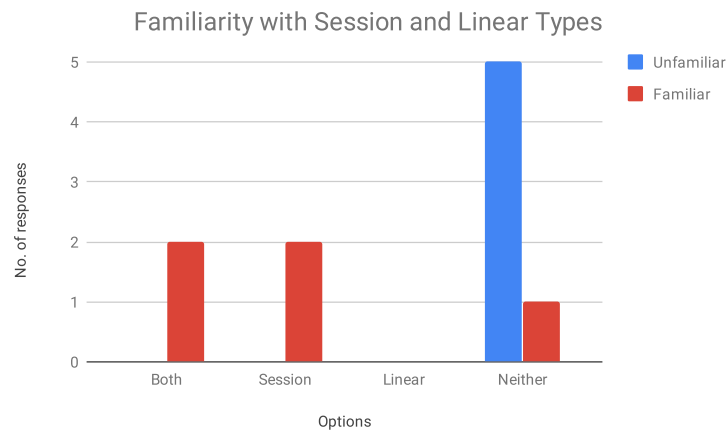


Figure 5.2: The responses to the question asking how familiar participants are with session types and linear types in π -calculus. As expected, no participants in the group unfamiliar with π -calculus were familiar with session types or linear types in π -calculus. Those already familiar with π -calculus were a mixture of people familiar and unfamiliar with session types or linear types.

The survey then prompted the participants to read the web app's user guide, and then perform three tasks in the web app. The survey did not record any information directly related to the participant's performance in these tasks. The first of these tasks was to examine an example program and consider how this program would typecheck, execute and encode, then to perform all three of these actions in the web app and compare the results to what they expected. This was designed to achieve **Aim 1**.

The second task gave them an incomplete example program containing a `branch` process with three branches, and asked the participants to write processes which would act as the counterpart to each branch. This was designed to achieve **Aim 2**.

The third task gave an example program which did not work and would throw errors when typechecked, and asked users to identify why. It then asked them to correct the issue with the program, so that it can successfully be typechecked, executed and encoded. This task was also designed to achieve **Aim 2**, but by checking their understanding of what is not valid π -calculus, rather than what is.

To gather the participants opinions on the web app, once they had finished all tasks, the survey asked them questions on how easy or difficult they found it to understand the user guide, understand the example programs, write π -calculus code and correct π -calculus code. It then also asked them how they felt the user study had affected their understanding of session types and linear types, and of π -calculus, to achieve **Aims 3 and 4**. Charts of the responses to these questions can be seen in Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8.

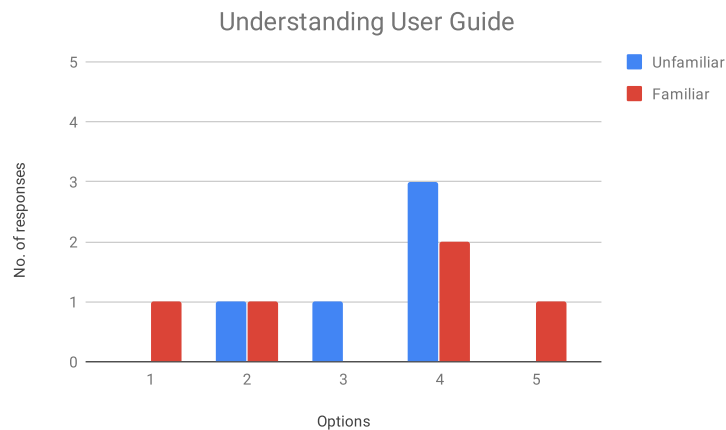


Figure 5.3: Responses to the question asking how easy or difficult participants found it to understand the web app's user guide. Options ranged from 1 to 5, 1 being "Very difficult", 5 being "Very easy". Results are mixed, leaning towards the user guide being easy to understand, showing that it is generally helpful but could use improvement.

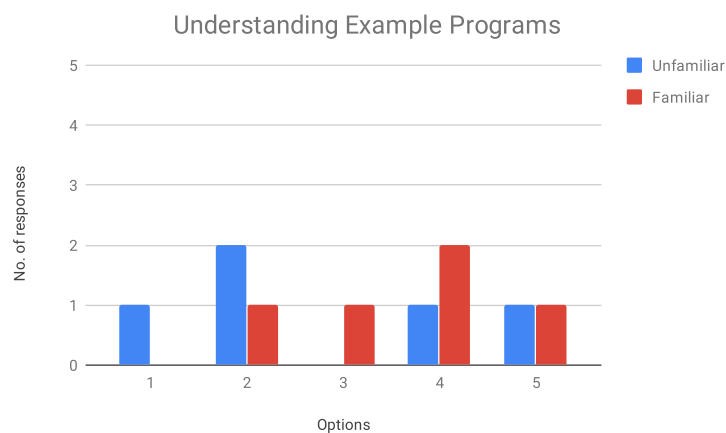


Figure 5.4: Responses to the question asking how easy or difficult participants found it to understand the examples programs from the survey. Options ranged from 1 to 5, 1 being "Very difficult", 5 being "Very easy". Results are mixed, showing that the syntax guide may not be sufficiently clear.

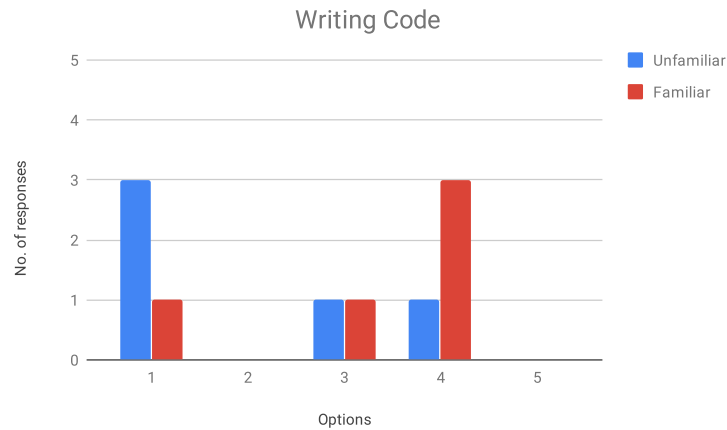


Figure 5.5: Responses to the question asking how easy or difficult participants found it to write their own π -calculus code. Options ranged from 1 to 5, 1 being "Very difficult", 5 being "Very easy". Many participants not already familiar with π -calculus struggled with this, showing that the user guide may not explain coding sufficiently well.



Figure 5.6: Responses to the question asking how easy or difficult participants found it to correct faulty π -calculus code. Options ranged from 1 to 5, 1 being "Very difficult", 5 being "Very easy". Results are largely negative. However, we think this may be due to the specific example used, as the flaw in the code was a channel restriction only encompassing two out of the three composed processes. This flaw is identified by the placement of parentheses, and so is not easily noticeable. This example therefore may not have been the most suitable, and had an example with a less subtle flaw been used, the results of this question may have been different.

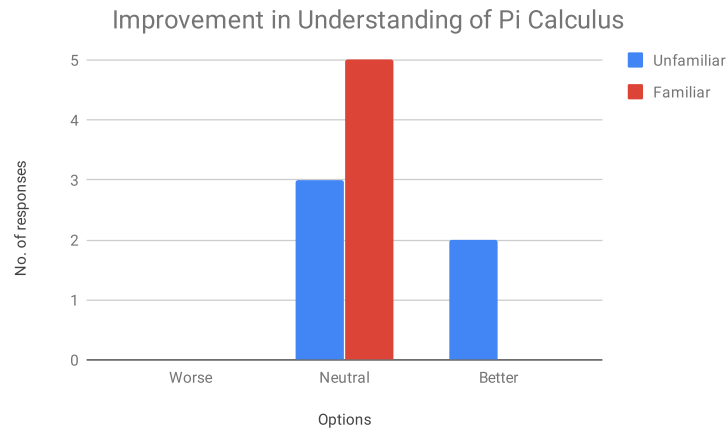


Figure 5.7: Responses to the question asking how the web app has affected the participant's understanding of π -calculus. As expected, those already familiar did not feel they had learned anything new. Some of those unfamiliar did feel they had learned about π -calculus, but more did not feel any more knowledgeable, showing that the web app may not be as effective as a teaching tool as hoped.

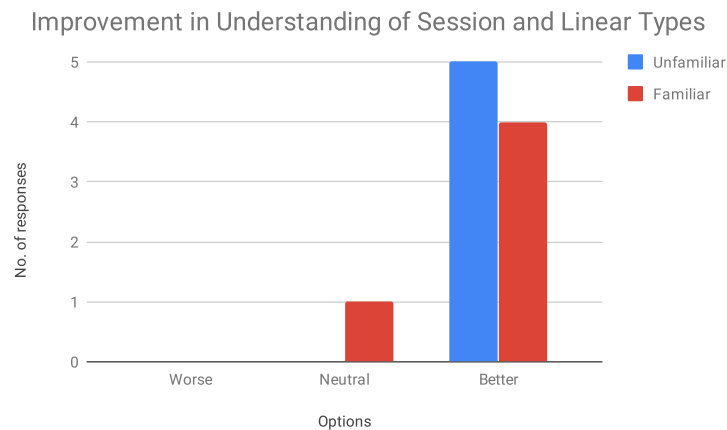


Figure 5.8: Responses to the question asking how the web app has affected the participant's understanding of session types and linear types. Almost all participants felt their understanding had been improved, showing that the web app is effective in teaching the concepts of session and linear types.

Participants already familiar with π -calculus were also asked for comments on the web app's syntax, in comparison to the standard syntax. The responses are as follows:

- "I personally prefer the standard syntax, as it's easier for me to read matching communications"
- "Both standard picalc syntax and the syntax of the webapp's picalc are VERY difficult to read due to how visually dense they are, plus how horizontal they are. I found myself constantly having to scroll horizontally to read various parts of the code, which hindered my understanding."
- "Seems a little bit more annoying to write with"
- 2 participants did not give any comment.

This question was used to achieve **Aim 5**. As can be seen, the web app's syntax was not preferable to the standard syntax among those who chose to comment. **This result is unsurprising, as the syntax modification was made with new users in mind.**

Overall, the results show that **while the web app succeeded in teaching most participants about session types and linear types, it was less successful in teaching them about π -calculus itself.** Additionally, while the user guide was generally helpful, there is room for improvement. As most of the troubles participants had appears to be with code, these improvements would likely come in the form of more detailed explanations of the processes, and perhaps including an example program in the user guide to help illustrate how the code is to be formed.

In hindsight, while the user study did give valuable information on the effectiveness of the web app as a teaching tool, more information could've been gathered with better questions. For example, while participants with experience of π -calculus were asked for comments on the syntax, inexperienced participants were not. Asking these users about the syntax could have been a valuable way of finding out whether the web app's syntax is in fact easier to understand, as is hoped. Additionally, some participants commented on the lack of a question asking for general comments about the web app. This could have given us an idea of why, for example, they found the user guide difficult to understand, and thus how it can be improved to be more helpful.

As none of the participants had been formally taught of the encoding, there was little focus on this in the user study, being mentioned only in the first task. In hindsight, this is perhaps something that should have been more prominent in the evaluation, as it is a major focus of the overall project.

A signed ethics checklist, denoting that the user study complied with the ethics requirements of the University of Glasgow's School of Computing Science, can be found in Appendix A.2. The introduction and debriefing scripts used in the user study can be found in Appendices A.3 and A.4 respectively.

6 | Conclusion

This project set out to develop a tool which could be used both as a teaching tool for π -calculus, session types, and linear types and as a tool to be used by people familiar with these concepts to aid them in using the encoding from session types to linear types. In those senses, the project has been reasonably successful. It has been shown through our user study that our tool can be helpful as a teaching tool for π -calculus, though not as much as hoped, and for session types and linear types. The project is also capable of automatically carrying out the encoding of session-typed π -calculus into linear-typed π -calculus, although the modified syntax of the tool, aimed at new users of π -calculus, may affect its usefulness for experienced users.

6.1 Related Works

Fuse is a library, detailed in "A simple library implementation of binary sessions" (Padovani 2017), which implements session types into OCaml. FuSe uses the encoding from session types to linear types to simplify its typechecking, in particular duality checking, while still using the semantics of session types. That is, while the types are represented with the continuation-passing principle, the semantics does not involve the creation or communication of any continuation channels. The encoding is used for types to allow a channel to be represented as a pair of capabilities, $\langle \alpha, \beta \rangle$, where α is the type the channel is capable of receiving and β is the type the channel is capable of sending. For the input and output session types, one of these is empty while the other is a product type containing the type of the payload, and the type of the continuation channel as used by the recipient. As a result, the dual of type $\langle \alpha, \beta \rangle$ can be expressed $\langle \beta, \alpha \rangle$, simplifying duality checking to a type equality check. That is, $\langle \delta, \gamma \rangle$ is the dual of $\langle \alpha, \beta \rangle$ if $\delta == \beta$ and $\gamma == \alpha$. As a result of this method of representing session types, using FuSe with a session-typed pi calculus program would require that you encode all of the types in said program to obtain their representation in FuSe. As our tool can provide the automatic encoding of types, it can be used to simplify this preparation of the session-typed program for use in FuSe.

"Lightweight Session Programming in Scala" (Scalas and Yoshida 2016) details a method of using session type methodologies in Scala via a library `lchannels`, also described in the paper. It makes use of the encoding of session types into linear types as an intermediary step of converting session types into Scala types that use the continuation-passing principle. `lchannels` differs from FuSe in that the use of the encoding is applied not just to the types but also to the semantics, with continuation channels being created and exchanged. As with FuSe, however, the automatic encoding of types from our tool could be used to assist in programming in this library, as it again removes the effort in manually transforming types in session-typed programs into the format used by the library.

"A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming" (Scalas et al. 2017) describes the methodology behind (and an implementation of) an encoding from multiparty session types to linear types. This encoding makes use of the concepts involved in the encoding from session types to linear types for the encoding of the projections of a multiparty session type onto its roles (a partial type representing the interactions a session endpoint has with

a particular other endpoint in the session). The overall multiparty session type is encoded as a labelled tuple containing the encodings of these projections.

6.2 Future Work

Future work on the project would involve both improvement of existing aspects, and addition of new aspects as an enhancement on the project overall. In terms of improvements to existing aspects, this would mostly take the form of improving the clarity and understandability of the web app. As the evaluation found, the user guide was not entirely helpful and easy to understand for all participants, so future work would most likely start with improvements to the user guide, such as example code as mentioned near the end of Section 5.1.

Additionally, other aspects were found to be unclear, for example, we noticed during the evaluation that some error messages the web app can return for invalid π -calculus code can be unclear or misleading. For example, when typechecking finds a process on a channel of the incorrect type, e.g. an output process whose channel has an input type, the web app produces an error message stating "Output process on channel not of output type.". This message is suitable for the example situation given, however, we have realized that this error message is also produced when typechecking finds a process on a channel not present in the typing context, i.e. a channel which does not exist. This gives the user the impression of an issue entirely different from what is causing typechecking to fail. In particular, this error message would be of no help when the issue is that, for example, the process attempting to use a channel is not within the restriction of that channel.

There are a few additional features which could be added to the project as future work. For example, as mentioned in Section 1.1, the project's automatic encoding produces linear-typed π -calculus that could be used with existing tools, with some intermediate link between the tools. This link could be considered for future work. The link could take the form of a conversion from code written in the web app's syntax into code written in the syntax of other existing tools for π -calculus, or perhaps even a setting for the web app to work entirely in these other syntaxes, although this would be significantly more work than just a conversion.

During development of the project, there were some features that were intended to be implemented, but were not. One of those was syntax highlighting of the user's code. This was attempted near the end of development, using CodeMirror. (CodeMirror Team) CodeMirror is a JavaScript library for creating in-browser text editors specialized for programming. One of the features of CodeMirror is syntax highlighting, for which it has many modes for different programming languages. As our project defined its own language, these modes were not helpful to us, and we would have to define a custom mode. Defining a full CodeMirror mode would have been complicated, and too large an undertaking for the amount of time left to develop the project, so we instead opted to try using the SimpleMode addon, which allows users to define a simple, less powerful CodeMirror mode using regular expressions. This was attempted, but was found to also be complicated and did not give the desired results. Due to this, the idea of syntax highlighting was dropped from the project. Making another attempt at it, this time by defining a full custom mode, could be a potential avenue for future work.

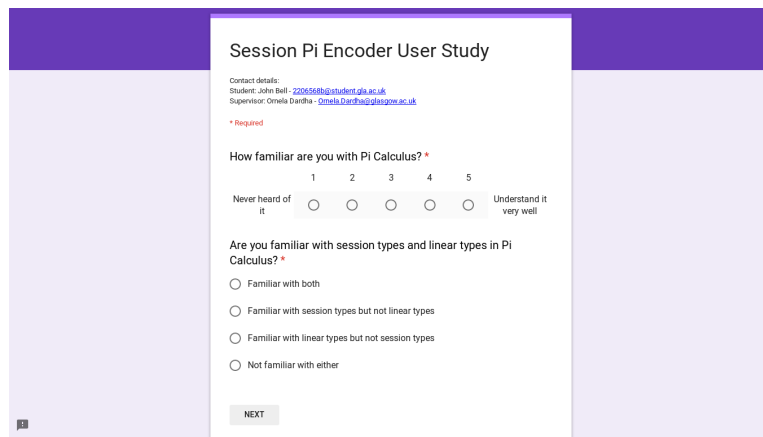
Another feature that was considered during development was the importing and exporting of files. The idea was that each of the input areas of the web interface would have another two buttons underneath, Import and Export. Import would allow the user to upload a text file's contents directly into the corresponding input area, and Export would save the current contents of the corresponding input area to a text file on the user's machine. This would allow users to more easily save processes for future use. This feature was considered a low priority and simply

did not get implemented before development time on the project had run out. As such, this could be considered for future work.

As mentioned in Section 5.1, the user study performed on the web app was in some ways lacking. After implementing some of the mentioned future work, it would be worth considering performing another, improved user study to find out both how the web app has improved in the ways that were already evaluated, and to evaluate it in ways that were overlooked in the first user study. For example, this second user study could contain questions asking how much syntax highlighting helps the readability of code, and test how well the user guide teaches the concepts behind the encoding.

A | Appendices

A.1 User Study Survey



The screenshot shows a survey form titled "Session Pi Encoder User Study". It includes contact details for the student and supervisor, a required field for familiarity with Pi Calculus, and a question about familiarity with session types and linear types in Pi Calculus. A "NEXT" button is visible at the bottom.

Session Pi Encoder User Study

Contact details:
Student: John Bell - 2206568@student.gla.ac.uk
Supervisor: Ornela Danthar - Ornela.Danthar@glasgow.ac.uk

* Required

How familiar are you with Pi Calculus? *

1 2 3 4 5
Never heard of it Understand it very well

Are you familiar with session types and linear types in Pi Calculus? *

Familiar with both
 Familiar with session types but not linear types
 Familiar with linear types but not session types
 Not familiar with either

NEXT

Figure A.1: The introductory questions of the user study survey.

Session Pi Encoder User Study

Tasks

Before performing these tasks, read the web app's user guide.

Task 1

Look at the following program:

```
type v sint = 5
(new x y :!sint ?!sBool end) ( { send(x, v).receive(x, isZero : sBool) stop | receive(y, num : sint) send(y, (num == 0)) stop } )
```

Think about what this program would do when executed. (Both overall, and in terms of reductions.)
Think about what typing rules would be used to typecheck this program.
Think about roughly how the encoding of this program might look.
Copy the code into the Session-Typed Pi Calculus text area in the web app.
Typecheck the program and examine the output, comparing it to what you expected.
Execute the program and examine the output, comparing it to what you expected.
Encode the program and examine the encoding, comparing it to what you expected.

Task 2

Look at the following program:

```
(new x y :!s(alpha : ?!sint :!sint.end, beta : ?!String :!String :!String.end, gamma : ?!sBool :!sBool :!sBool.end)) (
  {
    branch(x) {
      alpha : receive(x, i1 : sint) receive(x, i2 : sint) send(x, (i1 - (i1 % i2)) + i2) stop
      beta : receive(x, s1 : sString) receive(x, s2 : sString) send(x, (s2 ++ (s1 ++ s2))) stop
      gamma : receive(x, b1 : sBool) receive(x, b2 : sBool) send(x, ((NOT b1) OR b2)) stop
    }
  }
  Q
)
```

This code offers three options:
given two integers, return the first, rounded down to a multiple of the second
given two strings, return the first, prepended and appended with the second
given two booleans, return whether the first implies the second
In place of Q, write counterparts to each branch and check that they can typecheck, execute and encode.

Task 3

Look at the following program:

```
type v sint = 10
(new x y :!sint :!sint.end) (
  {
    (new z :!sBool) (
      receive(x, w : sint) send(z, (w == 10)) send(x, (w * w)) stop
      receive(z, atLeast10 : sBool) receive(z, atMost200 : sBool) send(z, (atLeast10 AND atMost200)) stop
    )
  }
  send(y, v) receive(y, wSpr : sint) send(z, (wSpr == 200)) receive(z, result : sBool) stop
)
```

This program doesn't work. Its intention is that for some integer v , it finds out whether v is a number higher than or equal to 10, whose square is less than or equal to 200.
Examine the code and find out why it doesn't work.
Fix this program.
As an added challenge, try to fix the program without moving the channel restriction on z . (Hint: The type of x and y must be extended.)

Never submit passwords through Google Forms.

Figure A.2: The second section of the survey, containing the tasks to be completed in the web app.

Session Pi Encoder User Study

* Required

How easy or difficult did you find it to understand the user guide? *

1 2 3 4 5

Very difficult to understand Very easy to understand

How easy or difficult did you find it to understand the programs in the tasks? *

1 2 3 4 5

Very difficult to understand Very easy to understand

How easy or difficult did you find it to write your own pi calculus code? *

1 2 3 4 5

Very difficult to write Very easy to write

How easy or difficult did you find it to change provided pi calculus code? *

1 2 3 4 5

Very difficult to change Very easy to change

If you are already familiar with pi calculus, did you have any thoughts on the web app's syntax compared to the standard syntax (e.g. $x! \leftarrow y?(w)$)

Your answer _____

How would you say this web app has affected your understanding of session types and linear types in pi calculus? *

I have a better understanding of them now.

My understanding of them is roughly the same as it was before.

The web app has confused me and made my understanding worse.

How would you say this web app has affected your understanding of pi calculus? *

I have a better understanding of it now.

My understanding of it is roughly the same as it was before.

The web app has confused me and made my understanding worse.

Never submit passwords through Google Forms.

Figure A.3: The final section of the survey, containing the questions used to gather the views of the participants.

A.2 User Study Ethics Checklist

School of Computing Science
University of Glasgow

Ethics checklist for 3rd year, 4th year, MSci, MRes, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please submit an ethics approval form to the Department Ethics Committee.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

-
1. Participants were not exposed to any risks greater than those encountered in their normal working life.
Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback
 2. The experimental materials were paper-based, or comprised software running on standard hardware.
Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, mobile phones, and PDAs is considered non-standard.
 3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.
If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.
 4. No incentives were offered to the participants.
The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.
6. No participant was under the age of 16.
Parental consent is required for participants under the age of 16.
7. No participant has an impairment that may limit their understanding or communication.
Additional consent is required for participants with impairments.
8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.
9. All participants were informed that they could withdraw at any time.
All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.
10. All participants have been informed of my contact details.
All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.
11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.
12. All the data collected from the participants is stored in an anonymous form.
All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title Encoding Session Types into Linear Types in π -Calculus
 Student's Name John Bell
 Student's Registration Number 2206568
 Student's Signature John Bell
 Supervisor's Signature Drusela Dardas
 Date 29/3/19

Figure A.4: The ethics checklist for the user study.

A.3 User Study Introduction Script

"The aim of this is to determine how usable and helpful this web app is for teaching people about π -Calculus and the encoding from session types to linear types. To do this, we need to show it to people and see how much it helped them understand those concepts. The survey starts with a few questions to work out how much you know already, then gives you a few tasks to complete on the web app and finally asks you some questions on how you felt about it. We're only recording your answers to the questions, not your performance in the tasks, so don't worry about how you do in those. Feel free to ask questions or withdraw at any point, and let me know when you're done. Do you agree to take part? And do you have any questions before starting?"

A.4 User Study Debriefing Script

"Like I said before starting, the aim of this survey was to determine how usable and helpful this web app is in teaching people about π -calculus and the encoding from session types to linear types. Do you have any questions about any aspects of the survey? My email, and my supervisor's email are both on the form, so if you think of any questions, you can reach us through those. Thank you for helping."

Bibliography

- CodeMirror Team. CodeMirror. <https://codemirror.net/>, 2011. Accessed : 25/3/2019.
- O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 139–150, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1522-7. doi: 10.1145/2370776.2370794. URL <http://doi.acm.org/10.1145/2370776.2370794>.
- O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. *Information and Computation*, 256:253 – 286, 2017. ISSN 0890-5401. doi: <https://doi.org/10.1016/j.ic.2017.06.002>. URL <http://www.sciencedirect.com/science/article/pii/S0890540117300962>.
- K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-69722-0.
- N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, Sept. 1999. ISSN 0164-0925. doi: 10.1145/330249.330251. URL <http://doi.acm.org/10.1145/330249.330251>.
- R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science. Springer-Verlag, 1980. ISBN 9783540102359. URL <https://books.google.co.uk/books?id=7L1PAQAATAAJ>.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992. ISSN 0890-5401. doi: [https://doi.org/10.1016/0890-5401\(92\)90008-4](https://doi.org/10.1016/0890-5401(92)90008-4). URL <http://www.sciencedirect.com/science/article/pii/0890540192900084>.
- M. Odersky. Polarized name passing. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 324–337, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49263-4.
- L. Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27:e4, 2017. doi: 10.1017/S0956796816000289.
- T. Parr. ANTLR. <https://www.antlr.org/>, 2014. Accessed : 20/3/2019.
- B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996. doi: 10.1017/S096012950007002X.
- Pocoo Team. Foreword – Flask 1.0.2 Documentation. <http://flask.pocoo.org/docs/1.0/foreword/>, 2010. Accessed : 20/3/2019.
- Python Software Foundation. Welcome to Python.org. <https://www.python.org/>, 2001. Accessed : 26/3/2019.

- D. Sangiorgi. An interpretation of typed objects into typed π -calculus. *Information and Computation*, 143(1):34 – 73, 1998. ISSN 0890-5401. doi: <https://doi.org/10.1006/inco.1998.2711>. URL <http://www.sciencedirect.com/science/article/pii/S0890540198927110>.
- A. Scalas and N. Yoshida. Lightweight Session Programming in Scala. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:28, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.21. URL <http://drops.dagstuhl.de/opus/volltexte/2016/6115>.
- A. Scalas, O. Dardha, R. Hu, and N. Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In P. Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-035-4. doi: 10.4230/LIPIcs.ECOOP.2017.24. URL <http://drops.dagstuhl.de/opus/volltexte/2017/7263>.
- V. T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52 – 70, 2012. ISSN 0890-5401. doi: <https://doi.org/10.1016/j.ic.2012.05.002>. URL <http://www.sciencedirect.com/science/article/pii/S0890540112001022>.