University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Session Types in π-calculus

Andrew Mcnab

March 26, 2019

Abstract

The π-calculus is a formal computational model for describing communicating and concurrent processes, which has recently had a resurgence in popularity. Session types extend the π-calculus with a type system. When students are taught the π-calculus with session types, they rely on evaluating processes by hand, which is cumbersome. This work explores a language based on the π-calculus, as well as an interpreter and type system implementation for it, which could be used in teaching. Evaluation of the work showed that the language has promise for teaching, although the interpreter could be made more user friendly.

Acknowledgements

I would like to thank my project supervisor, Dr Ornela Dardha, for her support and guidance during the project.

Education Use Consent

Contents

1 Introduction **1**
1.1 Motivation and Objectives 1
1.2 Summary 1

2 Background **2**
2.1 The π-calculus 2
2.1.1 Syntax 2
2.1.2 Semantics 2
2.2 Session Types 4
2.2.1 Syntax 4
2.2.2 Duality 5
2.2.3 Typing rules 5
2.2.4 Examples of the π-calculus With Session Types 6
2.3 ANTLR 7

3 Requirements **9**
3.1 Functional Requirements 9
3.2 Non-functional Requirements 9

4 Design **11**
4.1 Language 11
4.2 Interpreter 12
4.3 Type System 12

5 Implementation **13**
5.1 Language 13
5.2 Interpreter 13
5.3 Type System 17

6 Evaluation **20**
6.1 User study 20
6.2 Correctness 20
6.3 Usability 21
6.4 Effectiveness as a Learning Tool 22
6.5 Related Work 22
6.5.1 pi-calculus 22
6.5.2 JsonPi 22
6.5.3 SePi 23
6.5.4 GV 23

7 Conclusion **25**
7.1 Summary 25
7.2 Future work 25

Appendices **26**

A Appendices **26**
A.1 Pi-calc Grammar 26
A.2 User Guide 28
A.3 Ethics Approval 29
A.4 User Study Questionnaire 29

Bibliography **33**

List of Figures **34**

1 Introduction

1.1 Motivation and Objectives

When the π-calculus is taught to students, they have to rely on evaluating processes by hand. This can be tedious for any example that is non-trivial, and is prone to errors. This also goes for teaching session types, as writing out a type checking by hand can be time consuming. Because of these reasons, an interpreter for the π-calculus with a session type based type system could be used in helping students learn these topics.

The objectives of this work can be split in to two main sections. Firstly, to implement the π-calculus as a programming language which is executable through software. This will include a specification of the language, as well as an implementation of the language using an interpreter to execute code written by a user. Secondly, the extension of session types will be added, meaning implementing a type system within the implemented language. In addition to these aims, significant examples will be constructed to demonstrate the language's features and its correctness.

1.2 Summary

This chapter motivated the project, and set out the aims of the work. The remainder of this dissertation is structured as follows:

- **Chapter** 2 gives some background on the theory and tools used in this work, as well as defining some examples that will be shown in the implementation later.
- **Chapter** 3 outlines the requirements of the work.
- **Chapter** 4 describes the design of the work, expanding on the requirements outlined in Chapter 3.
- **Chapter** 5 describes the implementation of the language, interpreter, and type system.
- **Chapter** 6 evaluates the language, interpreter, and type system, and discusses if the aims of the work were met.
- **Chapter** 7 outlines potential future work.

2 Background

2.1 The π-calculus

2.1.1 Syntax

The π-calculus is a formal computational model for describing communicating and concurrent processes. The π-calculus was originally developed by Robin Milner, building on his previous work on the Calculus of Communicating Systems (CCS) (Milner 1982). It has a very simple construction, and its syntax can is described by the following Backus-Naur Form (BNF) grammar:

$$
\begin{array}{rlr}
P, Q ::= & \bar{x}\langle y\rangle.P & \text{(output)} \\
\mid & x(y).P & \text{(input)} \\
\mid & P \mid Q & \text{(composition)} \\
\mid & (\boldsymbol{\nu} x)\, P & \text{(channel restriction)} \\
\mid & (\boldsymbol{\nu} xy)\, P & \text{(new session)} \\
\mid & x \triangleright \{l_i : P_i\}_{i \in I} & \text{(branch)} \\
\mid & x \triangleleft l_j.P & \text{(select)} \\
\mid & 0 & \text{(inaction)}
\end{array}
$$

Figure 2.1: *Syntax of π-calculus processes with session types (Dardha et al. 2017, Fig.1).*

In more detail, the input rule can be read in the following way:

- On channel x, receive some information
- Bind the result to y
- Continue to run P

Similarly for output, we send the value y on channel x, and continue to run P. The composition rule allows us to run two processes P, Q in parallel. Channel restriction $(\boldsymbol{\nu} x)\, P$ creates a new channel, and restricts its scope to the process P. The branch, select, and new session processes are included with session types. Branch allows us to define choices between processes, with each process having a label, l. We can select a label with the select process, which executes the selected process.

2.1.2 Semantics

The π-calculus has the prominent notion of *names*, which in Figure 2.1 are the symbols x, y. We call x and $\bar{x}$ *co-names*. In interactive behaviour, x and $\bar{x}$ can be thought of as complimentary

$$\bar{x}\langle v\rangle.P \mid x(z).Q \rightarrow P \mid Q[v/z] \tag{R-StndCom}$$
$$(\nu xy)\ (\bar{x}\langle v\rangle.P \mid y(z).Q \rightarrow (\nu xy)\ (P \mid Q[v/z]) \tag{R-Com}$$
$$(\nu xy)\ (x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i \in I}) \rightarrow (\nu xy)\ (P \mid P_j),\ \ j \in I \tag{R-Case}$$
$$P \rightarrow Q \implies (\nu x)\ P \rightarrow (\nu x)\ Q \tag{R-StndRes}$$
$$P \rightarrow Q \implies (\nu xy)\ P \rightarrow (\nu xy)\ Q \tag{R-Res}$$
$$P \rightarrow Q \implies P \mid R \rightarrow Q \mid R \tag{R-Par}$$

Figure 2.2: *Semantics of the π-calculus with session types.*

actions, each being one side of a communication between two agents. Together, x and $\bar{x}$ represent a *channel*, which allow communication between multiple agents.

The semantics of the language is defined by the relation $\rightarrow$ over processes, which we call the *reduction* relation. If $P \rightarrow Q$, then we say *P reduces to Q*. The semantics of the π-calculus are defined using the reduction relation in Figure 2.2. The rule (R-StndCom) defines communication between channels, where if we have a matching send and receive in parallel, communication occurs. The notation $Q[v/z]$ means that we replace all occurrences of z in Q with v. For example, the process $\bar{x}\langle 1\rangle.0 \mid x(y).0$ is a composition of two processes, which allows communication by (R-StndCom). In this example, the complimentary send and receive communicate, and the value 1 is bound to the name y. In this case, we write the following to notate the reduction (and therefore communication):

$$\bar{x}\langle 1\rangle.0 \mid x(y).0 \rightarrow 0 \mid 0$$

An important feature of the π-calculus is *mobility*. Mobility is the concept of sending channels, instead of specifically values. Mobility was a major addition to the π-calculus (Milner 1999) from CCS (Milner 1982). A simple example of this is the process:

$$\begin{aligned} & a(x).\bar{x}\langle 5\rangle.0 \mid \bar{a}\langle b\rangle.b(y).0 \\ \rightarrow\ & \bar{b}\langle 5\rangle.0 \mid b(y).0 \\ \rightarrow\ & 0 \end{aligned}$$

As we can see, the first send over channel a sends the channel b, which is then used for the second communication.

It should be noted that communication is the fundamental semantic of the calculus, as it allows the transmission of information, but also permits variables through *binding*. For example, the in reduction $\bar{x}\langle 3\rangle.0 \mid x(y).\bar{x}\langle y\rangle.0 \rightarrow 0 \mid \bar{x}\langle 3\rangle.0$, we first send the value 3 over channel x, which is then bound to the name y. By binding the value 3 to y, we can then use it in a later part of the process, in this example sending it over x.

Branches give the option of choice between any number of processes, e.g. a math server might have branches *add*, *sub*, and *isEqual*; where each branch provides different behaviour. The select process is used to choose from one of these branches, the semantic for branch and select is $(\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i \in I}) \rightarrow (\nu xy)(P \mid P_j),\ j \in I$. It should be noted that the process $(\nu xy)\ P$ binds the channels x, y, so that communication occurs between them, instead of requiring matching names in the non-extended π-calculus. As an example, $(\nu xy)\ [\bar{x}\langle 3\rangle.0 \mid y(z).0]$ communicates, with $z = 3$ after communication. We can read the new session notation as declaring x and y as dual endpoints.

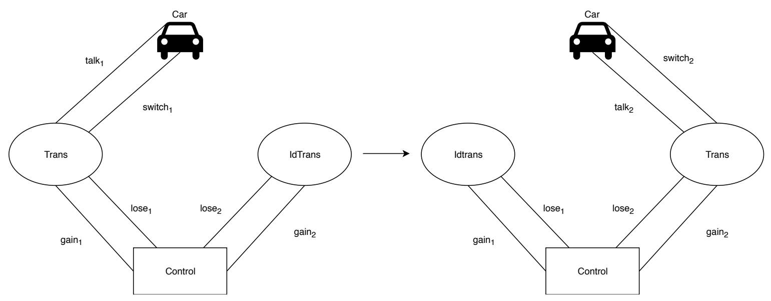


Figure 2.3: *The "car" system*

A significant example of the π-calculus and mobility is the "car" system, as specified by Milner (1999). The system models a car, which communicates with a transmitter. The car can switch to a secondary transmitter, or talk with the primary transmitter. This system can be seen in the left side of Figure 2.3. We define the system by each of its components:

$$\begin{aligned}
Trans(talk, switch, gain, lose) &:= talk.Trans\langle talk, switch, gain, lose\rangle \\
&\quad + lose(t, s).\overline{switch}\langle t, s\rangle.Idtrans\langle gain, lose\rangle \\
Idtrans(gain, lose) &:= gain(t, s).Trans\langle t, s, gain, lose\rangle \\
Control_1 &:= \overline{lose_1}\langle talk_2, switch_2\rangle.\overline{gain_2}\langle talk_2, switch_2\rangle.Control_2 \\
Control_2 &:= \overline{lose_2}\langle talk_1, switch_1\rangle.\overline{gain_1}\langle talk_1, switch_1\rangle.Control_1 \\
Car(talk, switch) &:= \overline{talk}.Car\langle talk, switch\rangle \\
&\quad + switch(t, s).Car\langle t, s\rangle
\end{aligned}$$

And the system together as a process of these components in parallel:

$$\begin{aligned}
System_1 := new\ & talk_1, switch_1, gain_1, lose_1, talk_2, switch_2, gain_2, lose_2 \\
& (Car\langle talk_1, switch_1\rangle \mid Trans_1 \mid Idtrans_2 \mid Control_1)
\end{aligned}$$

where $Trans_i := Trans\langle talk_i, switch_i, gain_i, lose_i\rangle$ and $Idtrans_i := Idtrans\langle gain_i, lose_i\rangle$.

The transition shown in Figure 2.3 shows the car switching between the two transmitters, by using mobility to send the new channels it would like to communicate on to the transmitter.

2.2 Session Types

2.2.1 Syntax

Session types (Vasconcelos 2012) specify the type and direction for communications in processes. The BNF grammar for session types is specified in Figure 2.4. The syntax of types are categorised into two groups: session types, and standard π-types which include session types. The session type *end* is the type of a session endpoint which is terminated. The type $?T.S$ is the type of a session endpoint which receives a value of type T, and then continues with session type S. Similarly, $!T.S$ denotes an endpoint which sends a value of type T and continuing with type S. The branch session type is a set of labelled session types, allowing a choice between the labels. Select allows us to choose a single label from those specified in a branch type.

$$
\begin{array}{llr}
S ::= & end & \text{(termination)} \\
 & \mid\ ?T.S & \text{(input)} \\
 & \mid\ !T.S & \text{(output)} \\
 & \mid\ \oplus\{l_i : S_i\}_{i\in I} & \text{(select)} \\
 & \mid\ \&\{l_i : S_i\}_{i\in I} & \text{(branch)}
\end{array}
$$

$$
\begin{array}{llr}
T ::= & S & \text{(session type)} \\
 & \mid\ \#T & \text{(channel type)} \\
 & \mid\ \texttt{Unit} & \text{(unit type)} \\
 & \mid\ \ldots & \text{(other constructs)}
\end{array}
$$

Figure 2.4: *Syntax of session types (Dardha et al. 2017).*

2.2.2 Duality

An important idea in session types is *duality*. Duality encapsulates the idea of the complimentary actions in π-calculus, in a well-typed process, opposite endpoints (channels) will have types that are dual to each other. Duality is defined inductively on a session type, the duality relation is shown in Figure 2.5. From this definition, we can see that two channels that are dual to each other will communicate (the dual of a send is a receive).

$$
\begin{array}{rcl}
\overline{end} & \equiv & end \\
\overline{!T.S} & \equiv & ?T.\overline{S} \\
\overline{?T.S} & \equiv & !T.\overline{S} \\
\overline{\oplus\{l_i : S_i\}_{i\in I}} & \equiv & \&\{l_i : \overline{S_i}\}_{i\in I} \\
\overline{\&\{l_i : S_i\}_{i\in I}} & \equiv & \oplus\{l_i : \overline{S_i}\}_{i\in I}
\end{array}
$$

Figure 2.5: *The duality relation on session types (Dardha et al. 2017).*

2.2.3 Typing rules

Session types are either *linear* or *unrestricted*. A session type T is linear if and only if T is a session type, and $T \neq end$. If T is linear, we write $lin(T)$. If a type is not linear, it is unrestricted. We have $un(\Gamma) \iff \nexists\, T \in \Gamma$ such that $lin(T)$. The typing rules for the π-calculus with session types are given in Figure 2.6.

Session types provide some specification for a process, as the structure of the session type mirrors the structure of the process it is typing. Although the session type gives the structure of a process, the session type is for a channel, *not* a process. To illustrate this, let $x :: ?int.!int.end$, and $P = (\nu xy)[x(a).\bar{x}\langle a+1\rangle.0 \mid \bar{y}\langle 3\rangle.y(b).0]$. Here, channel x is given a session type, and channel y is given the dual of that type. As the process reduces, the session types for the channel change.

(T-Var) $\dfrac{un(\Gamma)}{\Gamma, x : T \vdash x : T}$ (T-Inact) $\dfrac{un(\Gamma)}{\Gamma \vdash 0}$ (T-Par) $\dfrac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q}$ (T-Res) $\dfrac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu xy)\, P}$

(T-StndRes) $\dfrac{\Gamma, x : T \vdash P \quad T \text{ is not a session type}}{\Gamma \vdash (\nu x)\, P}$ (T-In) $\dfrac{\Gamma_1 \vdash x : ?T.S \quad \Gamma_2, x : S, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}$

(T-Out) $\dfrac{\Gamma_1 \vdash x : !T.S \quad \Gamma_2 \vdash v : T \quad \Gamma_2, x : S, y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \bar{x}\langle v\rangle.P}$ (T-Brch) $\dfrac{\Gamma_1 \vdash x : \&\{l_i : T_i\}_{i \in I} \quad \Gamma_2, x : T_i \vdash P_i}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i \in I}}$

(T-Sel) $\dfrac{\Gamma_1 \vdash x : \oplus\{l_i : T_i\}_{i \in I} \quad \Gamma_2, x : T_j \vdash P \; \exists j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P}$

Figure 2.6: *Typing rules for the π-calculus with session types (Dardha et al. 2017).*

This is outlined in the typing rules, where each send or receive will remove the leading term in the session type.

2.2.4 Examples of the π-calculus With Session Types

Enough theory has been built up to demonstrate a significant example of the π-calculus with session types. Gay and Hole (2005) specified a mathematical server which offers a choice of different mathematical operations. In this case addition, subtraction, and equality checking are the options offered by the server. The session type for the server endpoint, x, is given by:

$$\begin{aligned} server := \&\{ \; & plus : ?int.?int.!int.end, \\ & sub : ?int.?int.!int.end, \\ & eq : ?int.?int.!bool.end \; \} \end{aligned}$$

The type of the client that will interact with the server is given by the dual of the server type:

$$\begin{aligned} \overline{server} := \oplus\{ \; & plus : !int.!int.?int.end, \\ & sub : !int.!int.?int.end, \\ & eq : !int.!int.?bool.end \; \} \end{aligned}$$

The server itself is defined by the following process:

$$\begin{aligned} server := x \triangleright \{ \; & plus : x(u).x(v).\bar{x}\langle u + v\rangle.0, \\ & sub : x(u).x(v).\bar{x}\langle u - v\rangle.0, \\ & eq : x(u).x(v).\bar{x}\langle u == v\rangle.0 \; \} \end{aligned}$$

There are many choices for a client, as we can select any of the three names given by the server, and send any integer values. A possible client is given by:

$$client := y \triangleleft plus.\bar{y}\langle 1\rangle.\bar{y}\langle 2\rangle.y(result : int).0$$

by the typing rule *T-Sel*, the client will be type checked against the dual of the type of the branch selected, in this case, $!int.!int.?int.end$. The interaction between the server and client can be seen by the following reductions:

$$\begin{aligned}
&(\nu xy)(server \mid client)\\
\rightarrow\ &(\nu xy)(server \mid y \triangleleft plus.\bar{y}\langle 1\rangle.\bar{y}\langle 2\rangle.y(result : int).0)\\
\rightarrow\ &(\nu xy)(x(u).x(v).\bar{x}\langle u+v\rangle.0 \mid \bar{y}\langle 1\rangle.\bar{y}\langle 2\rangle.y(result : int).0)\\
\rightarrow\ &(\nu xy)(x(v).\bar{x}\langle 1+v\rangle.0 \mid \bar{y}\langle 2\rangle.y(result : int).0)\\
\rightarrow\ &(\nu xy)(.\bar{x}\langle 1+2\rangle.0 \mid y(result : int).0) \rightarrow 0
\end{aligned}$$

2.3 ANTLR

ANTLR (Another Tool for Language Recognition) is a tool, which given an extended Backus-Naur form (EBNF) grammar, builds a lexer and a parser. As well as this, it provides a framework for building programming languages by providing access to the parse tree generated by the parser. It does this by building a *listener* (or alternatively, a visitor) which we can use with a *parse tree walker* to traverse the parse tree and define behaviour. The workflow of the tool can be summarised by the following steps:

1. Define a grammar
2. Generate the parser
3. Give the parser some input text
4. Using a listener or visitor, read the parse tree and define behaviour

An example of a parse tree can be seen in Figure 2.7, generated from the input text `a<1>.zero`. This is using a grammar defined for this project, which very closely follows the grammar of the π-calculus. The parse tree gives structure to the input, as defined in the grammar. Each node represents a *rule*, and the leaves of the tree are the *tokens* (text that represents some construct in the grammar, e.g. a variable name `myVariable`). In other words, the parse tree at a rule has children which are exactly the subrules and tokens of that rule. In Figure 2.7, the rule **send** has two child rules **chan**, **sendable**, and two child tokens < and >.

As previously mentioned, we can traverse this tree with a walker. A listener is a design pattern where we define a method for each rule, and these are called by the walker. ANTLR creates an abstract Listener for the grammar, and for every rule R in the grammar, it creates the methods `enterR(RContext ctx)` and `exitR(RContext ctx)`. The walker visits the parse tree in pre-order, and at each node calls the corresponding method. The behaviour of the language is then implemented in these methods. As an example, on entering a **send** node, we can implement the behaviour that manages the communication with respect to that send. At each node, ANTLR provides the *context* object which contains information about the current node, including the children of the node.

A way we can view ANTLR projects is that for a language, the grammar defines the *syntax*, and the listener implementation defines the language's *semantics*.

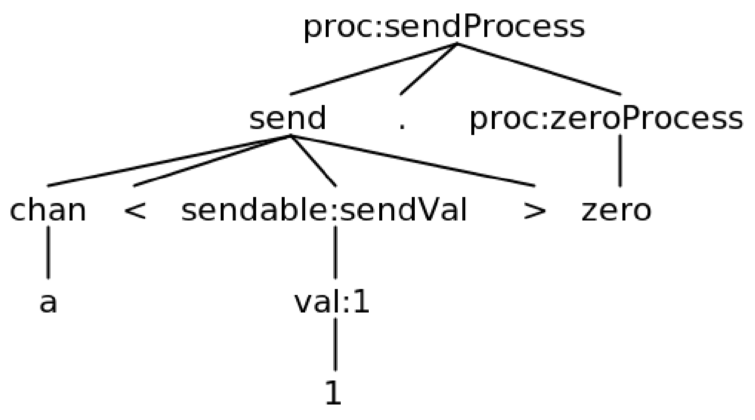


Figure 2.7: *An ANTLR parse tree generated from the π-calculus process "a<1>.zero"*

3 Requirements

The requirements for the project are categorised by the MoSCoW method, that is a requirement is assigned one of the four following categories:

- Must have
- Should have
- Could have
- Won't have

3.1 Functional Requirements

Must have The most important functional requirement is an interpreter for the base π-calculus. This means that given a file containing valid π-calculus, some software will execute it and provide output, showing the communications as they occur.

Should have The language implementation should support some additional constructs to make it more usable, such as `let` bindings. The implementation should also have a type system, based on session types (Dardha et al. 2017).

Could have The interpreter could step through each reduction, for example for an input program `a<3>.zero | a(y).zero` the following would be output:

```
a<3>.zero | a(y).zero
zero | zero            (send 3 on a)
zero                   (inaction)
end of process         (inaction)
```

Another feature that the project could have is support for the *polyadic* π-calculus, where multiple values can be sent/received in a single send/receive process, e.g. $\bar{a}\langle 1,\ 2,\ 3\rangle$, which is equivalent to the process $\bar{a}\langle 1\rangle.\bar{a}\langle 2\rangle.\bar{a}\langle 3\rangle$.

Won't have A web application UI was suggested, but due to scope restrictions on the project (particularly that it is worth 20 credits), this was not chosen as a requirement to implement.

3.2 Non-functional Requirements

Must have The interpreter must be semantically correct, valid π-calculus input must give the expected output, as per the specification of the π-calculus. The overall product must be usable, and accessible to anyone with a computing background - although not necessarily a background with π-calculus. A related requirement is that the output must be clear and easy to understand, as it is aimed as a learning tool.

Should have The interpreter should be portable, and a user should be able to run it on standard hardware. It should also be error-resilient, in that invalid input should be dealt with in a meaningful way rather than crash the program.

Could have The type system could have a structured output, showing the typing rules applied and using whitespace to illustrate the structure of the process.

Won't have Formally verifying the interpreter was discussed, by using a theorem prover such as Coq. This requirement was categorised as a 'won't have', as it would have added a significant amount of work without necessarily adding much to the project.

4 Design

4.1 Language

Language design starts with a grammar, defining the syntax. There already exist many slightly different grammars for the π-calculus, many of which are hard to write with a keyboard, for example the grammar defined in Figure 2.1 would require the user to type a bar over a channel name to send. An important part of the design of the language is that it should be easy for a user to type a program in any text editor. Because of this, a grammar requiring no special characters should be used.

The base π-calculus is very limited, only covering communication, parallel processes, and scope restriction. To make the language usable, additional constructs need to be added, thus extending the π-calculus. This is a big step towards a normal programming language, as it allows the user to write complex programs in a clearer way. Constructs added include:

- `let` bindings, and the `do` keyword to define and run processes respectively.
- a non-deterministic choice operator `P + Q`, which runs *either* `P` or `Q`.
- brackets, e.g. `[a<3>.zero] + [a<5>.zero]`.
- replication `!P`, which infinitely repeats a process `P` until no communications occur.
- a `show` statement, which takes a variable as an argument and prints the value of it to the console.

Despite adding these constructs and making the π-calculus more abstract, an important design decision was to not make it *too* abstract. In making the language too abstract, we might make it too hard for those new to π-calculus, and therefore ineffective as a learning tool. The language is intended to be as pure to the base π-calculus as possible, whilst still being usable as a programming language.

The semantics of the language should be identical to the semantics in Figure 2.2. That is, given any of the rules, the interpreter should behave in the same way. The chosen syntax for session types is inspired from Dardha et al. (2017), which can be seen in the math server example.

```
server :: &{ plus :: ?int.?int.!int.end
           , eq   :: ?int.?int.!bool.end }

let server(x) = -> { plus: x(u).x(v).x<u+v>.zero
                   , eq:   x(u).x(v).x<u==v>.zero }

let client(y) = select equal.y<3>.y<5>.y(eq :: bool).zero

(new x y)[do server(x) | do client(y)]
```

4.2 Interpreter

The interpreter must take a *.pi* file, and interpret it. This means that it must satisfy the semantics outlined in figure 2.2. The interpreter must give meaningful output for the interpreted file, which in the context of this work means showing the communications that occur, with the channel the communication occurred on as well as the data communicated. An example output for the input text `a<3>.zero | a(x).zero` is shown in the following snippet:

```
--> received on channel a: val 3
File finished execution
```

The design of the interpreter is chosen to be serial, rather than threaded. This may seem like an odd choice given the π-calculus is fundamentally parallel, however a serial implementation was chosen given the scope of the project, as a parallel implementation may be much harder to develop and debug.

4.3 Type System

An approach of *type checking* was taken, meaning that the user must correctly type their programs using type annotations, and the type system will verify if the program is well typed as a result of the type annotations. To determine if a process is well typed or not, the rules defined in Figure 2.6 were used. The system involves pattern matching the correct rule and applying it, given the process and type context Γ. Because the system type checks a process, for any type checking we construct a tree starting at the bottom, where the root is the process and type context given, with nodes of each rule, and leaves ending in $un(\Gamma)$.

The system should be smart enough to infer the types of values, (e.g. in sending the value `3`, the type system should be able to tell that `3::int`) so that the user does not have to annotate it themselves. type checking occurs at runtime, at each time the interpreter enters a process. If a process is well-typed it then executes, but if it does not type check and is therefore not well-typed it is stopped from executing. If a process does not type-check, we then continue running the rest of the program.

Error messages should be provided in the case of a non well-typed process, so that the type system can be more useful to the user. An error would occur when one of the typing rules is not satisfied. An example error message might be:

```
Error in type check: [T-In] Expected {!int} but got {?int} in process
    {x(v).x<u+v>.zero}.
```

This message shows the user the rule that failed (T-In), the expected and actual types, as well as the process the type check failed on.

5 Implementation

5.1 Language

The language was implemented using ANTLR. As mentioned in Section 4.1, the language is designed to be as simple and close to the π-calculus as possible. A snippet of the grammar can be seen in Listing 5.1. We can see that this gives the syntax for the base π-calculus, the extensions given by session types, and the extensions added in this work. The full grammar for processes is shown in listing A.1. The implementation of the language itself is simply an ANTLR grammar file, as ANTLR builds the lexer and parser itself. The syntax chosen for the language is shown in the table below:

π-calculus syntax	Chosen syntax
$\bar{x}\langle y\rangle.P$	`x<v>.P`
$x(y).P$	`x(y).P`
$(\nu x)\ P$	`(new x)P`
$!P$	`!P`
0	`zero`
$(\nu xy)\ P$	`(new x y)P`
$x \triangleright \{l_i : P_i\}_{i \in I}$	`x -> {l:P, ...}`
$x \triangleleft l_j.P$	`x <- l.P`

Subrules (such as `send` in Listing 5.1) were used to split up the process grammar, as well as to allow for matching classes in the interpreter. An instance where this was used was storing sends/receives in queues to be used later in the execution of a program. The syntax was chosen such that it would be easy to type for a user. Particular parts of the grammar would be very hard to write normally, for example the character ν (Greek 'nu') is not present on a standard UK keyboard. Communication is the most important part of π-calculus, so the simplest syntax for send and receive was chosen, while still being reminiscent of the standard π-calculus syntax.

5.2 Interpreter

The interpreter was implemented in Java. This was because Java is a target language of ANTLR, so the implementation can use ANTLR to parse and provide some framework for interpreting. The interpreter and language itself were implemented incrementally, starting with the very basics of communication, with the aim of implementing the base π-calculus as well as the extensions discussed in Section 5.1 as well as support for the extensions added with session types.

To create the serial implementation, some workarounds were required for interpreting the parallel processes. For example, say we had a queue of processes sending (`LinkedList sendingChannels`) and a queue of processes receiving (`LinkedList receivingChannels`, and if there's a matching send/receive in the queue then do the communication, otherwise add it to the queue. In this implementation, the process `a<3>.a(y).zero` would communicate with

```
proc: '[' p=proc ']' (SEQ proc)?        # parens
    | ZERO (SEQ proc)?                  # zeroProcess
    | send    SEQ proc                  # sendProcess
    | receive SEQ proc                  # receiveProcess
    | scopeRestrict proc                # scopeRestriction
    | REPL proc                         # replicate
    | proc PAR proc                     # parallel
    | proc '+' proc                     # choice
    | 'do' NAME (SEQ proc)?             # doProc
    | 'do' NAME '(' args ')' (SEQ proc)? # doProcArgs
    | newSession proc                   # sessionProcess
    | chName=NAME SELECT
      procName=NAME SEQ proc            # selectProcess
    | NAME BRANCH
      '{' (branch',')* branch '}'       # branchProcess
    ;
```

Figure 5.1: *Grammar of processes written in the ANTLR format.*

itself, which is semantically incorrect. To get around this issue two memory objects were used, a 'temp' and 'main' memory. Communications are added to the temp memory queue, and we check for a matching communication only in the main memory queue.

Scope restriction is another non-trivial feature to implement. In the interpreter, scope restricted names would get mapped to a name that does not exist. Uniqueness is guaranteed, for each name the interpreter appends the character "^" until the name is unique, for example `a` $\mapsto$ `a^`. The character "^" was chosen here, as it is not in the grammar, and so the user cannot type it themselves. A list of scope restricted names are kept, so that we know when we have a unique name and so that we use the correct one in the case of nested scope restrictions.

In implementing choice, some steps had to be taken to ensure the correct behavior. An example parse tree is shown in Figure 5.2. The idea behind the current implementation is on entering a choice, construct a list of all of the processes in the choice, adding them recursively. Then, on exiting the choice, execute one of them non-deterministically. A problem with this implementation is that after entering the choice, the rest of the process will run normally. This is because the tree will continue to be walked, so the `sendProcess` will be visited. To stop this from happening, on entering the first choice we add all of the choices by using another method (instead of walking the tree and adding on each choice), and once all of the choices are added, delete all of the children of the first choice. By deleting the children, we ensure that those nodes are not visited, and all of the information is kept by adding the choices to a list. Of course, an ideal scenario would involve not mutating the parse tree, however ANTLR does not provide functionality for not visiting certain parts of a tree.

Recursion was by far the most difficult feature of the language to implement. It was especially difficult in the context of π-calculus, as the language allows infinitely recursing processes to run in parallel, e.g. the following program will communicate forever:

```
let P = a<"hello">.do P
let Q = a(y).do Q
do P | do Q
```

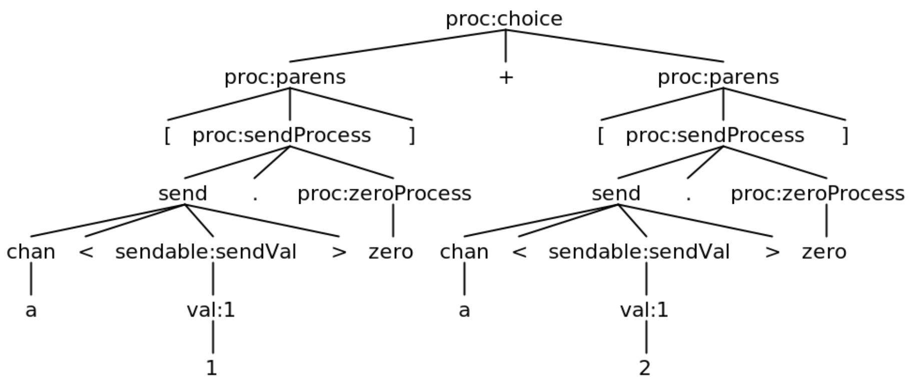


Figure 5.2: *The parse tree from input text "[a<1>.zero]+[a<2>.zero]".*

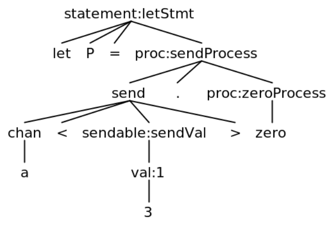


Figure 5.3: *The parse tree generated from input text "let P = a<3>.zero"*

Recursion (in `do` processes only, not replicated processes `!P`) was implemented by choosing selectively when to execute the recursive call. A process `P` is recursive if it contains the sub-process `do P`. Choosing when to run the call starts by keeping a list of running processes. On entering a `do` process, we first check if the name of the process is an element in the list, and if it is an element we do not run it but instead add it to a list of recursive calls. If the name is not already running, we add it to the list of running names, and continue walking the tree. When we exit a `do` process, we then remove the name from the list of names. On exiting a line, we then run all of the recursive calls in the list once. This permits infinite recursion, as when we exit the line, running the recursive calls in the list will allow us to add the recursive call again.

A frequently used pattern in the interpreter is to avoid running some part of the parse tree, and instead execute it later (usually if some predicate is true). An example of this is `let/do` statements, where we do not want to run the process subtree of the `let` statement, shown in Figure 5.3, but instead run it when we enter the `do`. If we did not delete the `sendProcess` subtree, the walker would visit those parts of the tree, and in this case send on channel a. To implement this, I rely on Java's behaviour of adding to data structures like `List` and `Map`. On adding a `context` to some data structure, it provides a deep enough copy that we can delete all children (to not visit them), but also preserve all of the information of the node. To visit a saved context `ctx`, ANTLR provides methods to enter and exit the rule, and we can also visit children of `ctx` with the code snippet `ctx.children.forEach(t -> walker.walk(this, t))`.

To help ensure correctness, testing was used heavily during development. Unit testing (testing single methods) is very difficult in ANTLR projects. The biggest issue is that in using a listener, each enter and exit rule has a return type of `void`, so unit testing would not be as simple as checking return values from entering a rule. Another problem with unit testing is that any non-trivial program will require multiple rules to be visited, which detracts from the point of unit testing, that is to only test the smallest units of the software. Instead, integration tests were used. The approach taken was to write small programs, and measure the communications that occur. This was a sensible approach, as in the π-calculus communications tell us everything about the process. Each test program was designed to test specific expected behaviour, for example if a replicated process communicated enough times, or recursion was functional.

To illustrate the correctness of the interpreter, consider the "car" system as outlined in Figure 2.3. We can write this system in the language:

```
let trans(talk, switch, gain, lose) = [talk(x).do trans(talk, switch, gain,
    lose).zero]+[lose(t).lose(s).switch<t>.switch<s>.do idtrans(gain, lose).zero]

let idtrans(gain, lose) = gain(t).gain(s).do trans(t, s, gain, lose).zero

let control1 = lose1<talk2>.lose1<switch2>.gain2<talk2>.gain2<switch2>.do
    control2.zero
let control2 = lose2<talk1>.lose2<switch1>.gain1<talk1>.gain1<switch1>.do
    control1.zero

let car(talk, switch) = [talk<"hello world!">.do car(talk, switch)] +
    [switch(t).switch(s).do car(t, s)]

let trans1 = do trans(talk1, switch1, gain1, lose1)
let trans2 = do trans(talk2, switch2, gain2, lose2)

let idtrans1 = do idtrans(gain1, lose1)
let idtrans2 = do idtrans(gain2, lose2)

# --- #

let system = (new talk1)(new switch1)(new gain1)(new lose1)(new talk2)(new
    switch2)(new gain2)(new lose2)
             [do car(talk1, switch1) | do trans1 | do idtrans2 | do control1]

do system
```

It should be noted that this system consists on recursive processes in parallel, and should infinitely recurse with infinite output. The output will be different every time the file is ran, due to the non-deterministic choice in the definition for An substring of an example output might be:

```
--> received on channel talk1: val "hello world!"
--> received on channel gain2: channel talk2
--> received on channel gain2: channel switch2
--> received on channel lose1: channel talk2
--> received on channel lose1: channel switch2
```

The interpreter is packaged as a jar file, using the maven build system. Also included is the rest of the application, meaning the parser and type system, which together form the interpreter. The usage is simple, we can interpret a file from the command line with `java -jar pi.jar [file]`. Because the interpreter is written in Java, it is portable and runs on any machine with Java.

5.3 Type System

The type system was implemented in Scala. Scala was chosen as it has some very useful features, most notably its pattern matching system, as well as its ability to be integrated with Java projects. The language allows us to pattern match anything, with a system similar to Haskell's. In the context of the type system, the pattern matching of Scala was used to match the typing rules from Figure 2.6, as each ANTLR rule corresponded to exactly one typing rule.

To model a typing context Γ, Scala's `immutable.Map` was chosen. The type definition `type TypingContext = immutable.Map[String, TypeContext]` was used, mapping from a standard `String` to an ANTLR `TypeContext`. An immutable Map was chosen over a mutable one so that the recursive calls would work properly, avoiding mutating the Map in one rule, making another rule not type check when it should.

The type checker itself is simply a recursive function, `def typeCheck(p: ProcContext, gamma: TypingContext): Boolean`, which is called from the interpreter on entering a process. The function pattern matches `p`, and applies the relevant rule with the `TypingContext`. An example case is to match the process `(new x y :: typeName) P`:

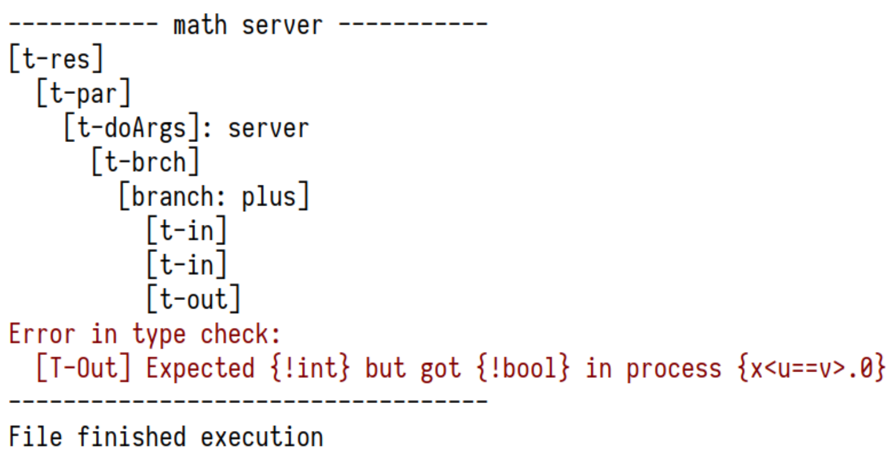
```
// T-Res
case p: SessionProcessContext => {
  val xName: String = p.newSession().x.getText
  val yName: String = p.newSession().y.getText
  val T = gamma(p.newSession().typeName.getText)
  val gamma1 = gamma + (xName -> T) + (yName -> dual(T)) -
    p.newSession().typeName.getText

  val result = typeCheck(p.proc(), gamma1)
  result
}
```

We can see that this process rule (an ANTLR SessionProcessContext) corresponds to the typing rule [T-Res], specifically the lower half of the rule. We extend gamma to have the types of `x` and `y`, and remove the key/value pair of the type name to avoid the type context being linear at the end of the type check. We then type check `P` in a recursive call with the extended gamma, going from the lower half of the rule to the top half. The structure of the tree is identical to the structure shown in the output of the type system.

Some of the rules have the form $\Gamma_1 \circ \Gamma_2 \vdash P$, where $\circ$ denotes the disjoint union of Γ_1 and Γ_2. This is known as a context split. In following the typing rules downwards, we simply start with the two typing contexts and can union them to continue the proof. However, when constructing a tree the other way, the inverse is not as simple. In other words, when type checking we have to take a union of two or more contexts, and split it into two disjoint contexts, which each type some process determined by the rule. In the implementation, unrestricted types (only the base types, in the specific implementation of session types used here) can be safely added to all of the typing contexts, as in the leaf rules we check $un(\Gamma)$. This was implemented with the function `splitContext2`. This function takes a list of channel names `channels`, and puts any keys from the `TypeContext` which are an element in `channels` into one `TypingContext`, and any others into a second `TypingContext`. If any element is unrestricted, add it to both.

Error messages were implemented with exceptions. If a rule was not satisfied due to one of the predicates of that rule, an exception is thrown at that point. Using exceptions provides a cleaner implementation, as we can have the recursive type checking (each check returns a boolean),



```
----------- math server -----------
[t-res]
  [t-par]
    [t-doArgs]: server
      [t-brch]
        [branch: plus]
          [t-in]
          [t-in]
          [t-out]
Error in type check:
  [T-Out] Expected {!int} but got {!bool} in process {x<u==v>.0}
-----------------------------------
File finished execution
```

Figure 5.4: *Output from the type system, with an error message where the type specified a channel sending an int, but the channel instead sent a bool.*

whilst allowing for helpful error messages. The exceptions are caught in the interpreter, where the error messages are printed. An example error message can be seen in Figure 5.4.

Type inference (in sending) was implemented by matching text in the send. The exact regular expression patterns from the parser were used to ensure correctness.

To help with learning session types, the user can choose to print labels of what is being type checked. An example of this is shown in Figure 5.4. The labels provide the structure of the type checking, and can also help with debugging if the user does not find the error messages helpful.

To demonstrate the type system, consider the math server example defined in Section 2.2. The session type and process definitions, as well as the server-client process, are written in the language as so:

```
@typecheck:labels
server :: &{ plus :: ?int.?int.!int.end
           , sub  :: ?int.?int.!int.end
           , eq   :: ?int.?int.!bool.end }

let server(x) = x -> { plus: x(u).x(v).x<u+v>.zero
                     , sub:  x(u).x(v).x<u-v>.zero
                     , eq:   x(u).x(v).x<u==v>.zero }

let client(y) = y <- plus.y<1>.y<2>.y(result::int).zero

(new x y :: server)[do server(x) | do client(y)]
```

Which when interpreted, results in the following output:

```
[t-res]
    [t-par]
        [t-doArgs]: server
            [t-brch]
                [branch: plus]
                    [t-in]
                    [t-in]
                    [t-out]
                    [t-inact]
                [branch: sub]
                    [t-in]
                    [t-in]
                    [t-out]
                    [t-inact]
                [branch: eq]
                    [t-in]
                    [t-in]
                    [t-out]
                    [t-inact]
        [t-doArgs]: client
            [t-sel]
            [t-out]
            [t-out]
            [t-in]
            [t-inact]
Process is well typed; continuing...
--> received on channel y: val 13
--> received on channel y: val 5
--> received on channel y: val 18
```

This shows the typing rules used, and notifies the user that the process is well typed. The interpreter then continues to execute the process, showing the communications.

6 Evaluation

6.1 User study

To help evaluate the work, a user study was run. The study was designed to evaluate the following aspects of the work:

- Correctness of the interpreter
- Usability of the language and interpreter
- Effectiveness as a learning tool

These were chosen as the focus of the study to evaluate the requirements as outlined in Chapter 3. The study was structured as follows:

- Give the user an introduction to the π-calculus
- Show the user through several example files, and get the user to interpret them
- Allow the user to edit the example files, or write their own to try the interpreter

During the final section, I would talk with the user about their thoughts as they write code, similar to a 'think-aloud' methodology (JÃÿrgensen 1990). This was to try and determine if the syntax was hard to understand, or if the output was confusing, as well as bringing up anything else that may confuse the user.

The participants were categorised into two groups; those with experience of π-calculus, and those without. Each of the participants had a background in computer science. These groups were equal in size, with four students in each group. As some of the users targeted in the study had either no experience, or very little experience, the introduction aimed to show the most important constructs of the π-calculus. The example files were structured in such a way that it would progressively introduce concepts in the language to the user. The first file showed a simple communication: `(new a)[a<"hello, world!">.zero | a(x).zero]`. The concepts introduced in order were: sequential communication, mobility, session communication with types, branch/select, and recursion.

Allowing the user to write their own programs in the language was included with the aim to find bugs in the interpreter, as especially those without prior experience with π-calculus will likely write erroneous input, thus highlighting bugs in the interpreter.

Significant results of the user study will be shown in the following sections.

6.2 Correctness

Correctness was continually checked during implementation. As mentioned in Section 5.2, this was achieved by integration testing. However, some bugs were highlighted during the user study. As an example, sending the value `0` caused a parse error, due to the ANTLR grammar not being precise enough; the regular expression for the zero process and the number zero coincided. This was a simple fix, by changing the original regular expression for the zero process from `ZERO`

`: '0' | 'zero'` to simply `ZERO : 'zero'`. Although this bug was identified and easily fixed, it shows that there are likely to be more bugs in the implementation, which may be more subtle. It also shows that the integration test suite was not comprehensive enough, although it was expected that the test suite would not completely verify the implementation. As mentioned in Section 3.2, a theorem prover such as Coq could have been used to verify the implementation, however this would have been a significant task, and outside of the scope allowed by the project.

An example of a more subtle bug that was highlighted in the evaluation is in the implementation of the choice process. The semantic defining the choice operator as written by Milner (1999) is given by $\bar{a}\langle x \rangle.P + R \mid a(y).Q + S \rightarrow \bar{a}\langle x \rangle.P \mid a(y).Q$. In the implementation however, the choice is completely non-deterministic, that is to say that there is no guarantee that two chosen processes in parallel can communicate.

Although the test suite was not comprehensive, it did cover a significant portion of the π-calculus, verifying the absolute basics of the language, as well as some 'edge cases', for example two restricted names in parallel should not communicate. A more formal approach to the project might have avoided these implementation bugs, a formal workflow would have been:

- Define the grammar (syntax)
- Define the semantics
- Implement these semantics exactly

However, this workflow would have been difficult given the use of ANTLR to build the interpreter. A possible way to implement the interpreter in this way would have been similar to the type system implementation, by using structural pattern matching to match the reduction rule and implementing the rule in code.

6.3 Usability

One of the aims of the work is that the language must be close to Milner's π-calculus, but with added constructs for usability. Feedback from the user study showed that the implemented language was very 'pure', however could benefit from more constructs outside of those specified in Section 4.1. Another suggestion to help make the language more usable was adding some syntactic sugar, particularly if/else statements and loops. I personally think this would be a useful addition, which would still be pure enough in the context of the π-calculus, and as the target userbase would be computing science students these constructs would not require additional learning. A different balance might need to be struck between simplicity and additional constructs.

The language does not support many mathematical operations. This was mostly because I was focused on implementing the core functionality to get the two significant examples (the math server and car system) working. Implementing further operations would not require a significant amount of work, and would add significant functionality to the language.

An issue that many of the users brought up was that the interpreter's error messages were lacking. The implementation did not focus on error messages, and if the user gave an input with a syntax error, the interpreter would simply exit. This was a problem for new users, as when they were learning the language they may make a subtle error, for example missing the zero process at the end of a process.

Feedback from the user study showed that output from the type system was very clear. In particular, the labels were useful in debugging, as they could be used alongside the error message to debug a type error.

6.4 Effectiveness as a Learning Tool

Overall, data from the user study showed that for users with no experience of π-calculus, the language and interpreter helped their understanding of the π-calculus. Users liked the error messages and output from the type system, and overall the response from the questionnaire implies that these features helped understanding of the π-calculus and the session type extension. Further user studies would be required to gain a deeper insight into the effectiveness of the tool for learning. If time had permitted, students currently learning the π-calculus would have been targeted for the user study.

6.5 Related Work

This section will discuss some related work over π-calculus interpreters and projects using session types.

6.5.1 pi-calculus

Pi-calculus (renzyq19 2014) is an interpreter for a small language based on the *applied* π-calculus. The aim of the project is to provide an implementation of the applied π-calculus such that web protocols can be executed with existing implementations. The language uses haskell-like syntax. An example program in pi-calculus can be seen in the following example, which communicates a single message:

```
let client =
    out(stdout,"Enter Host"); in(stdin,host);
    out(stdout,"Enter Port"); in(stdin,port);
        let ch = chan(host,port) in
            in(stdin,msg);out(ch,msg);in(ch,msg)
let server =
    out(stdout,"Enter Port");
    in(stdin,port);
        let ch = {port} in
            in(ch,msg);in(stdin,reply);out(ch,reply)
```

As we can see, the syntax is abstracted the standard π-calculus, and requires a lot of boilerplate for a simple communication. For receiving, instead of $x(y)$ the language uses `in(x, y)`, which is not in the standard π-calculus grammar.

Despite the more complicated syntax, pi-calculus has great potential for application, and can be used to model complex processes such as the handshake protocol used in cryptography.

6.5.2 JsonPi

JsonPi (Braun 2018) is a π-calculus interpreter. It uses syntax very close to the definition given by Milner (1999), for example the simple communication is written:

```
world<hello>; | world(message);
```

where `;` represents the zero process in π-calculus. The language allows any valid JSON for names, so that we can send arbitrary JSON in communications. It even allows for branching through structural pattern matching, for example in the following snippet, which behaves similarly to Java's `switch` statement:

```
choose
    when x(a) then ;
    when y(b) then z<b>;
    when [c=d]x<e> then y(f);
    default z(g);
end
```

Other features include replication (`!P`), conditional execution (`[a=b]P` - if `a=b` then do `P`), let binding (`let P = x<a>;`), and modules to allow structure in large projects.

6.5.3 SePi

SePi is a language utilising session types. Session types are written in the language with identical syntax to Dardha et al. (2017), for example to describe a channel that receives an integer then stops, we write `?integer.end`. The syntax for sending and receiving in processes is `a!x` and `a?y` respectively. The language supports the branch and select processes from session types, an example showing a branch and select in parallel is shown in the following snippet:

```
new w r: +{setDate: end, commit: end}
w select setdate |
case r of setDate -> printString!"Got setDate"
          commit  -> printString!"Got commit"
```

which selects the branch `setDate` and executes it. The syntax is close to, but not exactly, the syntax in the definition of session typed π-calculus.

SePi also supports *recursive* types, which could be used for example in a server that offers many branches, and allows any number of clients to communicate with it. This is a useful addition to session types, as without it a server could only communicate once in a single session.

6.5.4 GV

'GV' is a multithreaded functional language with session types (Gay and Vasconcelos 2010). The syntax of session types is that of the syntax defined by Dardha et al. (2017). The syntax of the language itself is similar to Haskell. An example of the language is shown with an online shop. The type of the shop server is given by the session type:

$$\text{Shop} = \&\langle add : ?Book.Shop, checkout : ?Card.?Address.end\rangle$$

The type of a shopper client is the dual of this, given by:

$$\text{Shopper} = \oplus\langle add : !Book.Shopper, checkout : !Card.!Address.end\rangle$$

The language uses the notation $\langle Shop \rangle^a$ to show that *Shop* is capable of *accepting* connections, and $\langle Shopper \rangle^r$ to show that the shopper can *request* a connection.

The shop is implemented by a recursive function:

```
shop :: ⟨Shop⟩ᵃ → end
shop shopAccess = shopLoop (accept shopAccess) emptyOrder

shopLoop :: Shop → Order → end
shopLoop s order =
    case s of {
        add  ⟹ λs.let (book, s) = receive s in
                   shopLoop s (addBook book order)
        checkout  ⟹ λs.let (card, s) = receive s in
                        let (address, s) = receive s in s
    }
```

Although this language supports session types, the syntax of the language is far abstracted from the π-calculus. The syntax would also likely be confusing for those without experience in functional languages.

7 | Conclusion

7.1 Summary

The aim of this work was to design and implement a programming language based on the π-calculus with session types, which could be used in teaching. This was split into three main objectives, which were designing and implementing the programming language, implementing an interpreter for the language, and implementing a type system for the language based on the specification of session types. 'Pi-calc' is the language designed and implemented in the work, which is bundled as a Java application containing the parser, interpreter, and type system.

The evaluation has shown that the language is an effective interpreter of the π-calculus with session types, and has promise for being used in education. More language constructs could help to make the language more usable, but a balance between purity and syntactic sugar should be found.

7.2 Future work

The most significant suggestion for future work would be to rework the interpreter to directly apply reductions, rather than traversing ANTLR's generated parse tree. As mentioned in Section 6.2, I believe that implementing the exact semantics defined in Figure 2.2 would avoid any bugs in interpreting processes. This approach would also make implementing the requirement of showing reductions described in Section 3.1 much easier, as the interpreter could apply each reduction stepwise.

Another important suggestion is error messages in the parser. In the event of a parse error, the interpreter will not give any information about what the parse error was, which is unhelpful for any users but especially for those learning π-calculus.

A different approach in the interpreter may be to have a threaded implementation rather than a sequential, that is to have each process in a parallel composition running as its own thread. This would make sense given the parallel nature of the π-calculus, and could make implementation of communication easier, if a reduction-driven approach was not taken. This is how Golang implements *goroutines* (Ajmani 2014), which are similar to π-calculus processes, although sequential implementations also exist.

The language would benefit from more constructs to aid usability, as the language is possibly too pure. This might include if statements in the form $[x = y]\ P$, where P is executed if and only if $x = y$. As well as control flow constructs, a useful addition would be more mathematical operations, which would extend functionality and allow for more significant programs to be executable by the interpreter.

Given more time, formally verifying the language using a theorem prover may be useful work. However, this would be a significant undertaking, and may be excessive given the intended audience of students. If this work was intended to be used in a more formal setting, such as verifying π-calculus models work as intended, this would be an important addition.

A Appendices

A.1 Pi-calc Grammar

```
proc
    : '[' p=proc ']' (SEQ proc)?         # parens
    | ZERO (SEQ proc)?                   # zeroProcess
    | send    SEQ proc                   # sendProcess
    | receive SEQ proc                   # receiveProcess
    | scopeRestrict proc                 # scopeRestriction
    | REPL proc                          # replicate
    | proc PAR proc                      # parallel
    | proc '+' proc                      # choice
    | 'do' NAME (SEQ proc)?              # doProc
    | 'do' NAME '(' args ')' (SEQ proc)? # doProcArgs
/* --------------------Session types-------------------- */
    | newSession proc                    # sessionProcess
    | chName=NAME SELECT
      procName=NAME SEQ proc             # selectProcess
    | NAME BRANCH
      '{' (branch',')* branch '}'        # branchProcess
/* ----------------------------------------------------- */
    ;

type
  : sessionType                          # sessionTypeT
  | baseType                             # baseTypeT
  ;

sessionType
    : 'end'                                    # stEnd
    | '?' baseType SEQ sessionType             # stRec
    | '!' baseType SEQ sessionType             # stSend
    | '&' '{' (branchType ',')* branchType '}' # stBranch
    | '+' '{' (branchType ',')* branchType '}' # stSelect
    ;

baseType
    : 'int'                              # intT
    | 'str'                              # strT
    | 'bool'                             # boolT
    ;

branchType
    : label=NAME '::' type
    ;
```

```
args
    : (NAME',')* NAME
    ;

branch
    : label=NAME ':' proc
    ;

send
    : name=chan '<' sendable '>'
    ;

sendable
    : val                                  # sendVal
    | 'var' var                            # sendVar
    | chan                                 # sendChan
    | expression                           # sendExpression
    ;

expression
    : NAME '+' NAME                        # addExpression
    | NAME '-' NAME                        # subExpression
    | NAME '==' NAME                       # eqExpression
    ;

receive
    : name=chan '(' var ('::' type)? ')'
    ;

scopeRestrict
    : '(' NEW name=chan ')'
    ;

newSession
    : '(' NEW x=chan y=chan '::' name=NAME ')'
    ;

val
    : INT
    | STRING
    | bool
    ;

bool
    : TRUE
    | FALSE
    ;

chan
    : NAME
    ;

var
    : NAME
    ;
```

Listing A.1: *The .g4 grammar of processes in pi-calc*

A.2 User Guide

```
## Written by Andrew M - 2193329m@student.gla.ac.uk

**Found a bug or a program that doesn’t run as expected? Please send me an email
    or talk to me!** Including whatever .pi was run would be useful for me : )

## Download latest jar:

**You should probably run this on a lab machine - this is built using a public
    gitlab runner and I’m not sure how secure they are.** Alternatively, you can
    compile everything yourself by running ‘mvn package‘ in the root (mvn ==
    maven), the jar is located in ‘target/pi.jar‘.

## Running the interpreter:

- ‘java -jar pi.jar [file.pi]‘

## Writing ‘.pi‘ files:

The following grammar gives the main functionality:

‘‘‘
file = line*

line = process | statement

process = [process]                # brackets
        | zero                     # zero process, 0
        | chan<val>.process        # send val on chan
        | chan(variable).process   # receive on chan, bind result to variable
        | (new chan) process       # scope restriction
        | !process                 # process replication
        | process | process        # parallel

statement = show var               # prints the value of variable var
‘‘‘

Comments are started with a single ‘#‘.

## Example programs

(Output is prepended with ‘>‘)

#### Basic send and recieve in parallel
‘‘‘
a<3>.zero|a(x).zero
show x

> --> received on channel a: 3
> x=3
‘‘‘

#### Multiple send/receives
‘‘‘
b<5>.b<3>.b<4>.zero|b(x).b(y).b(x).zero
```

~~~
show x
show y

> --> received on channel b: 5
> --> received on channel b: 3
> --> received on channel b: 4
> x=4
> y=3
```

#### Sending a channel
```
a(x).x<5>.zero|a<b>.b(y).zero  # send the chan b on a
show y

> --> received on channel a: b
> --> received on channel b: 5
> y=5
```

#### Replication
```
a(x).a(y).zero|!a<"hello">.zero
show x
show y

> --> received on channel a: "hello"
> --> received on channel a: "hello"
> x="hello"
> y="hello"
```
~~~

A.3 Ethics Approval

A.4 User Study Questionnaire

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 Session Types in pi-calculus

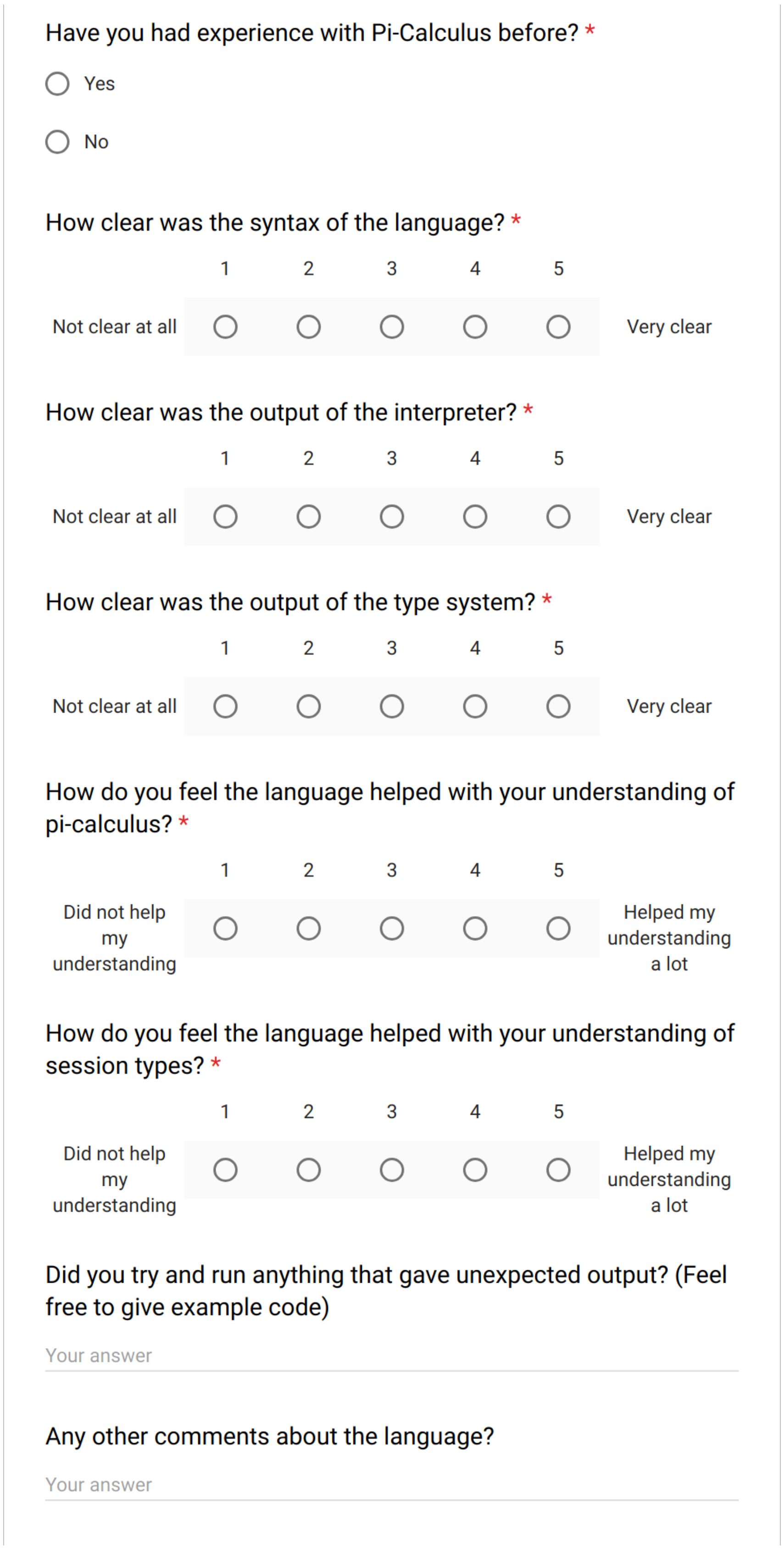
Student's Name Andrew McNab

Student's Registration Number 2193329

Student's Signature [illegible]

Supervisor's Signature [illegible]

Date 25/3/2019



Have you had experience with Pi-Calculus before? *

○ Yes

○ No

How clear was the syntax of the language? *

	1	2	3	4	5	
Not clear at all	○	○	○	○	○	Very clear

How clear was the output of the interpreter? *

	1	2	3	4	5	
Not clear at all	○	○	○	○	○	Very clear

How clear was the output of the type system? *

	1	2	3	4	5	
Not clear at all	○	○	○	○	○	Very clear

How do you feel the language helped with your understanding of pi-calculus? *

	1	2	3	4	5	
Did not help my understanding	○	○	○	○	○	Helped my understanding a lot

How do you feel the language helped with your understanding of session types? *

	1	2	3	4	5	
Did not help my understanding	○	○	○	○	○	Helped my understanding a lot

Did you try and run anything that gave unexpected output? (Feel free to give example code)

Your answer

Any other comments about the language?

Your answer

7 Bibliography

S. Ajmani. Go concurrency patterns: Pipelines and cancellation. https://blog.golang.org/pipelines, 2014. Go statements, Accessed 22/3/2019.

G. Braun. Jsonpi. https://github.com/glenbraun/JsonPi, 2018. Accessed 16/3/2019.

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.

S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2): 191–225, Nov 2005. ISSN 1432-0525. doi: 10.1007/s00236-005-0177-z. URL https://doi.org/10.1007/s00236-005-0177-z.

S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19âĂŞ50, 2010. doi: 10.1017/S0956796809990268.

A. H. JÃÿrgensen. Thinking-aloud in user interface design: A method promoting cognitive ergonomics. *Ergonomics*, 33, 04 1990. doi: 10.1080/00140139008927157.

R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982. ISBN 0387102353.

R. Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, 1999.

renzyq19. pi-calculus. https://github.com/renzyq19/pi-calculus, 2014. Accessed 16/3/2019.

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.

7 List of Figures

2.1 Syntax of π-calculus processes with session types (Dardha et al. 2017, Fig.1). 2
2.2 Semantics of the π-calculus with session types. 3
2.3 The “car” system 4
2.4 Syntax of session types (Dardha et al. 2017). 5
2.5 The duality relation on session types (Dardha et al. 2017). 5
2.6 Typing rules for the π-calculus with session types (Dardha et al. 2017). 6
2.7 An ANTLR parse tree generated from the π-calculus process "a<1>.zero" 8

5.1 Grammar of processes written in the ANTLR format. 14
5.2 The parse tree from input text "[a<1>.zero]+[a<2>.zero]". 15
5.3 The parse tree generated from input text "let P = a<3>.zero" 15
5.4 Output from the type system, with an error message where the type specified a channel sending an int, but the channel instead sent a bool. 18