Bruce Collie

An Implementation of the π-Calculus

Dissertation for Part II of the Computer Science Tripos
University of Cambridge, 2015–16
Trinity Hall

Proforma

Name:	Bruce Collie
College:	Trinity Hall
Project Title:	An Implementation of the π-Calculus
Examination:	Computer Science Tripos, Part II (2015–16)
Word Count:	11797
Project Originator:	Prof. Alan Mycroft
Supervisor:	Prof. Alan Mycroft

Original Aims of the Project

To produce a practical implementation of the process algebra known as the π-calculus, first presented by Milner, Parrow & Walker in their paper "*A Calculus of Mobile Processes, Part I*". It should provide a specification for a language based on the abstract syntax and semantics of the π-calculus, along with some modifications. A compiler should be produced that can transform this language into an executable format, for which an interpreter should be developed. A suite of programs that demonstrate the language and its correspondence to the abstract semantics of the π-calculus should be written.

Work Completed

The implementation satisfies the success criteria set out in the project proposal. A language (PCL) was designed along with an executable bytecode format. A compiler was implemented that can compile PCL to this bytecode format, and a virtual machine implemented to execute the compiled bytecode. A set of PCL programs were written to demonstrate the satisfaction of the success criteria.

Special Difficulties

None.

Statement of Originality

I, Bruce Collie of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background

In computer science, computation of a particular kind is often modelled by a *calculus*—a mathematical definition of computation semantics. The most well known of these is the untyped λ-calculus, first described by Alonzo Church [1] in the 1930s. The basic model Church provided is now frequently used to reason about and describe functional programming languages such as Haskell or Standard ML.

A current trend in industry is the need for increased parallelism in programs due to limitations in the single-core performance of microprocessors. As a result, research into parallel computation is a popular topic in theoretical computer science today. Reasoning about the behaviour of such parallel programs is difficult in practice (when compared to reasoning about sequential programs). As well as this, many different models of parallel computation exist—for example, the run-forever concurrent processes of Erlang [2], or the SIMD[1] processing used in GPU-based computation [3].

Perhaps the first attempt to formalise the notion of *parallel composition* as a way of structuring programs was Tony Hoare's *Communicating Sequential Processes* (CSP) [4] in 1978. The initial specification lacked a mathematical semantics, and as such was closer in form to a programming language than to a process algebra.[2] In 1980, Robin Milner presented his *Calculus of Communicating Systems* (CCS) [5]—a conceptually similar language, but mathematically described as a process algebra.

[1]Single Instruction, Multiple Data—the same operation is applied to many independent data simultaneously.

[2]A semantic model that uses algebraic laws to formalise the interactions between concurrently executing processes.

The π-calculus was presented by Milner in a 1992 paper [6] as a process algebra based only on the communication of names—a model that is more abstract than CCS, while being fully expressive (Milner showed that the π-calculus is in fact Turing complete [7] by demonstrating an encoding of the λ-calculus).[3]

Since then, the π-calculus has since been studied in detail, with variants of it being used to model systems as diverse as cryptographic protocols [9] and biological systems [10]. As well as this, several programming languages and libraries have been developed that integrate ideas from the π-calculus.

The semantics of the π-calculus are based on a collection of concurrently executing *processes* that communicate by a *name passing* mechanism. Processes synchronise with each other on a complementary action pair (input $x(a)$ and output $\overline{x}\langle b\rangle$—one process sends the name b, the other receives it, and both continue execution). Besides input, output and parallel composition, the only other fundamental operation in the π-calculus is the infinite replication of a process.

1.2 Previous and Related Work

There exist various previous implementations of concurrent languages based on the concurrency models described by CSP, CCS and the π-calculus. As well as full languages, multiple virtual machine architectures for executing compiled concurrent bytecode exist (of which the most well known is BEAM, the Erlang virtual machine [11]).

One of the earliest such languages to be developed was occam, which drew on CSP as a basis [12]. The language was originally intended to be the native programming solution for the Transputer[4] series of microprocessors developed by the British company INMOS in the 1980s. Following the end of development by INMOS, Fred Barnes and Peter Welch at the University of Kent produced a new version of the language named occam-π that incorporates features from the π-calculus [13].

A similar language is Pict, designed to incorporate the semantics of the π-calculus [14], while adding features to improve its usefulness as a general-purpose language. In this way Pict is similar to the language I implement in this project, although Pict has a far greater scope of features than would be feasible for a Part II project (e.g. a static type system with partial infer-

[3]Banach and van Breugel show that CCS is also Turing-complete, by showing an encoding of the π-calculus in CCS [8].

[4]An architecture designed specifically for parallel computing by connecting many individual processors in a network.

ence, recursive types, record types and pattern matching). Additionally, the implementation of Pict described in [14] compiles to C, while in this project I compile to a bytecode format that is executed by a virtual machine.

Chapter 2

Preparation

In this chapter I give an analysis of the project requirements and how they will be met from a software engineering perspective. Additionally, I give a summary of the syntax and semantics of the π-calculus. I give a design for PCL, a language based on the π-calculus, and a set of virtual machine instructions that can be used as a target for compilation of PCL.

2.1 Requirements Analysis

In this section I lay out the primary requirements that were identified for the project, and how they were analysed in order to formally specify the goals of the project.

2.1.1 Components

The primary components of the system as proposed are:

Language Design A language with well-defined syntax and semantics should be designed. The primary work of the project will then be concerned with the implementation of this language.

For syntax, a formal description of the language's format should be produced that details its grammar and lexical structure. With regard to semantics, it will be beyond the scope of the project to produce a formal operational semantics for the language that can be shown to be equivalent to the π-calculus in some sense. Instead an informal description of program behaviour, together with a demonstration of behavioural correspondence with the π-calculus will suffice.

Virtual Machine Design In order to simplify the compilation of the language, the project will not aim to compile to native code. Instead a custom instruction set will be designed. This instruction set should allow richer instruction behaviour—this in turn will simplify the task of code generation.

The design of the virtual machine should however be low-level enough that it could potentially (with some modification and abstraction) be treated as an abstract machine for process calculi in general—though performing this modification is beyond the scope of the project. Having a degree of coupling between the semantics of the π-calculus and the virtual machine design will therefore be acceptable.

Interpreter An interpreter should be built to execute a single sequential program using the instruction set described above. Concurrency should be implemented by a scheduler that coordinates the execution of multiple interpreter threads.

The interpreter should not be a direct implementation of the operational semantics of the language (i.e. it should not encode the π-calculus transition relation directly—rather it should execute a sequence of compiled instructions).

Lexical Analysis Based on the definition of the language syntax, a lexer should be built that can generate a tokenised representation of source text.

Parser From a list of tokens generated by the lexer, a parser should be implemented that can produce a suitable representation of the program's abstract syntax. An appropriate parsing algorithm will have to be decided on and implemented for this component.

Code Generation Given an abstract syntax tree, virtual machine instructions should be emitted that correspond to the behaviour of the program. Inefficient or redundant instruction sequences will be tolerated as optimisation is beyond the scope of the project.

Link to External Code A mechanism for programs written in the language to call external code should be devised. This mechanism will allow the implementation to call programs that perform computation that the language cannot easily.

2.1.2 System Design

It will obviously be advantageous to consider how the separate parts of the implementation will be composed to produce a full system. A modular approach will be useful as the components of the system present clear demarcations, with distinct program representations at each boundary between components. A high-level overview of the system's design is given in Figure 2.1.

Based on this proposed design, it makes sense to implement each phase separately with well-defined data structures acting as the interface between phases. Writing the project in this way means that once these data structures are defined, work can be carried out on different sections independently (potentially using hand-coded stub data to provide valid input to phases if necessary—for example, to allow for out-of-order development or unit testing).

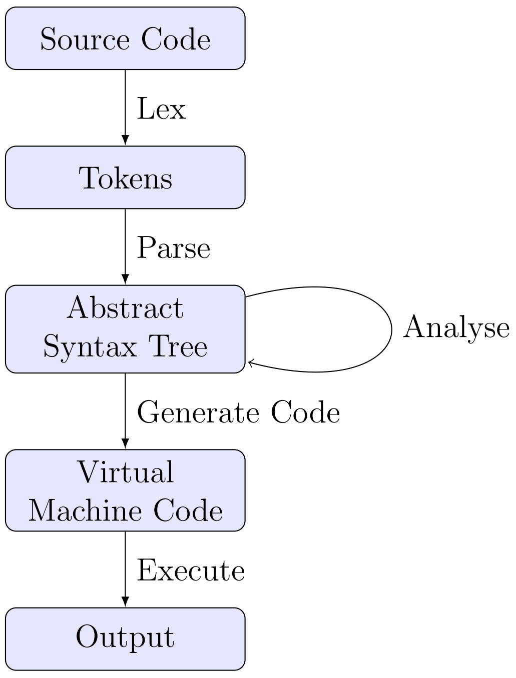

Figure 2.1: Data flow through the proposed system design

2.1.3 Required Algorithms and Techniques

In order to implement the project as described, some amount of background knowledge and several key algorithms will be required. Some of this background knowledge will come from Tripos courses, while some will require independent research.

Language Design In order to properly design a programming language, consideration must be given to both the syntax and semantics of the language.

In order to properly reason about and implement the semantics of the π-calculus, knowledge from the Tripos courses *Semantics of Programming Languages* and *Topics in Concurrency* will be necessary. Techniques used will include the analysis of operational semantics of a language, as well as the algebraic manipulation of process calculi.

Virtual Machine The implementation of the virtual machine will require knowledge of how such machines are implemented in the real world — this will partly require background research and investigation, as well as some knowledge from the Tripos course *Compiler Construction*.

The scheduler is likely to be the most complex component of the virtual machine. Implementing it will require strong working knowledge of common problems faced when implementing concurrent systems (e.g. deadlock, race conditions etc.), as well as general programming techniques in such environments. The Tripos course *Concurrent and Distributed Systems* is the most relevant to this section.

Compiler Frontend The front end of a compiler is generally taken to encompass a lexer and a parser, which together convert textual source code into an abstract syntax tree. The algorithms most often used for this are described primarily in the Tripos course *Compiler Construction*.

While lexing is a conceptually simple task, a suitable parsing algorithm will have to be selected based on the structure of the grammar given by the language design. Multiple candidates are available, but some algorithms place constraints on the grammar they parse, while others may be equivalently useful for the language but more complex to implement. It will therefore be necessary to choose an appropriate algorithm.

Code Generation The Tripos courses *Compiler Construction* and *Optimising Compilers* both cover this topic in some detail. The design of the virtual machine instruction set will inform this stage a great deal, though the general technique is likely to remain similar whatever instruction set is chosen.

As a broad technique, instructions are likely to be generated by a postorder traversal of the abstract syntax tree, possibly with alterations or corrections made after further analysis.

2.2 Language and Technology Evaluation

The language Scala [15] was chosen as the implementation language for the project for a number of reasons:

Type System The Scala type system is one of the language's strongest points. Full static type checking provides a non-trivial assurance of the correctness of code (in comparison to a dynamically typed language), while features such as traits, mixins and case classes allow for high level concepts to be expressed naturally in the type system.

Standard Library Scala is compiled to run on the JVM[1], and as such has access to the entire Java standard library. In addition to this, Scala has its own standard library that makes use of idiomatic language features. The availability of high-quality standard library code means that less time will be spent working on tasks not directly relevant to the project.

Multi-Paradigm Code written in Scala is not strictly constrained to a particular idiomatic style (compared to Haskell or Java, for example), and so different modules may be written differently as appropriate [16]. For example, some parts of the project might be better suited to a functional style, with others being more easily written with imperative code. Being able to write code in this way is perhaps not ideologically pure, but results in a flexible development process.

While there are other languages that match some of these benefits (e.g. Haskell has a similarly strong type system, and the Python standard library is perhaps as practically useful as Java's), no language I evaluated matched all of them while still being a stable language to develop a large project in.

Development tools were chosen based on their ease of use with Scala code—the standout contender on this front was IntelliJ IDEA 15, an IDE primarily designed for use with Java, but which has strong Scala support from an official plugin. Beyond the IDE, the only major technologies used were git (for source control, with BitBucket used as a remote repository host) and LaTeX (for typesetting documents).

[1] Java Virtual Machine

2.3 Starting Point

The project does not directly build on any other project, or on any domain-specific code. The only third party code used in the implementation of the project is that of the Java and Scala standard libraries, with all code specific to the project being written by myself.

There exist other implementations of languages and instruction sets based on the π-calculus (such as occam [12] and Pict [14]). While the project is similar to these implementations in some respects, the scope of their features is much greater than the time constraints for this project would allow. No code or implementation strategy has been directly taken from any similar projects.

2.4 Language Design

Before an implementation can be constructed, the language itself must be designed. The primary criteria for this design phase were:

- Encompass the semantics of the π-calculus—anything expressible in the π-calculus should be expressible in the language, but not necessarily vice-versa (as it may be desirable to add features that aid practical programming).
- Syntactically emulate the π-calculus to a reasonable extent, with concessions made for readability and ease of writing code in the language.
- Add a language feature that allows for external code to be called in some way, while still fitting into the semantics of the language and feeling subjectively "natural" in the context of a program.
- Integrate data types of some kind beyond the pure names of the π-calculus, while still preserving the core name-passing semantics.

2.4.1 The π-calculus

In order to create such a design, the π-calculus must be examined. The core structure is very minimal—there are only six syntactic constructions that can form a process. In this section I give a description of the semantics of the π-calculus.

Informal Description

The π-calculus is a process algebra—a set of algebraic laws that describe the interactions between concurrent processes. These laws specify a *transition relation* (written $\rightarrow$) that specifies when and how a process can reduce into another. For some processes this relation is nondeterministic. A full definition of the relation is given in Appendix D.

Data in the π-calculus is in the form of *names*, which represent both messages themselves and the links across which messages are sent.

The only way in which concurrent processes can interact is through *synchronisation*—if two processes perform complementary actions, they can transition together to a new state. Information is exchanged here by a name being passed from one process to the other.

π-calculus Grammar

The grammar of the π-calculus is given formally in Figure 2.2, with P defining the abstract syntax of a π-calculus process, and x, y being names from some set X.

$$\boxed{\begin{array}{rcl} P & ::= & x(y).P \\ & | & \overline{x}\langle y\rangle.P \\ & | & P|P \\ & | & (\nu\, x)P \\ & | & !P \\ & | & \boldsymbol{nil} \end{array}}$$

Figure 2.2: The π-calculus grammar

The productions of this grammar describe the operations possible in the π-calculus:

Input Prefixing $x(y).P$
: A process can accept a name on x, then continue as P with the received name substituted for y .

Output Prefixing $\overline{x}\langle y\rangle.P$
: A process can output the name y on x, then proceed as P.

Parallel Composition $P \mid P$
: The two processes proceed in parallel, with their transition steps being interleaved arbitrarily.

Name Restriction $(\nu\, x)P$

A name x is introduced with scope restricted to the process P.

Replication $!P$

As many copies of P as needed to synchronise with every other process are run in parallel.

Nil Process ***nil***

No transitions are possible from this process.

π-calculus Semantics

The semantics of the π-calculus are defined by the structure of processes (Figure 2.2) together with the transition relation $\rightarrow$. Transitions between processes are defined up to *structural congruence*, an equivalence relation between processes that identifies semantically identical process with different grammatical structure. Capture-avoiding substitution of names is also defined.

The most important rule in the transition relation is the synchronisation rule that allows two concurrent processes to communicate:

$$\overline{x}\langle z\rangle.P \mid x(y).Q \rightarrow P \mid Q[z/y]$$

The left process performs an output, and the right process performs an input (on the same name). The two processes can then both transition, with Q having z substituted for y.

The full semantics of the π-calculus is given in Appendix D. Throughout the rest of the dissertation, all π-calculus semantics used are with reference to this full definition.

2.4.2 Core Features

Here I give a design for the concrete syntax of PCL, based on the π-calculus syntax in Figure 2.2.

This language design incorporates a version of each π-calculus operation in order that equivalent behaviour can be implemented. Examples of this syntax for PCL are given in Listing 2.1. I write `P` for an arbitrary program as defined by this syntax.

```
in x(y).P
out x(y).P
( P | P )
fresh x { P }
!( P )
end
```

Listing 2.1: Design for PCL syntax

The syntax `fresh x { P }` corresponds to the $(\nu\, x)P$ scope restriction operator—`x` is a fresh name (i.e. it does not appear anywhere else in the program) in the context `P`.

An example of a π-calculus process is:

$$P \triangleq (\nu\, x)(x(y).\overline{y}\langle w\rangle.\textbf{\textit{nil}} \mid \overline{x}\langle z\rangle.z(u).\textbf{\textit{nil}}) \tag{2.1}$$

This process P will reduce by the following sequence of transitions:

$$\begin{aligned}P &\rightarrow (\nu\, x)((\overline{y}\langle w\rangle.\textbf{\textit{nil}})[z/y] \mid z(u).\textbf{\textit{nil}})\\ &\rightarrow (\nu\, x)(\overline{z}\langle w\rangle.\textbf{\textit{nil}} \mid z(u).\textbf{\textit{nil}})\\ &\rightarrow (\nu\, x)(\textbf{\textit{nil}} \mid \textbf{\textit{nil}}[w/u])\\ &\rightarrow (\nu\, x)(\textbf{\textit{nil}} \mid \textbf{\textit{nil}})\\ &\rightarrow (\nu\, x)(\textbf{\textit{nil}})\\ &\rightarrow \textbf{\textit{nil}}\end{aligned}$$

A translation of P to PCL is given in Listing 2.2.

```
fresh x {
    (
        in x(y).
        out y(w).
        end
    |
        out x(z).
        in z(u).
        end
    )
}
```

Listing 2.2: A simple process translated to PCL syntax

This syntax is acceptable to read and write, and provides a useful starting point for discussion of how best the language can be extended and formalised.

2.4.3 Names, Data and Variables

Concrete Data

The π-calculus maintains only a single type of data—names. While this is theoretically adequate for performing arbitrary computation [7], it is conceptually very different to the way in which real-world programming languages (even those fundamentally based on communicating processes) handle data. If computation is restricted to only names, then any data a process operates on must be defined only in terms of names.

It is for this reason that it is desirable to add some other types of data to PCL, while still maintaining the core semantics based on synchronisation and name passing between concurrent processes. However, within the scope of this project, full support for the range of data types usually available in programming languages (e.g. strings, arrays, floating point numbers or structured types) will not be feasible to implement.

In place of a full range of data types, the proposed language will include support for integer data and computation. The operations and syntax available for operating on such values will serve as an example upon which a richer language and type system could be built.

The basic integer operations `+, -, *, /` should be incorporated into the language in some form, as well as syntactic support for literal integer values.

Variables

Having concrete data in the form of integer values now presents an incompatibility with the π-calculus model of pure names—how should integer values be included into the name passing model in a way that allows it to remain consistent with the π-calculus semantics?

The solution chosen is to have PCL depart from having only names. Instead, PCL will have two distinct data types—*channels* and *integers*. Channels can be seen as being analogous to names in the π-calculus—a link over which data can be sent or received.

When writing a program in PCL, values of each of these types can be introduced by a literal value. Elements of the channel type are identified by a string (their "name"), and written in PCL syntax as `@name`.

Substitution as it is defined for the π-calculus is not practical to implement directly, as it would require rewriting of bytecode sequences during execution. This is not desirable, and so an alternative mechanism for achieving indirection is needed.

The solution chosen to deal with indirection is to add *variables* to PCL that can store a single element of either type. These variables will be distinguished syntactically from channel literals by writing them in PCL syntax as `Name`.

This design no longer matches the π-calculus directly, but it is still possible to translate π-calculus processes into PCL—details of this are given in Table 4.1.

The language will not include explicit compile-time type checking for variables; any errors that occur (in a program with valid syntax) will do so at runtime. For example, an error will occur at runtime if the PCL expression `1 + V` is executed while `V` contains a channel.

2.4.4 Extending the π-calculus

Arithmetic

Described in Section 2.4.3 is the inclusion of integers to the language as a new data type. Naturally the language needs to be extended with operations on the integers—an obvious selection of operations is $+, -, *, /$, with $/$ representing truncated integer division.[2]

The language should be designed such that arbitrary combinations of numeric literals, variables and operators can be combined into a valid arithmetic expression. The language should respect the usual precedence rules for evaluation, and should allow for valid parenthesization of expressions (for disambiguation when reading, or for manual precedence assignment).

As the language allows expressions such as `1 / 0`, it is possible that the virtual machine will throw a runtime exception if such an expression is evaluated. This matches the behaviour of other language virtual machines such as the JVM or CPython, and is an acceptable design choice.

The way in which arithmetic expressions are evaluated internally will depend on the design of the virtual machine instruction set, though this choice will not affect the externally observable behaviour of any program.

Value Binding

A mechanism for assigning values to variables is necessary in the extended language.

[2]The fractional part of any division is discarded, e.g. $8/3 = 2$.

The language construct `fresh X { ... }` is such a mechanism—it introduces a variable `X` with a limited scope, with the semantic assertion that `X` is initially bound to a name unused anywhere else in the process (similarly to $(\nu\, x)$P).

A value-binding construct should behave similarly to `fresh`, but with the value of the variable being specified manually instead of being generated internally. The PCL syntax for value-binding is given in Listing 2.3.

```
1  let X = 0 {
2      let Y = 10 + X {
3          end
4      }
5  }.
6  out @a(X)
```

Listing 2.3: PCL value-binding syntax

Line 6 of Listing 2.3 represents a scope error—the variable `X` does not exist outside of the outer scope. If the code is executed, then an error should occur (possibly only at runtime, but this could in principle be checked at compile time).

Conditional Execution

Some descriptions of the π-calculus add a conditional process structure

$$[x = y]P$$

that proceeds as P if x and y represent the same name. I have not included this in my formal definition of the π-calculus, but as it represents a useful control structure for programming I have included it in the language. The PCL syntax for conditional checks is given in Listing 2.4.

```
[1 = 0] {
    end
}

let X = 1 {
    [1 = X] {
        end
    }
}
```

Listing 2.4: PCL conditional test syntax

Conditional tests will not cause type errors in PCL; if two values of different types are compared, the result is always false. This is the only form of conditional test—PCL has no "if–then–else" structure, or logical operations other than equality testing.

External Code

The project specification requires some mechanism to call external code from a PCL program. The most natural way to do this is to extend the idea of *send* and *receive* actions to this external code in some way.

The chosen design for this feature is to mark a set of channels as being *external*—if the program performs a send or receive action on one of these channels, some external code is called and the process can proceed without synchronisation.

If the external code were to be written in Scala, the code in Listing 2.5 gives an interface that external code would conform to—when a process performs a *receive* action on an external channel, it should send back a datatype representing either an integer or a channel identifier (and vice-versa). Using the type `Either[Channel, Long]` emulates dynamic typing for the data being sent or received.

```
trait ExternalChannel {
    def send(): Either[Channel, Long]
    def receive(data: Either[Channel, Long])
}
```

Listing 2.5: Interface to external code

Syntax for marking channels as external is given in Listing 2.6.

```
external @channel_name
```

Listing 2.6: PCL syntax for channels using external code

2.4.5 Summary

In this section I have presented an overview of the π-calculus and the way in which it performs computation. From this I have given a design for a language with syntax based on the π-calculus, but with additional features to allow for more useful programming.

2.5 Virtual Machine Design

At a high level, the virtual machine will consist of multiple threads of execution. These threads will be coordinated in order to implement synchronisation between *send* and *receive* actions. In this section I describe the execution model for one such thread, along with the strategy used to implement coordination between multiple threads.

2.5.1 Execution Model

A single thread of execution will execute a linear sequence of instructions, updating its own internal state as it does so.

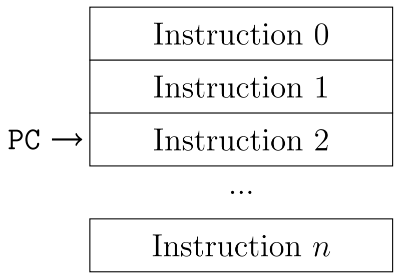

Figure 2.3: Instruction sequence with program counter.

The virtual machine will need to be able to compute arithmetic expressions, make conditional jumps (i.e. for control flow arising from conditional tests or parallel composition), manage synchronisation between communicating processes, and handle indirection through variables in order to implement the core semantics of the language described in Section 2.4.

It will be helpful to select a rich set of instructions. Designing a more general, low-level instruction set would greatly increase the amount of work done both at the design phase, and when implementing bytecode generation. Within the scope of the project, using a rich instruction set rather than a low-level one is an acceptable decision, as the core task of compiling to linearised bytecode can still be achieved.

2.5.2 Concurrency

Concurrency in the virtual machine will be implemented by having a *scheduler* coordinate the execution of many independent threads of execution. The platform-independent threading facilities provided by the Java standard library can be used to implement this model, with virtual machine execution threads being implemented by a single JVM thread.

There should be a rough correspondence between the executing JVM threads and the collection of concurrent π-calculus processes in the formal semantics (though this correspondence will not be exact, as the structure of the program representation will change when compiling to linear bytecode).

Each individual thread of execution is independent of the others—all coordination is done through the scheduler. Individual threads will be able to use the Java `wait` and `notify` primitives to block and resume execution, emulating π-calculus states where they have no transitions, and states in which a transition becomes available.

Chapter 3

Implementation

In this chapter I detail the work that was undertaken during the implementation of the project. Full definitions for the language and virtual machine instruction set designed, and the π-calculus itself can be found in Appendix A, Appendix B and Appendix D respectively.

After the sections on how the language and virtual machine instructions were designed, the sections in this chapter are arranged in roughly the same order as data passing through the application (i.e. beginning with program text, and ending with an executing program and output).

3.1 Lexical Analysis

The first operation that is performed when we compile a program is to tokenise or "lex" it into a sequence of atomic elements (tokens) based on the textual content of the file.

Lexing the program source performs a useful normalisation on the structure of the program (e.g. non-semantic whitespace is ignored), while also acting as a first pass of error checking.

Source Code
Lex
List of Tokens

Figure 3.1: Data transformation performed by the lexer

3.1.1 Token Definitions

To perform this lexical analysis, a formal definition of each token in the language is needed. These definitions are given as a regular expression for each token in the language. Some example definitions are given in Table 3.1, and the full lexical definition of the language can be found in Appendix A.1.2. All regular expressions are in the format prescribed by the Java standard library [17].

Token	**Definition**
`INT[n]`	`^(-?[0-9]+)(.*)`
`CHANNEL[name]`	`^@((?:[a-z]|_)+)(.*)`
`IN`	`^in(.*)`

Table 3.1: Example definitions of lexical tokens

The token definitions share some common structural elements that facilitate a simple lexing algorithm. Each one begins with ^ such that the expression will only match at the beginning of a string, and each one ends with (.*), a capturing group for the remaining string after the matched token. Additionally, the tokens that include associated data use another capturing group to extract this data.

3.1.2 Lexing Algorithm

The algorithm used to lex a program text into a list of tokens is conceptually simple—we attempt to match each token definition in turn against the program text, and if one succeeds we append the token to the list of tokens matched so far, then recurse for the remaining string (using the capturing group as defined above). An extract from the lexer source code is given in Listing 3.1.

```scala
def lex(p: String, a: List[Token]): List[Token] = {
    eat(p) match {
        case Some((t, r)) => tokenize(r, t::a)
        case None => p match {
            case "" => a reverse
            case s =>
                throw new LexException()
        }
    }
}
```

Listing 3.1: Extract from the lexing algorithm

In Listing 3.1, `eat(p:String)` is a function that attempts to apply each token's regular expression in turn, and returns the first one that matches together with the remaining string (if one does match).

Lexing succeeds when there is no remaining source to lex. If no token matches the remaining source, the program is malformed and an exception is thrown.

3.1.3 Preprocessing

In Section 2.4.4 the PCL syntax for designating external channels was given. The `external @channel` lines are not part of the lexical definition of the language, and so they are preprocessed into a collection of channel names before the lexer runs on the program itself.

The preprocessing performed at this step is comparatively simple compared to the lexing algorithm itself, and a full specification for `external` statements can be found in Appendix C.1.

3.2 Parsing

Once the source code has been tokenised, the next task is to *parse* the sequence of tokens into a structure that better represents the abstract syntax of the program. Most commonly (and indeed in this implementation), this structure is a recursively constructed tree.

Because the language syntax is minimal, I decided that the overhead of incorporating a parser generator (e.g. YACC, bison etc.) into my project would not be a useful strategy—writing a parser by hand would be simpler and more illustrative. The type of parser selected is a recursive descent parser for simplicity. Details of the algorithm used are given in Section 3.2.2.

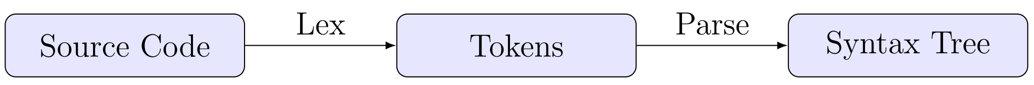

Figure 3.2: Data flow at the parsing stage

3.2.1 The Parse Tree

For simply structured languages, the construction of the parse tree will mirror the way in which the grammar is constructed (the full grammar, along with notational conventions for the language is given in Appendix A.2). In a general sense, clauses in the BNF grammar for generating a non-terminal will each represent a constructor for a subtree of the parse tree.

Because the language includes extensions beyond the pure form of the π-calculus, the grammar and matching parse tree designed are extended beyond that of the π-calculus. Modifications are also made in order to support recursive descent parsing.[1]

Arithmetic

The design of a formal grammar for arithmetic expressions is of little interest in the context of this project, and so the full details are omitted—the treatment of precedence and parenthesization are standard. The only non-standard component is the production allowing a channel name to be an expression, which serves to simplify the grammar in a number of places. The grammar fragment pertaining to arithmetic is given in Figure 3.3.

AddOp	::=	`+`
	|	`-`
MulOp	::=	`*`
	|	`/`
Factor	::=	`var`
	|	`int`
	|	`(` *Expression* `)`
Term	::=	*Factor Term′*
Term′	::=	*MulOp Factor Term′*
	|	ε
Expression	::=	*Term Expression′*
	|	`channel`
Expression′	::=	*AddOp Term Expression′*
	|	ε

Figure 3.3: Grammar for arithmetic expressions

[1]Specifically, the factoring of left recursion from the naïve grammar.

It is worth noting that the grammar has been factored to remove left recursion—this requirement is discussed in Section 3.2.2.

Processes

Because of the way in which the language is based on the π-calculus, the grammar for describing a language process is similar to the grammar given in Figure 2.2, though with extensions representing the additional language features. The language grammar for processes is given in Figure 3.4.

Name	::=	`var`
	|	`channel`
Process	::=	`out` *Name* `(` *Expression* `)` *Process′*
	|	`in` *Name* `(` `var` `)` *Process′*
	|	`(` *Process* | *Process* `)`
	|	`!` `(` *Process* `)`
	|	`[` *Expression* `=` *Expression* `]` { *Process* } *Process′*
	|	`let` `var` `=` *Expression* { *Process* } *Process′*
	|	`fresh` `var` { *Process* } *Process′*
	|	`end`
Process′	::=	`.` *Process* *Process′*
	|	ε

Figure 3.4: Grammar for language processes

Similarly to the grammar for arithmetic, left recursion occurs in the naïve translation of the π-calculus grammar—this has been factored to give the grammar in Figure 3.4. The grammar itself is not complicated; it extends the π-calculus grammar with variable names and arithmetic literals where they can be used, and adds productions for value binding and variable name restriction.

3.2.2 Parsing Algorithm

As discussed previously, the easiest and simplest algorithm for parsing the language grammar is recursive descent. Conceptually, a recursive descent parser is an algorithm directly encoding the grammar of the language —

functions are defined to match each kind of non-terminal symbol. These functions "consume" terminal symbols (tokens) and recurse into other matching functions, eventually returning a parsed representation of the program.

Listing 3.2 gives the implementation of a non-recursive matching function (for matching Name non-terminals, which can only expand to a single terminal). In this function, we reach the base case of the recursive descent, returning a subtree structure dependent on what the current token is. The `advance()` method simply "consumes" a token and moves to the next one.

```
def matchName(): Result[Name] = {
    currentToken() match {
        case Some(VarName(vn)) =>
            advance()
            Left(VariableName(vn))
        case Some(ChannelName(cn)) =>
            advance()
            Left(ChannelName(cn))

        case _ =>
            syntaxError()
    }
}
```

Listing 3.2: Matching function for Name non-terminals

```scala
def matchExpression(): Result[Expression] = {
    currentToken() match {
        case Some(ChannelName(cn)) =>
            advance()
            Left(ChannelExpression(ChannelName(cn)))
        case Some(_) =>
            val term = matchTerm()
            val aux = matchExpressionAux()
            (term, aux) match {
                case (Left(t), Left(a)) =>
                    Left(TermAuxExpression(t, a))
                case (Right(e), _) => Right(e)
                case (_, Right(e)) => Right(e)
            }
        case _ => syntaxError()
    }
}
```

Listing 3.3: Matching function for Expression non-terminals

The recursive descent parser is implemented by creating a matching function for every non-terminal class in the grammar to be parsed. These functions are individually responsible for implementing all the production rules that can produce the relevant non-terminal, and for returning an appropriate structure to represent the data they have parsed.

3.3 External Code

In this section I describe the implementation of a mechanism by which external code is called by programs written in PCL. External channels for PCL are written as Scala classes that implement the interface given in Listing 3.4.

3.3.1 Class Loading

Because the project is written in Scala, it is possible to access and use the Java standard library. This includes the ability to dynamically load compiled `.class` files at runtime, with one major caveat—type checking information is lost when using this method.

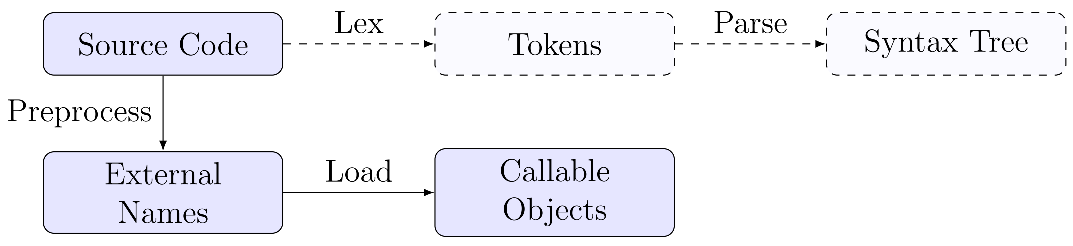

Figure 3.5: Data flow at the external code loading phase

```scala
trait ExternalChannel {
    def send(): Either[Channel, Long]
    def receive(data: Either[Channel, Long])
}
```

Listing 3.4: Interface to external code (repeated)

An object with the `ExternalChannel` trait (from Listing 3.4) can be extracted from a loaded class file by using reflection and unsafe type casts. This is because methods of dynamically loaded classes always have the return type `Any`—to use these values elsewhere in the implementation, an unsafe cast of the returned value to `Either[Channel, Long]` is necessary.

3.3.2 Implementation

The steps taken to load external code before a program is executed are:

1. For every external channel name identified by the preprocessing step in Section 3.1.3, load a corresponding `.class` file from the directory specified in Appendix C.1. If any do not exist, exit with an error.

2. Load the class files using the Java standard library, and use reflection to extract references to the `send` and `receive` methods.

3. Cast the references to `send` and `receive` to their type as specified in Listing 3.4. If this is not possible, exit with an error. The references can then be used to construct type-safe objects (that have methods that call `send` and `receive`). These objects are then stored in a map from channel names to objects implementing `ExternalChannel`.

This map of channel names to objects implementing is then available to the virtual machine during execution. The way in which the loaded code is executed is described along with other elements of the program interpretation in Section 3.6. The exact syntax and search paths for compiled libraries are given in Appendix C.1.

3.3.3 Problems and Improvements

The external code interface in the project serves primarily as a proof-of-concept that such an interface is possible to implement. There are some obvious issues with the strategy—the loader will simply fail at runtime if the `.class` files are not present, or if the loaded class file does not satisfy the interface. Additionally, the mechanism is insecure—arbitrary compiled Java bytecode can be executed by programs in the language. Fixing these issues is beyond the scope of the project; to do so would be possible, but not relevant to the success criteria.

3.4 Instruction Set

This section gives the definition of a full instruction set for the virtual machine—the algorithms used internally to implement the design are addressed in full in Sections 3.6.2 and 3.7. A full reference to the instruction set is given in Appendix B.

In general, when referencing instructions in Appendix B, some shared behaviour is assumed—any instruction that pops an item from the arithmetic stack or looks up a variable name in the store will throw a runtime error if the stack is empty, or the name is not in the store.

3.4.1 Arithmetic

The way in which the virtual machine is to handle arithmetic is one of the less interesting parts of the design and implementation, and as such a simple design was implemented—a stack machine[2] where operands are pushed onto the stack, and the top of the stack is manipulated by operations. The arithmetic operations are given in Appendix B.1.1.

3.4.2 Environment & Variables

Each virtual machine thread will maintain a data structure mapping variable names to their corresponding values. An alternative approach would be to implement some kind of register-based system, but because the virtual

[2]i.e. a direct translation of Reverse Polish Notation.

machine is implemented in a managed language, the resulting complexity is unnecessary.

The store instructions handle a number of different cases depending on the type of data being assigned, as well as the target of the assignment. For example, the PCL assignment `let X = 0` corresponds to the `STORE-INT` instruction, while `let Y = @c` corresponds to `STORE-CHANNEL`. A full reference to the store instructions is given in Appendix B.1.2.

3.4.3 Control Flow

The instruction set requires instructions to alter the way in which the program counter moves through the instruction sequence. In particular, linear flow of control may be deviated from by conditional execution, termination, or execution of a parallel process. Examples of PCL code that causes non-linear control flow are given in Listing 3.5.

```
[X = 0] { end }

( out @x(0) | in @x(Y) )
```

Listing 3.5: Non-linear control flow in PCL code

The instructions for control flow are based on the value on top of the stack—jumps can be made conditional on the top of the stack being either zero or non-zero (`JUMP-ZERO` and `JUMP-NON-ZERO`). Comparison of two arithmetic expressions is not encoded directly; instead each expression is evaluated and a subtraction performed to determine equality. A full reference to control flow instructions is given in Appendix B.1.3.

3.4.4 Synchronisation

A program can block if it reaches a state where it must either send or receive, but no concurrent process is executing the dual action. When this happens in the π-calculus, no further transitions can occur until a synchronisation is available.

Based on this behaviour, the virtual machine instruction set includes instructions that trigger this blocking and synchronisation (all are variants of `SEND` or `RECEIVE`). The mechanisms by which virtual machine processes will block and resume execution when necessary are given in Section 3.7.

With these instructions, there is a high degree of repetition between instruction variants. This approach makes each individual instruction simpler and more explicit in its intent than an overloaded equivalent would be. Each instruction has a *direct* and an *indirect* variant—the direct variant has the relevant channel name encoded in the instruction arguments, while the indirect variant has a variable name that is used to look up the relevant channel.[3]

A full reference to the instructions supporting synchronisation between two concurrent threads of execution is given in Appendix B.1.4 (for example, `RECEIVE-DIRECT` or `SEND-INT-INDIRECT`).

3.5 Code Generation

Given an abstract syntax tree, the next task of the compiler is to perform code generation[4] to transform the tree structure into a linear sequence of bytecode instructions ready for execution. This can be achieved by traversing the structure of the tree and emitting code based on a set of recursively defined rules. In this section I describe the mechanism by which bytecode is generated using this strategy.

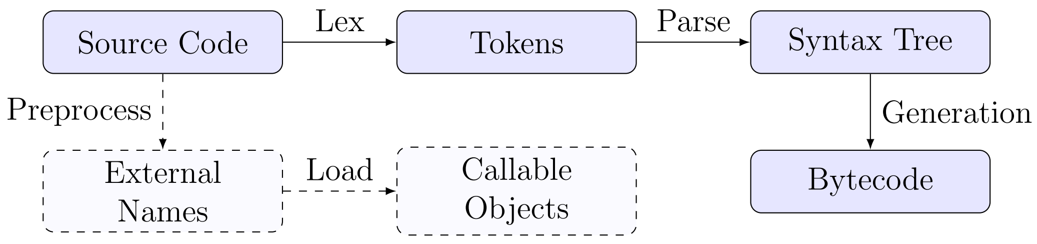

Figure 3.6: Data flow at the code generation phase

3.5.1 Example Generation Rules

In general, the pattern followed by the code generator is of a recursive traversal of the abstract syntax tree. Rules generate bytecode with a fixed format that contains "holes"—these holes are then filled compositionally by the generated bytecode from subphrases.

The full definition of most of the code generation rules includes a lot of pattern matching on subtree structures to determine how code is generated,

[3]c.f. the difference between `add r1,r2` and `addi r1,#n` in ARM assembler.

[4]For more complex languages, there may well be additional transformations or analyses performed as intermediate steps before code generation—the scope of this project is small enough that code generation can be performed directly from the syntax tree.

and as such the example rules given here are slightly simplified. For brevity, not all code generation rules are given.

Let Block

An illustrative example is that of the `let Var = Expr { Proc }` construct[5] from the grammar in Figure 3.4. The structure for this construct is similar to the tree in Figure 3.7 (arbitrary subtrees or supertrees are written as "...").

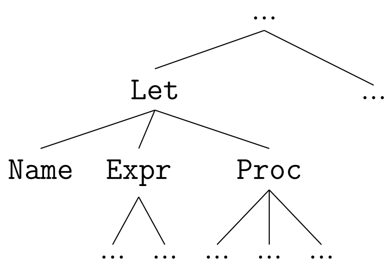

Figure 3.7: Syntax tree fragment for a let-binding

From Figure 3.7, the subtrees that need to have their code generated are:

Name is the variable name to which a value is being bound.

Expr is the expression that will evaluate to the value to be assigned.

Proc is the process to execute with the newly bound variable in scope.

I write $[\![\texttt{P}]\!]$ for the bytecode generated from the PCL program (or program fragment) `P`. There are three cases to consider when generating code for `Expr`—it can be an arithmetic expression that pushes a value onto the stack, a channel literal or a variable. The specialised bytecode generated for each of these different cases is given in Table 3.2. Each column of the table contains the sequence of instructions generated for a particular specialisation of the assignment. The cases do share some common structure—each one executes some code to set the value of a variable, then executes `Proc`, then deletes the variable assigned (to enforce variable scoping).

Parallel Process

A process that performs parallel composition serves as a good example of how control flow can be performed using the virtual machine architecture. The syntax tree we are aiming to generate code for is given in Figure 3.8.

[5]Analogous to ML-style let binding of variables.

Let-Block		
Arithmetic	**Channel**	**Variable**
⟦Expr⟧	STORE-CHANNEL[chan,name]	COPY[expr,name]
STORE-INT[name]	⟦Proc⟧	⟦Proc⟧
⟦Proc⟧	DELETE[name]	DELETE[name]
DELETE[name]	⟦Next⟧	⟦Next⟧
⟦Next⟧		

Table 3.2: Generated bytecode for let-binding of variables

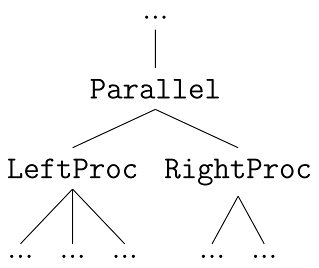

Figure 3.8: Abstract syntax tree fragment for parallel composition

Because the generated code will perform control flow, labels are needed to allow for jumping—the code generator provides a mechanism by which globally unique labels can be generated as needed. For clarity, the labels used in this example are ℓ_1, ℓ_2, ℓ_3 etc. in sequence as needed. The code generated for this syntax tree fragment is given in Table 3.3.

The code layout defines two regions demarcated by `LABEL` and `END` instructions—the `SPAWN` instructions then direct control flow concurrently into these regions. The "parent" thread of execution then terminates (the `END` instruction causes the executing thread to immediately terminate). Note that for symmetry and ease of understanding, the generated code performs an unnecessary `SPAWN` instruction—the code could be generated such that the parent thread spawns from ℓ_1, then executes `RightProc` itself. The behaviour of the generated code in Table 3.3 is the same as this version, though slightly less efficient.

3.6 Interpretation

In this section I describe the implementation of the interpreter, and how a compiled sequence of bytecode instructions can be executed. This section

Parallel Composition
`SPAWN[`ℓ_1`]`
`SPAWN[`ℓ_2`]`
`END`
`LABEL[`ℓ_1`]`
$[\![$`LeftProc`$]\!]$
`END`
`LABEL[`ℓ_2`]`
$[\![$`RightProc`$]\!]$
`END`

Table 3.3: Generated bytecode for parallel composition of processes

deals only with a single thread of execution—the way in which parallelism and concurrent execution is implemented is described in Section 3.7.

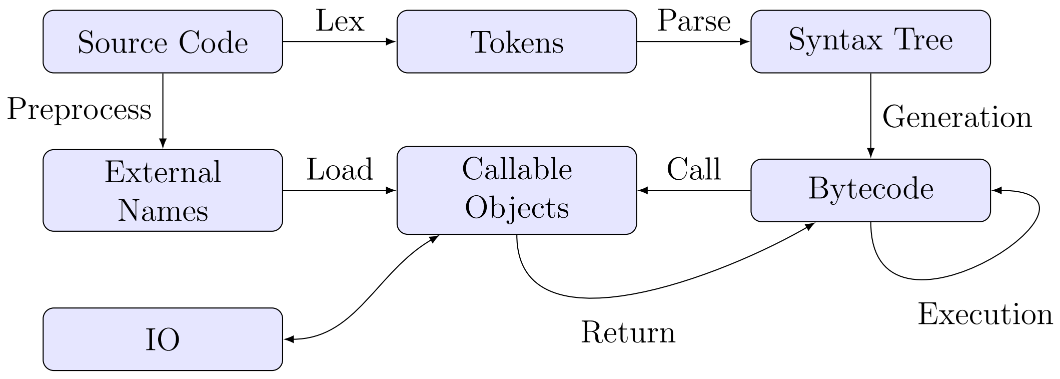

Figure 3.9: Data flow at the execution phase

In Figure 3.9, the full data flow diagram can be seen—using the external code interface, some bytecode instructions may result in a call to external code, then a return to the bytecode (possibly with an associated value). The diagram also encodes the fact that the externally loaded code may have side effects (e.g. arbitrary input and output). These side effects may affect the values sent over the external code's interface.

3.6.1 Core Algorithm & Interpreter State

The code generation phase generates a sequence of bytecode instructions. Program execution is then implemented by iterating through the sequence of instructions, updating the interpreter state and performing other actions as necessary.

The state that is stored by an interpreter is:

Program Counter is the the current index into the instruction sequence.

Environment stores a mapping from variable names to their respective values. As instructions are executed by an interpreter thread, the state of this mapping changes.

Label Map stores a mapping from label names to program indices to allow for jumping and spawning by name. It does not change as the program executes.

The label map for a program is extracted directly from the bytecode by iterating through the sequence, storing a new map entry every time a `LABEL` instruction is reached.

The program counter and environment are local to a single interpreter instance (i.e. they do not act as shared memory for cross-thread communication). The behaviour of each instruction (as described in Section 3.4) is encoded in a lookup table.

The core interpreter algorithm can therefore be described by the following steps (at the beginning of execution, the environment is empty, the program counter is 0, and the label map has been initialised):

1. Look up the current instruction in the behaviour lookup table to get a function. This function can be executed to update the interpreter state as appropriate.

2. Execute this function with any necessary parameters (e.g. variable names).

3. Update the environment and program counter if necessary.

4. Repeat if execution was not halted by an `END` instruction.

3.6.2 Replication Strategy

One π-calculus construct requires special treatment by the virtual machine—the replication operator $!P$ creates as many copies of the process P as necessary. Any number of parallel processes can synchronise with the replicated process. The completely naïve solution is to repeatedly spawn the replicated process, but this fails when threads "build up"—the system will run out of memory.

The solution chosen for the virtual machine is to allow for the number of active threads to increase to a fixed bound—at this level, any blocked threads with identical blocking conditions are pruned down such that only one remains. The mechanism that allows this is detailed in Section 3.7, along with a full definition of blocking conditions.

Alternative options that would be easier to implement for different virtual machine designs include a *lazy* model, where replicated threads are only spawned when necessary—this approach would be a possible efficiency improvement to the interpreter, but would require a different interpreter and instruction set design. As such this approach was not implemented.

3.7 Scheduling and Multithreading

In this section I describe the implementation of the scheduler—the component of the system that manages concurrent execution of multiple interpreters. In particular, the concurrency and shared data model (based on the Java standard threading library) that allows multiple interpreters to synchronise is described.

3.7.1 Scheduler

Each thread of execution as described in Section 3.6 is independent of all the others, and as such a mechanism that allows them to communicate with each other is needed. The core behaviours required from this mechanism are:

- Allow individual interpreters to block when they execute a `SEND-` or `RECEIVE-` instruction.
- When a matched pair of `SEND-` and `RECEIVE-` instructions are executed by concurrent interpreters, allow the two interpreters involved to resume their execution.

To achieve these goals, I implemented a global scheduler object that becomes the main entry point for bytecode execution—when a program is compiled, the bytecode is passed to the scheduler, which spawns an interpreter thread at the start of the bytecode.

The virtual machine `SPAWN` instruction can therefore be implemented by having interpreters call a method on the scheduler. This method will spawn another interpreter at a given index into the bytecode. The `END` instruction can be implemented similarly—the thread notifies the scheduler that it is terminating, then terminates.

It is also necessary for the scheduler to keep track of all the currently executing interpreters—it does this by adding each interpreter object it creates to a collection, and deleting them when they execute an `END` instruction.

3.7.2 Synchronisation

In this section I describe the implementation of synchronisation between two concurrent threads of execution (i.e. how the scheduler unblocks blocked threads).

Before synchronisation can be implemented, an implementation of blocking and resumption for interpreters needs to be introduced—a simple and effective way to do this is to have threads call their Scala `wait()` method when they execute a blocking instruction (any `RECEIVE` variant), and to have the scheduler call (again in Scala) `t.notifyAll()` when resuming the thread `t`.

In order to allow for synchronisation between interpreter threads, the blocking action for each interpreter should be stored along with the interpreter itself in the scheduler's collection (if one exists). When an interpreter blocks, it registers its blocking action with the scheduler, then suspends itself as described above.

The implementation used for the scheduler is in fact partly a naïve one—it has blocking instructions be restricted to `RECEIVE` instruction variants, and has `SEND` variants cause a loop on the interpreter thread. This loop iterates through the collection of interpreters to see if any are blocked on the corresponding action—if they are, their `notifyAll()` method is called and the sent value passed to them. This ends the loop and allows the sending interpreter to resume.

This implementation is somewhat naïve—the sending thread enters a busy loop, which is an inefficient way of solving the problem. However, the correct semantics are achieved by using this method despite the inefficiencies. A more prudent strategy would be to have all send and receive actions block the interpreter, and have the scheduler insert these actions into a collection. A check for matching pairs of actions could then be performed as new actions arrive. Because efficiency is not a primary concern of the project, and the desired semantics are met by the naïve version, this improvement was not implemented.

3.7.3 Treatment of Nondeterminism

The nondeterministic choice of an execution path is at the core of how the π-calculus executes—in this subsection I discuss the way in which the scheduler makes nondeterministic decisions of this kind.

Because any choice of execution path at a given choice point is valid in the π-calculus semantics, essentially any strategy can be used to choose a path — even seemingly “incorrect” ones (e.g. always choose a certain direction). However, such a strategy will not provide a good illustration of how a program in the language works. That is, it is desirable to pick a strategy that provides a representative sample of the possible choices.

In fact, this strategy is simple to implement—the nondeterminism that arises from the interleaving of instructions between different threads is sufficient to provide an acceptable sampling. In other words, the responsibility for nondeterministic choice is pushed down the implementation stack into the Java threading library and how it enforces context switches. In practice this is an acceptable decision—there is no way to reliably predict the interleaving of Java threads. A possible improvement to the project would be to implement some way of forcing the scheduler to resolve nondeterministic choices in a particular pattern.

3.8 User Interface

A command-line user interface for the compiler was implemented. The interface allows the user to invoke the compiler on a PCL source file, then run the compiled program. It also optionally gives debug output.

Assuming the executable is available on the system path as `pcl`, the compiler can be invoked on a source file `source.pcl` as:

```
$ pcl source.pcl [--dump] [--trace]
```

The `--dump` flag has the compiler print the internal representation (list of tokens, parse tree and generated bytecode) of the program as it is compiled. The `--trace` flag causes each interpreter thread to print the instructions that it executes—this gives an insight into how processes are interleaved during execution. A full reference to the usage of the executable program is given in Appendix C.

3.9 Summary

From the language and virtual machine designs given in Chapter 2, I have presented the implementation of a compiler that can transform a program written in this language into a sequence of bytecode instructions for the virtual machine.

Additionally, I have given the execution semantics of a single virtual machine interpreter that can execute a sequential program of bytecode instructions. Based on this single-thread design, I have described the implementation of a scheduling mechanism that allows multiple interpreters to execute concurrently and synchronise with each other. Finally, I described a command-line user interface to the compiler.

Chapter 4

Evaluation

In this chapter I discuss the strategies and methods used to evaluate the success of the project. The rationale behind these methods will be explained, and the results from the evaluation presented. Possible improvements to the project and evaluation strategy will also be examined.

4.1 Evaluation Strategy

In this section I describe a strategy to evaluate the project with respect to the success criteria laid out in the original project proposal (included at the end of the dissertation).

4.1.1 Success Criteria

The original proposal stated that the project would be a success if the following criteria are met:

- To be able to demonstrate the compilation of a suite of programs that demonstrate the capabilities of my language.
- To be able to demonstrate the execution of such programs by the virtual machine that I implement.
- My implementation will be judged as a success if these programs are shown to be behaviourally equivalent to the corresponding process in the π-calculus.

4.1.2 Compiler and Runtime Demonstration

With reference to the success criteria given above, the first two criteria specify that a demonstration of some kind is provided. As summarised in Section 3.8, the `--dump` and `--trace` command-line arguments can be used to print the internal program representations[1] and execution trace respectively.

Using these features I demonstrate how the compiler transforms the source code and executes the compiled program. A representative output is given in Listing 4.1.

```
$ cat input.pcl
external @stdio

in @stdio(Input).
out @stdio(Input)
$ pcl input.pcl --dump --trace
Tokens:
List(
    In(),
    ChannelName(stdio),
    OpenBracket(),
    VarName(Input),
    CloseBracket(),
    Sequential(),
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    VarName(Input),
    CloseBracket()
)

Parse Tree:
ProcessStart(
    InProcess(
        ChannelName(stdio),
        VariableName(Input),
        SequentialProcessAux(
            OutProcess(
                ChannelName(stdio),
```

[1]List of tokens, abstract syntax tree and bytecode listing.

```
                TermAuxExpression(
                    FactorAuxTerm(
                        VariableFactor(
                            VariableName(Input)
                        ),
                        EmptyTermAux()
                    ),
                    EmptyExpressionAux()
                ),
                EmptyProcessAux()
            ),
            EmptyProcessAux()
        )
    )
)

Bytecode:
1: ReceiveDirect(Channel(stdio),Variable(Input))
2: SendVariableDirect(Channel(stdio),Variable(Input))
3: End()

Thread 0: ReceiveDirect(
            Channel(stdio),Variable(Input))
> a
Thread 0: SendVariableDirect(
            Channel(stdio),Variable(Input))
a
Thread 0: End()
$
```

Listing 4.1: Example debug output

The program `input.pcl` reads an input from the user on standard input (this is the side effect associated with receiving from `@stdio`), then prints it back to standard output. Larger example programs produce debug output that is much too large to be included here.

The command-line interface to the compiler and runtime can also be used to demonstrate invalid programs being faulted. Listing 4.2 shows an example program with invalid syntax (there is no `.` between the first and second lines):

```
in @x(Y)
out @x(0)
```

Listing 4.2: Example program with a syntax error (`syntaxerr.pcl`)

The compiler error is given in Listing 4.3 below. The usability of error messages given by the compiler was not an implementation priority, and so error messages can be somewhat unhelpful.

```
$ pcl syntaxerr.pcl
Parse Error: tokens remain at end of parse.
```

Listing 4.3: Compiler output for a syntax error

As well as the static information, the debug dump in Listing 4.1 contains an instruction-by-instruction execution log of the program. For deterministic programs, this simply mirrors the bytecode listing; for nondeterministic programs it can be used to observe the interleaving of interpreter threads.

Using the debug output options made available by the compiler, I am able to demonstrate compilation and execution of arbitrary well-formed programs in the language, therefore meeting the first two success criteria. The strategy for evaluating the first two criteria is to provide compiler output for valid programs to demonstrate how internal representations are transformed, and to provide execution logs of these compiled programs.

4.1.3 Correspondence With the π-calculus

While the first two success criteria are satisfied by a demonstration of the compiler's debug output mode for a variety of different programs, the third criterion (correspondence with the abstract semantics of the π-calculus) is more subtle to define, and therefore harder to test and evaluate properly.

Providing a formal proof of correspondence with the π-calculus is unfortunately beyond the scope of the project. To demonstrate the correspondence with the π-calculus, I will instead provide a syntactic embedding of the calculus in PCL. Using this embedding, I show transition derivations for process expressions, then provide execution logs of the corresponding PCL programs that demonstrate equivalence.

There are some subtleties to consider with this approach. The first is that the π-calculus has no explicit mechanism to denote external behaviour, and so such a mechanism would have to be added to the transition semantics of the calculus in order to reason about programs that have side effects.

The second subtlety is that execution of a program on the virtual machine is nondeterministic—it may produce different effects each time it is executed. To account for this, we will need to consider all the possible transition paths of the π-calculus programs we use, as well as providing execution logs that give evidence for each possible transition path.

Finally, there are many different programs that produce the same external effects—while still not a formal method of verification, it will be useful to create programs representing each step of a process' transition sequence. These reduced programs can all be checked against the abstract semantics to provide a model for the full transition / execution correspondence.

The third success criterion will be achieved if I am able to demonstrate an translation of the π-calculus to PCL, such that the techniques described above can be used to show correspondence over all the syntactic categories and transition rules of the π-calculus.

4.2 Results & Analysis

4.2.1 Compiler & Execution Demonstration

The first two success criteria (demonstration of the compiler and interpreter) can be demonstrated by use of the compiler flags as described in Section 4.1.2. The strategy for evaluating these criteria has already been described in detail—a full set of examples demonstrating compilation and execution of a variety of programs can be found in Appendix E.

4.2.2 π-calculus Correspondence

In this section I will give a mapping from π-calculus processes to PCL programs, along with a method for testing the behavioural correspondence between a process and a program, such that the third success criterion (correspondence with the π-calculus semantics) can be demonstrated.

π-calculus Embedding

Here I present an embedding of the π-calculus syntactic terms into the definition of the language. In Section 2.4 I gave an informal motivation for the syntactic design of my language, based on the grammatical classes of the π-calculus (fully defined in Section 2.4.1).

To demonstrate this embedding, I will provide a complete mapping of any π-calculus process P to a PCL program $\mathcal{C}(P)$. The mapping is specified fully in Table 4.1. As well as this mapping for processes, each name $e \in E$ maps to an initial line `external @e`. Note that because the PCL `let` syntax is an additional programming convenience, no π-calculus process maps to a PCL program containing a `let`-binding.

π-calculus Process	$\mapsto$	**PCL Program**
$x(y).P$ where x bound	$\mapsto$	`in X(Y).`$\mathcal{C}(P)$
$x(y).P$ where x free	$\mapsto$	`in @x(Y).`$\mathcal{C}(P)$
$\overline{x}\langle y\rangle.P$ where x, y bound	$\mapsto$	`out X(Y).`$\mathcal{C}(P)$
$\overline{x}\langle y\rangle.P$ where x, y free	$\mapsto$	`out @x(@y).`$\mathcal{C}(P)$
$\overline{x}\langle y\rangle.P$ where x bound, y free	$\mapsto$	`out X(@y).`$\mathcal{C}(P)$
$\overline{x}\langle y\rangle.P$ where x free, y bound	$\mapsto$	`out @x(Y).`$\mathcal{C}(P)$
$P \mid Q$	$\mapsto$	`(` $\mathcal{C}(P)$ `|` $\mathcal{C}(Q)$ `)`
$(\nu\, x)P$	$\mapsto$	`fresh X {` $\mathcal{C}(P)$ `}`
$!P$	$\mapsto$	`!(` $\mathcal{C}(P)$ `)`
nil	$\mapsto$	`end`

Table 4.1: Translation of the π-calculus to PCL

Note that because of the difference between π-calculus names and language names (atoms and variables), a number of different cases to handle free and bound names are required. If a name is bound in a π-calculus process, it maps to a PCL variable, and if it is free, it maps to a PCL channel literal.

Using the embedding given, there is therefore an algorithmic method for converting a π-calculus process into a language program. Additionally, well-formed PCL programs can be mapped back into the π-calculus.

Extended π-calculus Semantics

As described in Section 4.1.3, it will be necessary to provide a slightly extended semantics for the π-calculus that allows for external actions to be reasoned about. Based on the embedding described above, an external action can only occur when an `in` or an `out` statement is made on a designated `external` channel. These channels can only exist in a program if they are declared as such at the beginning of the program. To relate these channel names back to the abstract semantics, we will associate a set E of external channel names with every π-calculus process P. Names in E are treated as free names when converting a π-calculus process to a PCL program.

The transition relation for the π-calculus is modified to reference E when necessary. Firstly a new rule is added that allows a process to undergo a *labelled transition* if it acts on a name in E:

$$\begin{aligned} x(y).P &\xrightarrow{x(y)} P && \text{if } x \in E \\ \overline{x}\langle y\rangle.P &\xrightarrow{\overline{x}\langle y\rangle} P && \text{if } x \in E \end{aligned}$$

This models the way in which PCL programs can proceed without explicit synchronisation. Additionally, synchronisation on external channels is prohibited by modifying the existing transition relation from the π-calculus:

$$\overline{x}\langle y\rangle.P \mid x(z).Q \rightarrow P \mid Q[y/z] \qquad\qquad \text{if } x \notin E$$

Informally, these modifications mean that communication on an external channel name must proceed only by the corresponding external action. When discussing the correspondence in subsequent sections, external actions as described above will be identified with the particular side effects of calling given external code (as described in Appendix C.1). Generally this will be the `@stdio` channel used to read from standard input and write to standard output.

Representative Example

Given the embedding in Table 4.1 and the modified operational semantics above, it is now possible to demonstrate a correspondence between the language and the π-calculus. This is done by constructing processes in the π-calculus, deriving their transition sequence, then observing the behaviour of the corresponding program in the concrete language (for each process in the transition sequence).

For brevity, only one representative example will be included in this section to illustrate the method of demonstration. Further examples using the same style of demonstration are included in Appendix E.

The first example of correspondence demonstrates synchronisation between parallel processes, and serves to illustrate the strategy used for the demonstration. We define:

$$\begin{aligned}&E \triangleq \{stdio\}\\&P \triangleq x(y).\overline{stdio}\langle y\rangle.\boldsymbol{nil} \mid \overline{x}\langle z\rangle.\boldsymbol{nil}\end{aligned}$$

P synchronises on x, then performs an external output action on *stdio*. This is formalised by demonstrating the transition sequence for P:

$$\begin{aligned}P &\triangleq x(y).\overline{stdio}\langle y\rangle.\boldsymbol{nil} \mid \overline{x}\langle z\rangle.\boldsymbol{nil}\\&\to \overline{stdio}\langle y\rangle.\boldsymbol{nil}[z/y] \mid \boldsymbol{nil}\\&\to \overline{stdio}\langle z\rangle.\boldsymbol{nil} \mid \boldsymbol{nil}\\&\xrightarrow{\overline{stdio}\langle z\rangle} \boldsymbol{nil} \mid \boldsymbol{nil}\\&\to \boldsymbol{nil}\end{aligned}$$

The PCL program corresponding to P is given in Listing 4.4.

```
external @stdio

(
    in @x(Y).out @stdio(Y).end
|
    out @x(@z).end
)
```

Listing 4.4: Example demonstrating the correspondence between the π-calculus and PCL

Note that in the transition sequence for P, a single labelled transition $\overline{stdio}\langle z\rangle$ occurs. Transitions of this kind are identified with the external side effect "the string `z` is printed to the console". An execution log for the program in Listing 4.4 is given in Listing 4.5.

```
$ cat simple.pcl
external @stdio

(
    in @x(Y).out @stdio(Y).end
|
    out @x(@z).end
)
$ pcl simple.pcl
z
```

Listing 4.5: Output log an example of the correspondence between the π-calculus and PCL

As described previously, it is possible to encode the process in the language after it has taken on transition step, still keeping the same semantics. This is demonstrated in Listing 4.6. Note that for `simple-2.pcl`, the reduction sequence no longer includes the $\overline{stdio}\langle z\rangle$ action, and so no externally observable behaviour is seen. In these examples, `simple-`$[n]$`.pcl` is the program corresponding to the n-step reduced version of the process that mapped to `simple.pcl`.

```
$ cat simple-1.pcl
external @stdio

(
    out @stdio(@z).end
|
    end
)
$ pcl simple-1.pcl
z
$ cat simple-2.pcl
external @stdio

(
    end
|
    end
)
$ pcl simple-2.pcl
$
```

Listing 4.6: Output logs for a reduced process

The full set of demonstrative examples is available in Appendix E, with each example following the same format as described here—a π-calculus process is mapped to a language program, then compared at each reduction step to the behaviour of the corresponding program.

4.3 Interpreting the Execution Logs

In this section I summarise the larger example programs and demonstrations found in Appendix E. As initially laid out in the the project proposal, the three types of demonstration program that would be required are:

- Behavioural correspondence by comparison of program behaviour with the π-calculus semantics.
- Some implementations of numerical algorithms to show the language being used for practical programming tasks.

- A sample encoding of a data structure—the ones given by Milner in his original text on the π-calculus were cited as possibilities.

4.3.1 Behavioural Correspondence

The first section of Appendix E contains a number of demonstrations of simple execution semantics. The demonstrations are broken down by the rules of the transition relation as defined in Appendix D.

Nondeterminism demonstrates the possibility for nondeterministic execution ordering in PCL. A π-calculus process with two possible execution paths is given, and execution logs for the corresponding program (`par.pcl`) provide evidence that both paths may occur during execution. Formal step-by-step derivations of possible behaviours are given for this example (in the same way as for `simple.pcl` above). Additionally, full debug output is provided to demonstrate the compiler's internal representations and the virtual machine's execution paths.

Restriction demonstrates execution proceeding under a restriction. This example (`res.pcl` and `nres.pcl`) is less complex than `par.pcl` as it is deterministic, and so only a single reduction sequence can be observed. Similarly, full compiler and runtime debug output is given.

Congruence collects a number of smaller examples (`assocl.pcl`, `assocr.pcl`, `xy.pcl`, `yx.pcl`, `resn.pcl`, `scope.pcl` and `scopeext.pcl`) that demonstrate each of the laws of structural congruence. Each rule is demonstrated by giving two structurally congruent processes (except for `reverse.pcl`, which is structurally congruent to `par.pcl`), along with the corresponding PCL programs. Execution logs are provided for these programs to demonstrate that they do in fact exhibit equivalent behaviour.

Together this set of examples provides a strong circumstancial demonstration that the semantics of the π-calculus transition relation are implemented correctly by the language, therefore meeting the first success criterion.

4.3.2 Algorithms

The second section contains implementations of several useful numerical algorithms and functions. Implemented are:

- Sign function
- Modulus function

- Exponentiation
- Greatest Common Divisor
- Fibonacci Numbers

Each of these algorithms is given as a language program together with a reference implementation written in Python.[2] For each algorithm, execution logs of the language program are compared to the reference implementation to test correctness on well-formed inputs.

These algorithms give evidence that the language is capable of performing useful numerical computation in a potentially recursive way, therefore meeting the second success criterion.

4.3.3 Data Structure

The final section of Appendix E gives an encoding of the Church numerals in the language, using a similar style to the numerical algorithms for encoding functions (i.e. as infinitely replicated processes that accept arguments via input prefixing).

By giving such an encoding as an example of how λ-terms might be encoded in the language, the third success criterion for this section is met.

4.4 Possible Improvements

While I have demonstrated that the project does in fact meet all three of the success criteria laid out in the original project proposal, there are several ways in which improvements could be made (but were not possible to incorporate due to time and scope constraints).

4.4.1 Animation

An alternative method for demonstrating the "execution" of a language or calculus based on an operational semantics is to *animate* it by encoding the transition rules directly. In [18], Pieter H. Hartel describes `latos`, a tool that can automate the general creation of such animators.

This method would provide an easier way to reason about the correspondence between the π-calculus and implementation—the semantic gap between an animating interpreter and the π-calculus is much smaller than

[2]Chosen for clarity of expression for these simple examples.

between a bytecode interpreter (such as the one implemented in this project) and the π-calculus.

An animating interpreter with a proven correspondence to the π-calculus could then be used to automate testing of a bytecode compiler and interpreter (if both used the same internal representation).

4.4.2 Programming Features

Because the language designed is so close to the π-calculus, it lacks many features included in mainstream programming languages, such as additional concrete data types or a way of defining modules of reusable code. Incorporating these features into the language would be possible (perhaps requiring some modifications to the virtual machine architecture)—the resulting language would then be closer to occam-pi or pict. A simple improvement that would make a big difference to the ease of programming would be to implement the *polyadic* π-calculus — allowing processes to send and receive arbitrary sized tuples of elements.

Additionally, improving the efficiency of the implementation would be an essential goal if the language were to be made into a more usable, general-purpose tool.

4.5 Summary

In this chapter I gave demonstrations of the compiler and virtual machine debug output that allow the compilation and execution of a program to be inspected.

I also gave an example of how language behaviour can be compared to π-calculus semantics by extending the semantics of the π-calculus to allow for externally observable behaviour.

Additionally, I showed informally that the implementation correctly matches the π-calculus semantics, by giving a suite of programs meeting the criteria set out in the project proposal. These programs collectively provide strong circumstantial evidence that the implementation is correct with regard to the π-calculus semantics, and is capable of useful computation using its extended functionality.

Chapter 5

Conclusion

The final implementation of the project meets the success criteria specified in the initial project proposal, and the goals for functionality, work undertaken and time plans were met.

The characteristics and semantics of the π-calculus were examined in detail with respect to the design of a concrete programming language based on it. A language (PCL) was designed, with its syntax and semantics specified to match those of the π-calculus closely. Additionally, some extensions to facilitate general-purpose programming were added to PCL.

A virtual-machine instruction set and corresponding execution semantics were designed to support execution of concurrent programs—in particular, concurrent programs that share data via synchronisation between processes.

The language design and virtual machine were brought together by a compiler that transforms source files written in the language into linear sequences of bytecode instructions. Demonstrations of the internal functionality of this compiler were made available through the use of command-line arguments.

An extension to the π-calculus semantics that allows for the encoding of side-effects was constructed. This extension was used to demonstrate a behavioural correspondence between the π-calculus and equivalent programs written in PCL.

An interesting approach not taken by the project is to provide an animating interpreter for the operational semantics—future work might well include an implementation of such an interpreter to improve the standard of proof when verifying compliance with the π-calculus semantics. Other areas that could be examined might also involve improvements to the general-purpose usability of the language and runtime, such as adding more data types or improving performance.

Bibliography

[1] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[2] Joe Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[3] J. Tompson and K. Schlachter. An Introduction to the OpenCL Programming Model. 2012.

[4] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978.

[5] R. Robin Milner. *A Calculus of Communicating Systems.* Lecture notes in computer science. Springer-Verlag, Berlin, New York, 1980.

[6] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, September 1992.

[7] Robin Milner. Functions as processes. Research Report RR-1154, INRIA, 1990.

[8] Richard Banach and Franck van Breugel. Mobility and Modularity: expressing π-calculus in CCS.

[9] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, CCS '97, pages 36–47, New York, NY, USA, 1997. ACM.

[10] Andrew Phillips and Luca Cardelli. Efficient, Correct Simulation of Biological Processes in the Stochastic Pi-calculus. In *Proceedings of the 2007 International Conference on Computational Methods in Systems Biology*, CMSB'07, pages 184–199, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Tõnis Pool. Comparison of Erlang Runtime System and Java Virtual Machine. `http://ds.cs.ut.ee/courses/course-files/To303nis%20Pool%20.pdf`, May 2015. Accessed: 2016-04-25.

[12] M. Elizabeth C. Hull. Occam—A Programming Language for Multiprocessor Systems. *Comput. Lang.*, 12(1):27–37, January 1987.

[13] Peter H. Welch and Frederick R.M. Barnes. Communicating Mobile Processes. In *Communicating Sequential Processes: The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Berlin Heidelberg, 2005.

[14] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language based on the π-Calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 1997.

[15] The Scala Programming Language. `http://www.scala-lang.org`. Accessed: 2016-02-02.

[16] A Postfunctional Language. `http://www.scala-lang.org/old/node/4960`. Accessed: 2016-02-02.

[17] Pattern (Java Platform SE 7). `https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`. Accessed: 2016-04-26.

[18] Pieter H. Hartel. LATOS—A Lightweight Animation Tool for Operational Semantics. Technical report, 1997.

Appendix A

Language Specification

A.1 Lexical Format

A.1.1 Text Encoding

Programs in the language are composed only of ASCII-encoded text. Any program text lying outside of the ASCII space will be considered invalid and result in a compilation error.

A.1.2 Tokens

All tokens are defined using the Java SE 7 regular expression syntax [17].

Token	Definition
`VAR[name]`	`^([A-Z][a-z]*)(.*)`
`CHAN[name]`	`^@((?:[a-z]|_)+)(.*)`
`INT[n]`	`^(-?[0-9]+)(.*)`
`OPERATOR[op]`	`^(\+|-|\*|\/)(.*)`
`IN`	`^in(.*)`
`OUT`	`^out(.*)`
`LET`	`^let(.*)`
`PARALLEL`	`^\\|(.*)`
`SEQUENCE`	`^\.(.*)`
`END`	`^end(.*)`
`REPLICATE`	`^!(.*)`
`OPENB`	`^\((.*)`
`CLOSEB`	`^\)(.*)`
`OPENSB`	`^\[(.*)`

`CLOSESB`	`^\](.*)`
`OPENCB`	`^\{(.*)`
`CLOSECB`	`^\}(.*)`
`EQUALS`	`^ = (.*)`
`FRESH`	`^fresh(.*)`

A.1.3 External Channels

A program's source may be preceded by zero or more external channel markers. These markers indicate to the virtual machine which channels link to external code, but are not considered to be a part of the program text (i.e. the data given to the lexer to produce a list of tokens).

External channel markers will be preprocessed before lexing. The format of each individual marker as a regular expression is (where `CHAN[name]` is the regular expression for a channel identifier given in A.1.2):

`^external CHAN[name]$`

Each of these markers must occupy a distinct line in the source file, with all markers coming before the beginning of the program itself. There may be blank lines between markers, and between the final marker and the program.

A.2 Grammar

In this section I describe the grammar of the language as used in the parser and abstract syntax tree. The grammar is given in Backus-Naur form, with terminal symbols being styled as "`terminal`" (including all punctuation marks), and nonterminals as "*Nonterminal*". The terminal symbols `var` and `channel` have a string associated with them (respresenting the variable or channel name, respectively), and the terminal `int` has an integer associated similarly.

For clarity, the text corresponding to a token (terminal symbol) in this grammar is given instead of the name of the token. Given formally, an example production might be:

`OUT` *Name* `OPENB` *Expression* `CLOSEB` *Process′*

A.2.1 Definition

$$\begin{array}{lll}
\mathit{Start} & ::= & \mathit{Process} \\
 & & \\
\mathit{Name} & ::= & \texttt{var} \\
 & | & \texttt{channel} \\
 & & \\
\mathit{AddOp} & ::= & \texttt{+} \\
 & | & \texttt{-} \\
 & & \\
\mathit{MulOp} & ::= & \texttt{*} \\
 & | & \texttt{/} \\
 & & \\
\mathit{Factor} & ::= & \texttt{var} \\
 & | & \texttt{int} \\
 & | & \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
 & & \\
\mathit{Term} & ::= & \mathit{Factor}\ \mathit{Term}' \\
 & & \\
\mathit{Term}' & ::= & \mathit{MulOp}\ \mathit{Factor}\ \mathit{Term}' \\
 & | & \varepsilon \\
 & & \\
\mathit{Expression} & ::= & \mathit{Term}\ \mathit{Expression}' \\
 & | & \texttt{channel} \\
 & & \\
\mathit{Expression}' & ::= & \mathit{AddOp}\ \mathit{Term}\ \mathit{Expression}' \\
 & | & \varepsilon \\
 & & \\
\mathit{Process} & ::= & \texttt{out}\ \mathit{Name}\ \texttt{(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Process}' \\
 & | & \texttt{in}\ \mathit{Name}\ \texttt{(}\ \texttt{var}\ \texttt{)}\ \mathit{Process}' \\
 & | & \texttt{(}\ \mathit{Process} \mid \mathit{Process}\ \texttt{)} \\
 & | & \texttt{!}\ \texttt{(}\ \mathit{Process}\ \texttt{)} \\
 & | & \texttt{[}\ \mathit{Expression}\ \texttt{=}\ \mathit{Expression}\ \texttt{]}\ \{\ \mathit{Process}\ \}\ \mathit{Process}' \\
 & | & \texttt{let}\ \texttt{var}\ \texttt{=}\ \mathit{Expression}\ \{\ \mathit{Process}\ \}\ \mathit{Process}' \\
 & | & \texttt{fresh}\ \texttt{var}\ \{\ \mathit{Process}\ \}\ \mathit{Process}' \\
 & | & \texttt{end} \\
 & & \\
\mathit{Process}' & ::= & \texttt{.}\ \mathit{Process}\ \mathit{Process}' \\
 & | & \varepsilon
\end{array}$$

Appendix B

Virtual Machine Specification

B.1 Instruction Set

B.1.1 Arithmetic

`PUSH[v]`: Push the value `v` onto the arithmetic stack.

`ADD`: Stack addition operator.

`SUB`: Stack subtraction operator.

`MUL`: Stack multiplication operator.

`DIV`: Stack truncated division operator.

B.1.2 Store and Arithmetic

`LOAD[n]`: Push the value of `n` in the environment onto the stack.

`STORE-INT[n]`: Pop a value from the stack and store it in the variable `n`.

`STORE-CHANNEL[v,c]`: Store the channel `c` in the variable `v`.

`COPY[t,f]`: Copy the contents of `t` to `f`.

`DELETE[v]`: Delete `v` from the environment.

B.1.3 Control Flow

`VAR-COMPARE[v1,v2]`: Push 1 to the stack if the contents of `v1` and `v2` are equal.

`CHAN-COMPARE[v,c]`: Push 1 to the stack if `v` contains the channel `c`.

`LABEL[name]`: Mark a label with `name`.

`JUMP-ZERO[name]`: Pop from the stack; if the result is 0 jump to `name`.

`JUMP-NON-ZERO[name]`: Pop from the stack; if the result is non-zero jump to `name`.

`SPAWN[name]`: Spawn a concurrent interpreter at `name`.

`END`: Stop execution.

B.1.4 Synchronisation

`RECEIVE-DIRECT[chan,name]`: Receive a value on `chan` and store it in `name`.

`RECEIVE-INDIRECT[var,name]`: Receive a value on the channel stored in `var` and store it in `name`.

`SEND-CHAN-DIRECT[chan,data]`: Send the channel `data` on `chan`.

`SEND-CHAN-DIRECT[var,data]`: Send the channel `data` on the channel stored in `var`.

`SEND-INT-DIRECT[chan]`: Pop a value from the stack and send it on `chan`.

`SEND-INT-DIRECT[var]`: Pop a value from the top of the stack and send it on the channel stored in `var`.

`SEND-VAR-DIRECT[chan,data-var]`: Send the contents of the variable `data-var` on `chan`.

`SEND-VAR-DIRECT[var,data-var]`: Send the contents of the variable `data-var` on the channel stored in `var`.

Appendix C

Using `pcl`

The implementation of the language as described in this dissertation can be deployed as a standalone `.jar` file. The usage of this executable is described here.

C.1 External Libraries

External code in the format described in Section 3.3 should be compiled to `.class` files using `scalac` and placed in the directory:

```
/usr/local/lib/picl
```

The basename of these files determines the channel name by which they are loaded at runtime.

C.2 Command Line Arguments

When running the executable, the argument format is:

```
pcl file [--dump] [--trace]
```

The individual arguments are defined as:

`file` The program file to compile and run. Non-optional.

`--dump` Print all intermediate representations to standard error. Optional.

`--trace` Print each instruction and a thread ID when it is executed. Optional.

Appendix D

π-Calculus Definition

D.1 Syntax

The full syntax of the π-calculus is given by the grammar:

$$\begin{array}{rcl} P & ::= & x(y).P \\ & | & \overline{x}\langle y\rangle.P \\ & | & P|P \\ & | & (\nu\, x)P \\ & | & !P \\ & | & \boldsymbol{nil} \end{array}$$

In this grammar, P is a π-calculus process, and x, y are names from a countable set of names X.

D.2 Name Scope

The free and bound names of a process are given by the functions $fn(p)$ and $bn(p)$ respectively. In the extended semantics that includes external channels, all names in the set E are taken to be free names.

Free names:

$$\begin{aligned}
fn(\boldsymbol{nil}) &= \emptyset \\
fn(\overline{x}\langle y\rangle.P) &= \{x, y\} \cup fn(P) \\
fn(x(y).P) &= a \cup (fn(P) - \{y\}) \\
fn(P \mid Q) &= fn(P) \cup fn(Q) \\
fn((\nu\, x)P) &= fn(P) - \{x\} \\
fn(!P) &= fn(P)
\end{aligned}$$

Bound names:

$$\begin{aligned}
bn(\boldsymbol{nil}) &= \emptyset \\
bn(\overline{x}\langle y\rangle.P) &= bn(P) \\
bn(x(y).P) &= bn(P) \cup \{x\} \\
bn(P \mid Q) &= bn(P) \cup bn(Q) \\
bn((\nu\, x)P) &= bn(P) \cup \{x\} \\
bn(!P) &= bn(P)
\end{aligned}$$

D.3 Structural Congruence

The structural congruence relation is defined by a number of rules. α-equivalence ($=_\alpha$) relates two processes if they are structurally equivalent, but differ in terms of their bound names.

$$\begin{aligned}
P &\equiv Q && \text{if } P =_\alpha Q \\
P \mid Q &\equiv Q \mid P \\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R) \\
P \mid \boldsymbol{nil} &\equiv P \\
(\nu\, x)(\nu\, y)P &\equiv (\nu\, y)(\nu\, x)P \\
(\nu\, x)\boldsymbol{nil} &\equiv \boldsymbol{nil} \\
!P &\equiv P \mid !P \\
(\nu\, x)(P \mid Q) &\equiv (\nu\, x)P \mid Q && \text{if } x \notin fn(Q)
\end{aligned}$$

D.4 Transition Relation

The transition relation describes how processes evolve through synchronisation with each other. It is defined by a number of rules and relies on the definition of structural congruence given above. $Q[z/y]$ is the process Q with every bound instance of y replaced with z.

$$\begin{aligned}
\overline{x}\langle z\rangle.P \mid x(y).Q &\rightarrow P \mid Q[z/y] \\
P \rightarrow Q &\implies P \mid R \rightarrow Q \mid R \\
P \rightarrow Q &\implies (\nu\, x)P \rightarrow (\nu\, x)Q \\
P \rightarrow Q \land P \equiv P' \land Q \equiv Q' &\implies P' \rightarrow Q'
\end{aligned}$$

Appendix E

Execution Logs

E.1 Basic Equivalences

Nondeterminism

Definition:

$$\begin{aligned} par &\triangleq \overline{stdio}\langle a\rangle \mid \overline{stdio}\langle b\rangle \\ E_{par} &\triangleq \{stdio\} \end{aligned}$$

Reduction sequences:

$$\begin{aligned} par &\xrightarrow{\overline{stdio}\langle a\rangle} \text{nil} \mid \overline{stdio}\langle b\rangle \\ &\xrightarrow{\overline{stdio}\langle b\rangle} \text{nil} \mid \text{nil} \end{aligned}$$

$$\begin{aligned} par &\xrightarrow{\overline{stdio}\langle b\rangle} \overline{stdio}\langle a\rangle \mid \text{nil} \\ &\xrightarrow{\overline{stdio}\langle a\rangle} \text{nil} \mid \text{nil} \end{aligned}$$

```
external @stdio

(
    out @stdio(@a)
|
    out @stdio(@b)
)
```

Listing E.1: `par.pcl`

```
external @stdio

(
    end
|
    out @stdio(@b)
)
```

Listing E.2: `par-1a.pcl`, one step reduction of `par.pcl`, first path

```
external @stdio

(
    out @stdio(@a)
|
    end
)
```

Listing E.3: `par-1b.pcl`, one step reduction of `par.pcl`, second path

```
$ pcl par.pcl --dump --trace
Tokens:
List(
    OpenBracket(),
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    ChannelName(a),
    CloseBracket(),
    Parallel(),
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    ChannelName(b),
    CloseBracket(),
    CloseBracket()
)

Parse Tree:
ProcessStart(
    ParallelProcess(
        OutProcess(
            ChannelName(stdio),
            ChannelExpression(
                ChannelName(a)
            ),
            EmptyProcessAux()
        ),
        OutProcess(
            ChannelName(stdio),
            ChannelExpression(
                ChannelName(b)
            ),
            EmptyProcessAux()
        ),
        EmptyProcessAux()
    )
)

Bytecode:
```

```
Spawn(a)
Spawn(b)
Jump(c)
Label(a)
SendChannelDirect(Channel(stdio),Channel(a))
End()
Label(b)
SendChannelDirect(Channel(stdio),Channel(b))
End()
Label(c)
End()

Thread 0: Spawn(a)
Thread 0: Spawn(b)
Thread 1: Label(a)
Thread 1: SendChannelDirect(
            Channel(stdio),Channel(a))
Thread 0: Jump(c)
Thread 0: Label(c)
Thread 2: Label(b)
Thread 0: End()
Thread 2: SendChannelDirect(
            Channel(stdio),Channel(b))
a
b
Thread 1: End()
Thread 2: End()
```

Listing E.4: `par.pcl` first execution path

```
$ pcl par.pcl --dump --trace
Tokens:
List(
    OpenBracket(),
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    ChannelName(a),
    CloseBracket(),
    Parallel(),
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    ChannelName(b),
    CloseBracket(),
    CloseBracket()
)

Parse Tree:
ProcessStart(
    ParallelProcess(
        OutProcess(
            ChannelName(stdio),
            ChannelExpression(
                ChannelName(a)
            ),
            EmptyProcessAux()
        ),
        OutProcess(
            ChannelName(stdio),
            ChannelExpression(
                ChannelName(b)
            ),
            EmptyProcessAux()
        ),
        EmptyProcessAux()
    )
)

Bytecode:
```

```
Spawn(a)
Spawn(b)
Jump(c)
Label(a)
SendChannelDirect(Channel(stdio),Channel(a))
End()
Label(b)
SendChannelDirect(Channel(stdio),Channel(b))
End()
Label(c)
End()

Thread 0: Spawn(a)
Thread 0: Spawn(b)
Thread 1: Label(a)
Thread 1: SendChannelDirect(
            Channel(stdio),Channel(a))
Thread 0: Jump(c)
Thread 0: Label(c)
Thread 2: Label(b)
Thread 0: End()
Thread 2: SendChannelDirect(
            Channel(stdio),Channel(b))
b
a
Thread 2: End()
Thread 1: End()
```

Listing E.5: par.pcl second execution path

```
$ pcl par-1a.pcl
b
```

Listing E.6: par-1a.pcl execution

```
$ pcl par-1b.pcl
a
```

Listing E.7: `par-1b.pcl` execution

Restriction

Definition:

$$\begin{aligned} nres &\triangleq \overline{stdio}\langle a\rangle \\ E_{nres} &\triangleq \{stdio\} \end{aligned}$$

$$\begin{aligned} res &\triangleq (\nu\, x)\ \overline{stdio}\langle a\rangle \\ E_{res} &\triangleq \{stdio\} \end{aligned}$$

Reduction sequences:

$$nres \xrightarrow{\overline{stdio}\langle a\rangle} \text{nil}$$

$$res \xrightarrow{\overline{stdio}\langle a\rangle} (\nu\, x)\ \text{nil}$$

```
external @stdio

out @stdio(@a)
```

Listing E.8: `nres.pcl`

```
external @stdio

fresh X {
    out @stdio(@a)
}
```

Listing E.9: `res.pcl`

```
$ pcl nres.pcl --dump --trace
Tokens:
List(
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    ChannelName(a),
    CloseBracket()
)

Parse Tree:
ProcessStart(
    OutProcess(
        ChannelName(stdio),
        ChannelExpression(
            ChannelName(a)
        ),
        EmptyProcessAux()
    )
)

Bytecode:
SendChannelDirect(Channel(stdio),Channel(a))
End()

Thread 0: SendChannelDirect(
            Channel(stdio),Channel(a))
a
Thread 0: End()
```

Listing E.10: `nres.pcl` execution

```
$ pcl res.pcl --dump --trace
Tokens:
List(
    Fresh(),
    VarName(X),
    OpenCurlyBracket(),
    Out(),
    ChannelName(stdio),
    OpenBracket(),
    ChannelName(a),
    CloseBracket(),
    CloseCurlyBracket()
)

Parse Tree:
ProcessStart(
    FreshProcess(
        VariableName(X),
        OutProcess(
            ChannelName(stdio),
            ChannelExpression(
                ChannelName(a)
            ),
            EmptyProcessAux()
        ),
        EmptyProcessAux()
    )
)

Bytecode:
Let(Variable(X),Left(Channel(~a)))
SendChannelDirect(Channel(stdio),Channel(a))
Delete(Variable(X))
End()

Thread 0: Let(Variable(X),Left(Channel(~a)))
Thread 0: SendChannelDirect(
            Channel(stdio),Channel(a))
a
Thread 0: Delete(Variable(X))
Thread 0: End()
```

Listing E.11: `res.pcl` execution

Congruence

Commutativity

Definition:

$$\begin{aligned} rev &\triangleq \overline{stdio}\langle b\rangle \mid \overline{stdio}\langle a\rangle \\ E_{rev} &\triangleq \{stdio\} \end{aligned}$$

Reduction Sequences:

$$\begin{aligned} rev &\xrightarrow{\overline{stdio}\langle b\rangle} \text{nil} \mid \overline{stdio}\langle a\rangle \\ &\xrightarrow{\overline{stdio}\langle a\rangle} \text{nil} \mid \text{nil} \end{aligned}$$

$$\begin{aligned} rev &\xrightarrow{\overline{stdio}\langle a\rangle} \overline{stdio}\langle b\rangle \mid \text{nil} \\ &\xrightarrow{\overline{stdio}\langle b\rangle} \text{nil} \mid \text{nil} \end{aligned}$$

```
external @stdio

(
    out @stdio(@b)
|
    out @stdio(@a)
)
```

Listing E.12: `reverse.pcl`

```
$ pcl reverse.pcl
a
b
$ pcl reverse.pcl
b
a
```

Listing E.13: `reverse.pcl` execution

Associativity

Definition:

$$\begin{aligned} assocl &\triangleq (\overline{stdio}\langle a\rangle \mid \overline{stdio}\langle b\rangle) \mid \overline{stdio}\langle c\rangle \\ E_{assocl} &\triangleq \{stdio\} \end{aligned}$$

$$\begin{aligned} assocr &\triangleq \overline{stdio}\langle a\rangle \mid (\overline{stdio}\langle b\rangle \mid \overline{stdio}\langle c\rangle) \\ E_{assocr} &\triangleq \{stdio\} \end{aligned}$$

```
external @stdio

(
    (
        out @stdio(@a)
    |
        out @stdio(@b)
    )
|
    out @stdio(@c)
)
```

Listing E.14: `assocl.pcl`

```
external @stdio

(
    out @stdio(@a)
|
    (
        out @stdio(@b)
    |
        out @stdio(@c)
    )
)
```

Listing E.15: `assocr.pcl`

```
$ pcl assocl.pcl
c
a
b
$ pcl assocl.pcl
c
b
a
$ pcl assocl.pcl
a
c
b
$ pcl assocl.pcl
b
c
a
$ pcl assocl.pcl
a
b
c
$ pcl assocl.pcl
b
a
c
```

Listing E.16: `assocl.pcl` execution

```
$ pcl assocr.pcl
a
c
b
$ pcl assocr.pcl
c
a
b
$ pcl assocr.pcl
c
b
a
$ pcl assocr.pcl
b
c
a
$ pcl assocr.pcl
a
b
c
$ pcl assocr.pcl
b
a
c
```

Listing E.17: `assocr.pcl` execution

Restriction Reordering

Definition:

$$xy \triangleq (\nu\, x)(\nu\, y)\ \overline{stdio}\langle a\rangle$$

$$yx \triangleq (\nu\, y)(\nu\, x)\ \overline{stdio}\langle a\rangle$$

```
external @stdio

fresh X {
    fresh Y {
        out @stdio(@a)
    }
}
```

Listing E.18: `xy.pcl`

```
external @stdio

fresh Y {
    fresh X {
        out @stdio(@a)
    }
}
```

Listing E.19: `yx.pcl`

```
$ pcl xy.pcl
a
```

Listing E.20: `xy.pcl` execution

```
$ pcl yx.pcl
a
```

Listing E.21: `yx.pcl` execution

Nil Restriction

Definition:

$$resn \triangleq (\nu\, x)\ \mathrm{nil}$$

```
fresh X {
    end
}
```

Listing E.22: `resn.pcl`

```
$ pcl resn.pcl
$
```

Listing E.23: `resn.pcl` execution

Scope Extension

Definition:

$$\begin{aligned} scope &\triangleq ((\nu\, x)\ \overline{stdio}\langle a\rangle) \mid \overline{stdio}\langle b\rangle \\ E_{scope} &\triangleq \{stdio\} \end{aligned}$$

$$\begin{aligned} scopeext &\triangleq (\nu\, x)\ (\overline{stdio}\langle a\rangle \mid \overline{stdio}\langle b\rangle) \\ E_{scopeext} &\triangleq \{stdio\} \end{aligned}$$

```
external @stdio

(
    fresh X {
        out @stdio(@a)
    }
|
    out @stdio(@b)
)
```

Listing E.24: scope.pcl

```
external @stdio

fresh X {
    (
        out @stdio(@a)
    |
        out @stdio(@b)
    )
}
```

Listing E.25: scopeext.pcl

```
$ pcl scope.pcl
b
a
$ pcl scope.pcl
a
b
```

Listing E.26: scope.pcl execution

```
$ pcl scopeext.pcl
a
b
$ pcl scopeext.pcl
b
a
```

Listing E.27: `scopeext.pcl` execution

E.2 Algorithms

This section contains code and execution logs for several numerical algorithms implemented in the language. Equivalent python code is also given for testing purposes.

Sign

```
external @stdio

fresh SignN {
fresh SignC {
    (
        in SignN(Num).
        in SignC(Cont).
        [Num = 0] {
            out Cont(0).end
        }.
        [Num = 1] {
            out Cont(1).end
        }.
        let Div = (1 - 2*Num) / (1 - Num) {
            out Cont(2*Div - 3)
        }
    |
        in @stdio(N).
        out SignN(N).
        out SignC(@stdio)
    )
}}
```

Listing E.28: `sign.pcl`

```python
n = int(input(">␣"))
if n == 0:
    print(0)
elif n > 0:
    print(1)
else:
    print(-1)
```

Listing E.29: `sign-ref.py`

```
$ pcl sign.pcl
> 0
0
$ python sign-ref.py
> 0
0
$ pcl sign.pcl
> 768
1
$ python sign-ref.py
> 768
1
$ pcl sign.pcl
> -3429
-1
$ python sign-ref.py
> -3429
-1
$ pcl sign.pcl
> 12
1
$ python sign-ref.py
> 12
1
```

Listing E.30: `sign.pcl` testing

Modulus

```
external @stdio

fresh Input {
    (
        in Input(Num).
        in Input(Mod).
        in Input(Cont).
        out Cont(Num - ((Num / Mod) * Mod))
    |
        in @stdio(N).
        in @stdio(M).
        out Input(N).
        out Input(M).
        out Input(@stdio)
    )
}
```

Listing E.31: `mod.pcl`

```
n = int(input("> "))
m = int(input("> "))
print(n % m)
```

Listing E.32: `mod-ref.py`

```
$ pcl mod.pcl
> 10
> 3
1
$ python mod-ref.py
> 10
> 3
1
$ pcl mod.pcl
> 21
> 9
3
$ python mod-ref.py
> 21
> 9
3
$ pcl mod.pcl
> 4
> 10
4
$ python mod-ref.py
> 4
> 10
4
$ pcl mod.pcl
> 0
> 3
0
$ python mod-ref.py
> 0
> 3
0
```

Listing E.33: `mod.pcl` testing

Exponentiation

```
external @stdio

fresh ExpX {
fresh ExpY {
fresh ExpA {
fresh ExpC {
    (
        !(
            in ExpX(X).
            in ExpY(Y).
            in ExpA(A).
            in ExpC(Cont).
            [Y = 0] {
                out Cont(1).end
            }.
            [Y = 1] {
                out Cont(A * X).end
            }.
            out ExpX(X).
            out ExpY(Y - 1).
            out ExpA(A * X).
            out ExpC(Cont)
        )
    |
        in @stdio(A).
        in @stdio(B).
        out ExpX(A).
        out ExpY(B).
        out ExpA(1).
        out ExpC(@stdio)
    )
}}}}
```

Listing E.34: `exp.pcl`

```python
x = int(input("> "))
y = int(input("> "))
print(x**y)
```

Listing E.35: `exp-ref.py`

```
$ pcl exp.pcl
> 2
> 3
8
$ python exp-ref.py
> 2
> 3
8
$ pcl exp.pcl
> 5
> 1
5
$ python exp-ref.py
> 5
> 1
5
$ pcl exp.pcl
> 6
> 0
1
$ python exp-ref.py
> 6
> 0
1
```

Listing E.36: `exp.pcl` testing

GCD

```
external @stdio
fresh ModParamN {
fresh ModParamM {
fresh ModParamC {
fresh GCDParamA {
fresh GCDParamB {
fresh GCDParamC {
    ((
        !(
            in ModParamN(Num).
            in ModParamM(Mod).
            in ModParamC(Cont).
            out Cont(Num - ((Num / Mod) * Mod)))
    |
        !(
            in GCDParamA(A).
            in GCDParamB(B).
            in GCDParamC(Cont).
            [A = 0] {
                out Cont(B).end
            }.
            [B = 0] {
                out Cont(A).end
            }.
            out GCDParamA(B).
            out ModParamN(A).
            out ModParamM(B).
            out ModParamC(GCDParamB).
            out GCDParamC(Cont))
    )|
        in @stdio(X).
        in @stdio(Y).
        out GCDParamA(X).
        out GCDParamB(Y).
        out GCDParamC(@stdio)
)}}}}}}
```

Listing E.37: gcd.pcl

```python
from fractions import gcd
a = int(input("> "))
b = int(input("> "))
print(gcd(a,b))
```

Listing E.38: `gcd-ref.py`

```
$ pcl gcd.pcl
> 100
> 40
20
$ python gcd-ref.py
> 100
> 40
20
$ pcl gcd.pcl
> 11
> 19
1
$ python gcd-ref.py
> 11
> 19
1
$ pcl gcd.pcl
> 56
> 56
56
$ python gcd-ref.py
> 56
> 56
56
```

Listing E.39: `gcd.pcl` testing

Fibonacci

```
external @stdio

fresh FibParamN {
fresh FibParamA {
fresh FibParamB {
fresh FibParamC {
    (
        !(
            in FibParamN(N).
            in FibParamA(A).
            in FibParamB(B).
            in FibParamC(Cont).
            [N = 0] {
                out Cont(A).end
            }.
            out FibParamN(N - 1).
            out FibParamA(A + B).
            out FibParamB(A).
            out FibParamC(Cont)
        )
    |
        in @stdio(X).
        out FibParamN(X).
        out FibParamA(0).
        out FibParamB(1).
        out FibParamC(@stdio)
    )
}}}}
```

Listing E.40: `fib.pcl`

```python
def fib(n,a,b):
    if n == 0:
        return a
    else:
        return fib(n-1, a+b, a)

n = int(input(">␣"))
print(fib(n,0,1))
```

Listing E.41: `fib-ref.py`

```
$ pcl fib.pcl
> 1
1
$ python fib-ref.py
> 1
1
$ pcl fib.pcl
> 4
3
$ python fib-ref.py
> 4
3
$ pcl fib.pcl
> 15
610
$ python fib-ref.py
> 15
610
```

Listing E.42: `fib.pcl` testing

E.3 Encoding of The Church Numerals

```
fresh Zero {
fresh Succ {
fresh True {
fresh False {
fresh LazyFalse {
fresh IsZero {
fresh IfElse {
    ((((((
        !(
            in Zero(Args).
            in Args(Output).
            in Args(F).
            in Args(X).
            out Output(X)
        )
    |
        !(
            in Succ(Args).
            in Args(Output).
            in Args(N).
            in Args(F).
            in Args(X).
            fresh NArgs {
            fresh FArgs {
                out N(NArgs).
                out F(FArgs).

                out NArgs(FArgs).
                out FArgs(Output).

                out NArgs(F).
                out NArgs(X)
            }}
        )
   )|
        !(
            in True(Args).
```

```
                in Args(Output).
                in Args(X).
                in Args(Y).
                out Output(X)
            )
    )|
            !(
                in False(Args).
                in Args(Output).
                in Args(X).
                in Args(Y).
                out Output(Y)
            )
    )|
            !(
                in LazyFalse(Args).
                in Args(Output).
                in Args(X).
                out Output(False)
            )
    )|
            !(
                in IsZero(Args).
                in Args(Output).
                in Args(N).
                fresh NArgs {
                    out N(NArgs).
                    out NArgs(Output).
                    out NArgs(LazyFalse).
                    out NArgs(True)
                }
            )
    )|
            !(
                in IfElse(Args).
                in Args(Output).
                in Args(P).
                in Args(A).
                in Args(B).
                fresh PArgs {
                    out P(PArgs).
```

```
                        out PArgs(Output).
                        out PArgs(A).
                        out PArgs(B)
                    }
              )
       )
}}}}}}}
```

Listing E.43: `church.pcl`

Proforma

Name	Bruce Collie
CRSID	`bsc28`
College	Trinity Hall
Title	An Implementation of the π-Calculus
Date	October 21, 2015
Supervisor	Prof. Alan Mycroft

1 Introduction & Description of Work

In theoretical Computer Science there exists the idea of a 'calculus' — a small language that exists to distill or demonstrate a particular programming paradigm in as minimal a way as possible. The most famous of these is the λ-calculus which demonstrates functional programming, and another example is the object calculus (which relates to object oriented programming).

In this project I will consider the π-calculus - a minimal model for concurrent programming based on name passing along named channels [1]. While the π-calculus has been demonstrated to be Turing-complete (by showing that the λ-calculus can be embedded within the π-calculus), it may be useful to have a concrete implementation of the language that can be used to demonstrate its behaviour.

To this end the project I propose is to implement a variant of the synchronous π-calculus by means of an end-to-end compiler. This compiler will take a concrete textual representation of the language and transform it into the executable bytecode format for a virtual machine I will design from scratch. The language variant proposed will include support for concrete data types (e.g. 64-bit integers) in order to make the model of computation less abstract for users of the language.

Implementing the language in this way will provide an interesting discussion on how nondeterminism can be dealt with in a practical implementation.

2 Starting Point

This project is not based on any previous Part II project that I am aware of, and at the time of writing this proposal I have not written any code that I would expect to use in the project for any reason.

3 Substance & Structure

The aim of the project is to implement a variant of the π-calculus as a compiler and a virtual machine as described above. This will involve the implementation of the following components:

1. A simple concrete syntax for the variant language must be designed. This syntax will be an as direct as possible textual representation of the abstract syntax of the π-calculus. Previous similar efforts in this area such as Pict [2] and OccamPi [3] will be examined with regard to their treatment of syntax. As well as this, I will also decide which concrete data types will be included as language primitives. This stage of the implementation will require careful thought and evaluation of the usability of the language design.

2. The instruction set and implementation strategy for the virtual machine must be decided. Different options at this stage will have relative advantages and disadvantages when used as a target for the compiler, particularly with regards to how nondeterminism is implemented. Initially programs in the bytecode format will be able to perform simple read / print IO to demonstrate their execution. Extension work could involve a visualisation of the execution by a modified virtual machine. Nondeterminism in the language will be provided by the VM calling into a threading library when parallel composition occurs. Extension work could allow for more advanced simulation or animation of the program's execution (e.g. by backtracking).

3. I will write a collection of small programs in my variant language that demonstrate the functionality and features of the π-calculus, as well as some larger programs that perform 'real world' computations:

    - I will write a program for each syntactic element of the calculus (e.g. sending a message on a channel, waiting for and receiving a message, parallel composition etc.). These programs can be evaluated by comparison with the semantics of the abstract π-calculus to demonstrate that my variant language does in fact implement the calculus correctly. As a representative example, one program may demonstrate the parallel composition of two processes evolving when one sends a message that the other is waiting to receive. To demonstrate success here I will present mathematical proofs of the execution of these small programs using the reduction semantics of the calculus.
    - For the larger programs, I will select several well–known algorithms (e.g. computing fibonacci numbers or calculating the GCD of two numbers) and implement them in my variant language to demonstrate that the language as a whole is capable of more concrete work. I will need to present a test set of data for these programs that can be passed to a reference implementation, then compared to my program's output for correctness.
    - Milner [1] demonstrates that it is possible to encode certain data structures (e.g. a list structure or the Peano natural numbers) in the π-calculus. I will demonstrate programs equivalent to the encoding given by Milner (i.e. as a direct syntax translation), then show that

their execution is correct according to the semantics of the language. This will require me to present a mathematical proof of the evaluation of such an encoding using the reduction semantics of the calculus.

- Possible extension work could involve modelling a security protocol (or similar), though this would be more difficult to test and evaluate.

4. A lexer and parser for the concrete syntax will be implemented. This will involve formally analysing the grammar and structure of the language that has been designed. A suitable formal parsing algorithm will be evaluated and decided upon based on the requirements of the grammar.

5. The parse tree that has been outputted from the parser will be converted into an intermediate representation using code generation techniques and algorithms (e.g. register allocation and basic block flow analysis). Some transformations and optimisations can be applied to the resultant flow graph.

6. Finally, I will generate target code from my intermediate representation. Extension work may be possible at this stage (e.g. to generate 'real world' code such as ARM assembler).

4 Success Criteria

For the project to be judged as a success I have the following criteria:

- To be able to demonstrate the compilation of a suite of programs (as described above) that demonstrate the capabilities of my variant language.

- To be able to demonstrate the execution of such programs by the Virtual Machine that I implement.

- My implementation will be judged as a success if these programs meet their success criteria as described above (i.e. being compliant with the abstract semantics or performing a computation correctly).

5 Plan of Work & Timetable

Weeks 1–2 Michaelmas Preparatory work on project including initial discussions with supervisor and writing of project proposal, as well as initial individual research into project topics. **Milestones**: Accepted project proposal.

Weeks 3–4 Michaelmas Formal design and evaluation of the languages to be used in the project. Begin to design skeleton layout for dissertation to facilitate writing at later stages. **Milestones**: Grammar for the variant language being implemented (to facilitate lexer and parser design), and a specification / semantics for the target virtual machine code.

Weeks 5–6 Michaelmas Implementation of the virtual machine according to the specifications designed in the previous section. **Milestones**: A functional program that can execute hand-written examples of the virtual machine code.

Weeks 7–8 Michaelmas Implementation of the compiler front end (i.e. lexer and parser). **Milestones**: A program that can transform a textual representation of a program compliant with the grammar designed previously and convert it to an internal parse tree.

Christmas Vacation Work on implementing a code generation phase. Start to flesh out sections of dissertation that have already been worked on. **Milestones**: A working (if not yet complete) compiler for the variant language that connects the front end and code generation phases.

Weeks 1–2 Lent Evaluate work done so far on the project and identify areas of implementation that could be improved, fixed or extended. **Milestones**: Progress report written and submitted as per the pink book. List of tasks to work on with the implementation.

Weeks 3–4 Lent Work on implementation areas identified as needing work done in previous section. **Milestones**: Comparision of current work with list identified before demonstrating improvement. If time has permitted then evaluate extension work done.

Weeks 5–6 Lent Fully evaluate success of implementation compared to success criteria from this proposal. **Milestones**: Draft of evaluation section of the dissertation. Demonstrations of project working.

Weeks 7–8 Lent Work on drafting remaining sections of dissertation as time permits. **Milestones**: Significant portion of dissertation completed in draft form (if not all).

Easter Vacation Complete draft of dissertation. **Milestones**: Draft dissertation sent to supervisor for feedback before returning for Easter term.

Weeks 1–2 Easter Minor tweaks to dissertation. **Milestones**: Dissertation tweaked as needed and almost ready for submission.

Weeks 3–4 Easter Fix any last-minute issues and submit dissertation and code. **Milestones**: Dissertation submitted as per the pink book.

6 Resource Declaration

The project will be implemented in Scala, a functional / multi-paradigm language that runs on the JVM. I will need to install the Scala compiler and libraries on the machines I use for development. This can be done locally with no root permissions so is compatible with the MCS.

I intend to carry out development work primarily on my own laptop for convenience. Its specifications are:

- 2013 Macbook Air 13"
- 1.7 GHz Intel Core i7 CPU
- 8 GB RAM
- 256 GB SSD

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. In the event of such failure, I will be able to use the MCS computing facilities to complete the project with minimal disruption.

In order to protect myself against data loss or corruption, I have a multi-location backup strategy in place. All my work will be pushed to a private BitBucket git repository as I complete parts of it, so that revision history and the full corpus of work as it stands is available to clone if necessary. As well as this, my laptop will be backed up automatically every night to an external hard disk drive using Apple's Time Machine facility. Finally, I will set up a weekly snapshot of all my work that will be remotely copied to my MCS file space.

References

[1] Robin Milner. *Communicating and Mobile Systems: The -calculus.* Cambridge University Press, 1999.

[2] Benjamin C Pierce and David N Turner. "Pict: A Programming Language Based on the Pi-Calculus". In: *Indiana University CSCI Technical Report #476* (1997).

[3] Peter H. Welch and Frederick R. M. Barnes. "Communicating mobile processes: introducing occam-pi". In: *In 25 Years of CSP.* Springer Verlag, 2005, pp. 175–210.