Imperial College
London

3$^{\text{RD}}$ Year Computing

Individual Project

`phi>`
The Applied π-calculus Interpreter

Author:
William de Renzy-Martin

Supervisor:
Sergio Maffeis

June 17, 2014

Abstract

Typically, when defining a security protocol, we do not concern ourselves with how it might be implemented. The languages we use to describe protocols are too abstract to execute, and the languages we use to implement protocols are too convoluted to reason about.

Here we provide our implementation of the Applied π-calculus, with which one can combine definition and reasoning with implementation.

Acknowledgements

I would like to thank my supervisor, Sergio Maffeis for introducing me to the concepts presented in this paper, and for taking the time to guide me through the process.

I would also like to thank the Imperial College Department of Computing for helping me come to the conclusion that I should leave the world of computers behind me and pursue my dreams of becoming an actor.

Contents

1 Introduction **6**

1.1 Objective 6

1.2 Approach 7

2 Background **8**

2.1 Process Calculi 8

2.1.1 Communication 8

2.1.2 Sequential Composition 8

2.1.3 Parallel Composition 8

2.1.4 Reduction Semantics 9

2.1.5 Hiding 9

2.1.6 Recursion and Replication 9

2.1.7 The Null Process 9

2.2 π-calculus and the Calculus of Communicating systems . . . 9

2.3 Interpretation 9

2.4 The applied π-calculus 10

2.4.1 Syntax 10

2.4.2 Simplified Syntax 10

2.4.3 Starting Restrictions 11

2.5 Haskell 11

2.5.1 Data Type 12

2.5.2 Parsec 13

2.5.3 Sockets and the Network module 15

3 Package Overview **17**

3.1 TypDefs 17

3.2 Parser 17

3.3 Channel 17

3.4 Primitives 17

3.5 PatternMatching 17

3.6 PiCalculus 18

4 Types and Parsing **19**

4.1 Types 19

4.1.1 Terms 19

4.1.2 Processes 21

4.1.3 Channels 22

4.1.4 Values 23

4.1.5 Environment 24

4.1.6 Error Handling 24

4.2 Parser 25

4.2.1 parseTerm 26

4.2.2 parseTerm 27

5 Channels and Processes 30

5.1 Channels 30

5.1.1 Serialising Channels 31

5.1.2 Standard Channels 31

5.1.3 Dummy Channels 32

5.1.4 Socket Channels 32

5.1.5 Other Handle Types 35

5.2 Processes 35

6 Primitives and Term Manipulation 36

6.1 Definitions and Explanations 36

6.1.1 List, Pair and Number Functions 36

6.1.2 HTTP Data Functions 37

6.1.3 Cryptographic Functions 38

7 Pattern Matching 40

7.1 Implementation 41

8 Evaluation Strategy and Bringing it All Together 44

8.1 Evaluation Strategy Influence 44

8.2 Main module 44

8.3 User Input and Output 44

8.3.1 Running the program 44

8.4 Evaluation of Data Structures 46

8.4.1 Environment 47

8.4.2 eval 47

8.4.3 evalTerm 50

8.4.4 Function Application 52

8.4.5 Pattern Matching 53

8.5 Data from Channels 54

9 Project Evaluation 55

10 Further Extensions 56

A Program Use 57

A.1 Installation 57

A.2 Running 57

A.3 Language Syntax 57

B Full Parser Implementation 60

C Code Samples **66**
C.1 Simple One-Shot Chat Server 66
C.2 Chat Server with Anonymous Channels 66
C.3 Handshake Protocol 66
C.4 Following Redirects and Getting a Resource 67

1 Introduction

The applied π-calculus is an expressive formal language describing computations as communicating processes. Its primary use in both industry and academia is to model security protocols. These models being built, they can then be statically analysed either by hand, or automatically by using a tool such as ProVerif.

For the purposes of static analysis, models built using applied π-calculus have proven very useful. Applied π-calculus has been used to verify numerous security protocols, including but not limited to [RS13]:

- Email certification
- Privacy and verifiability in electronic voting
- Authorisation protocols in trusted computing
- Authentication protocols and key agreement

However, these models are limited by the fact that they cannot currently be executed directly, as there is no existing language implementation.

Without an implementation, any models built using the applied π-calculus cannot be used to actually demonstrate protocols they are modelling, and so those models can be very difficult to debug.

1.1 Objective

The aim of this project is to provide an implementation of the applied π-calculus such that one might be able to build a model of a web protocol and then execute it interoperably with existing implementations written in PHP, JavaScript or any other web language. The resulting implementation will hopefully not only be a very powerful and concise language for reasoning about and implementing protocols, but a useful scripting tool for the web.

At the very least, we would like to be able to write something similar to:

```
in(a,M);
out(b,M);
```

Compile it and have it execute successfully; receiving a message in on channel a, and sending the same message out again on channel b. However, ideally we would like to write something like the following:

```
out(net,httpRequest(uri,headers,httpGet()));
in(net,(hs : list(Header), message : String));
out(stdout, message);
|
in(net,(hs : list(Header), req: HttpReq));
```

```
    out(process,req);
    in(process, resp : HttpResp);
    out(net,httpResponse(origin(hs),resp));
```

which would start one process which would send out an HTTP GET request on the channel net, and await a response. Once that response has been received, it will de-structure it and send the contents of the HTTP Response to stdout. Meanwhile, another process is set up to receive the same request on net, de-structure it into its component parts, then handle the request, and send back an appropriate HTTP response. The latter may well be out of the our abilities, however the we would aim to create something capable of handling something a little more impressive than the former.

1.2 Approach

We aim to build an interpreter for the applied π-calculus. The language we plan to do this in is Haskell, due to familiarity using both the language itself and the parsec library [LM01], a powerful parser combinator library.

We will also need to make some restrictions as to exactly what our version of the π-calculus can do. With it in theory being such an expressive language, the scope of this exercise is potentially enormous. These restrictions will be covered in the following section.

We will also be building a little web playground for our initial efforts to interact with. This will give us a good idea of how our implementation will interact with real world servers, if at all.

2 Background

2.1 Process Calculi

Process calculi , sometimes referred to as process algebras are a family of languages and models for describing concurrent systems. They allow for the description of communication and synchronization between two or more concurrent processes. The algebraic laws which govern process calculi allow the process descriptions they provide to be reasoned about easily. All process calculi allow for the following operations [Pro14]:

- Communication
- Sequential Composition
- Parallel Composition
- Reduction Semantics
- Hiding
- Recursion and Replication
- The Null Process

2.1.1 Communication

Processes are able to send messages between each other. Process calculi will generally have a pair of operators defining both input and output. Formally these are often $\bar{x}\langle y\rangle$ for a process sending out message y on channel x, and $x\,(v)$ for a process receiving a message on channel x and binding the variable v to the value of that message in subsequent processes. It is the type of data that can be sent/received by processes which sets apart different process calculi

2.1.2 Sequential Composition

Processes can potentially perform communications in order. This is signified by the sequential composition operator, often "." . A process may need to wait for input on channel x before continuing with other processes, which could be formally written $x\,(v)\,.P$

2.1.3 Parallel Composition

Processes can perform actions concurrently and independently. Process P and Q running in parallel, written $P|Q$ are able to communicate across any shared channels, however they are not limited to one channel only. These channels may be either synchronous, where the sending process must wait until the message is received, or asynchronous, where no such waiting is required.

2.1.4 Reduction Semantics

The details of reduction semantics are different for each process calculus, but the theory is the same. The process $\bar{x}\langle y\rangle.P|x\left(v\right).Q$ reduces to the process $P|Q\left[\frac{y}{v}\right]$, which is to say the following: the left hand process sends out message y on channel x and becomes the process P, and the right hand process receives a message (y) on channel x, binding that message to the variable v for the remaining processes in Q.

2.1.5 Hiding

The ability to hide a name in a process is vital for the control of communications made in parallel. Hiding the name x in P could be written $P\backslash\left[x\right]$.

2.1.6 Recursion and Replication

Recursion and replication allow for a process to continue indefinitely. Recursion of a process is a sequential concept and would be written $P = P.P$. Replication is the concurrent equivalent i.e. $!P = P|!P$

2.1.7 The Null Process

Finally, the null process, generally represented as 0 or $\emptyset$, does not interact with any other processes. It acts as the terminal process, and is the basis for processes which actually do things.

2.2 π-calculus and the Calculus of Communicating systems

The applied π-calculus [AF01] is an extension of π-calculus [MPW92] which itself as an extension of the work Robert Milner did on the Calculus of Communicating Systems (CCS) [Mil82]. All three languages are process modelling languages, that is to say that they are used to describe concurrent processes and interactions between them. CCS is able to describe communications between two participants, and has all of the basic process algebra components as above. π-calculus provides an important extension allowing channel names to be passed along channels. This allows it to model concurrent processes whose configurations are not constant.

2.3 Interpretation

Trying to interpret a process based language presents several difficulties from the offset. Such an interpreter needs to be able to generate processes, switch contexts, and perform cross-channel communication very quickly as these operations, which are normally considered computationally intensive, form the basis of any process calculus. [PT97] As such, it may be necessary either to reduce the feature set of the language in order to ensure that the interpreter performs acceptably.

2.4 The applied π-calculus

As mentioned before, the applied π-calculus is based on π-calculus, but it is designed specifically to model security protocols [RS13]. It is extended to include a large set of complex primitives and functions.

2.4.1 Syntax

The language assumes an infinite set of names and variables and a signature σ which is the finite set of functions and their corresponding arities[AF01]. A function with arity 0 is considered a constant. Given these, the set of terms is described by the following grammar:

$L, M, N, T, U, V ::=$	terms
$a, b, c, ..., s$	names
x, y, z	variables
$g(M_1, M_2, ...M_l)$	function application

The type system (or sort system) comprises a set of base types such as *Integer* and *Key*, but also a universal *Data*type. Names and variables can have any type. Processes have the following grammar:

$P, Q, R ::=$	processes
$\emptyset$	null process
$P \vert Q$	parallel composition
$P.Q$	sequential composition
$!P$	replication
$vn.P$	new
$ifM = NthenPelseQ$	conditional
$u(x).P$	input
$\bar{u}\langle N\rangle.P$	output

Where conditional acts as expected and ”new” restricts the name n in p.Processes are extended as follows with active substitutions.

The active substitution $\left[\frac{M}{x}\right]$ represents the process that has output M before and this value is now reference-able by the name x.

2.4.2 Simplified Syntax

As the Pict language did when creating an implementation of pure π-calculus we must first simplify the syntax of the language we are using [PT97]. Func-

$A, B, C ::=$	extended processes
P	plain process
$A \vert B$	process composition
$vn.A$	new name
$vx.A$	new variable
$\left[\frac{M}{x}\right]$	active substitution

tion application will remain the same, and the set of variables and names shall in theory still be infinite. We will do away with the null process, and assume that a process without a sequential process is implicitly followed by the null process.

$\emptyset$	`0`
$P \vert Q$	`P | Q`
$P.Q$	`P ; Q`
$!P$	`!P`
vn	`new x`
$ifM = NthenPelseQ$	`if p(M) then P else Q`
$u(x)$	`in(u,x)`
$\bar{u}\langle N\rangle.P$	`out(u,N)`
$\left[\frac{M}{x}\right]$	`let X = M in P`

This will be the syntax we refer to from now on, and which we will be attempting to interpret.

2.4.3 Starting Restrictions

The first build of our interpreter will only be able to handle a few basic types and functions. We will also only be concerning ourselves with HTTP traffic, and to begin with our channels will be for the most part duplex handles built from sockets [Net14b] handled by the Haskell Network library [Net14a]. The reasoning here is that getting to grips with the entirety of the low level C socket API (which is exposed by the Network.Socket module) would not be a productive use of time with respect to our main intended purpose (i.e. to handle HTTP traffic). By limiting our initial approach, we can familiarise ourselves with some of the concepts of sockets and later on we can expand to more generic sockets if necessary and once our language implementation is sound.

2.5 Haskell

Haskell is a pure non-strict functional programming language based on the λ-calculus. It is a strongly static typed language making it easy to ensure correctness of programs. It is highly expressive, but this combined with its laziness comes at a potential price in terms of execution time. We may well find that it simply is not possible to build a responsive enough system using Haskell, but there are several advantages to using it to build an interpreter.

2.5.1 Data Type

Haskell makes it trivial to create data types. As such we can easily use Haskell to build an abstract representation of our language, which we will later generate during the parsing process.

A small subset of this is presented below, we will go into more detail in Section 4

```
data PiProcess =
-- Our data type representing a process
                 In String String
-- Wait for message in on Channel and assign it to Variable
               | Out String String
-- Send out a message on a channel
               | Conc [PiProcess]
-- Perform a list of Processes concurrently
               | PiProcess ‘Seq‘ PiProcess
-- Perform two Processes sequentially

-- We can then define how our data type is to be printed,
-- here we choose to make it identical to the original input for readability:
instance Show PiProcess where
    show (In c m) =  "in(" ++ c ++ "," ++ m ++ ")"
    show (Out c m) =  "out(" ++ c ++ "," ++  m ++ ")"
    show (Conc procs) = intercalate "|" $ map show procs
    show (p1 ‘Seq‘ p2) = show p1 ++ ";" ++ show p2
```

We can test this as follows with the basic input from our introduction (of course these currently hold no intrinsic meaning, but this will be implemented later)

```
ghci>(In "a" "x") ‘Seq’ (Out "a" (TVar "x"))
in(a,x);
out(a,x)
```

It is also good to note that malformed structures will fail:

```
ghci>(In "a" "x") ‘Seq’ (Out "a" )

<interactive>:34:21:
    Couldn’t match expected type ‘PiProcess’
    with actual type ‘Term → PiProcess’
```

```
In the return type of a call of ‘Out’
Probable cause: ‘Out’ is applied to too few arguments
In the second argument of ‘Seq’, namely ‘(Out "a")’
In the expression: (In "a" "x") ‘Seq’ (Out "a")
```

2.5.2 Parsec

Parsec is a monadic parser combinator library for Haskell which is fast, robust, simple and well-documented [LM01]. We use parsec by building a series of low-level parsers and combining them into a single high level one. For example, if we start with a low-level parser to match brackets, from that we can build a higher level parser which can then return the contents of those brackets as a list of strings:

```haskell
import Text.ParserCombinators.Parsec

openB :: Parser Char
openB = char ’(’

closeB :: Parser Char
closeB = char ’)’

betweenB :: Parser [String]
betweenB = do{
            openB;
            [out] <- endBy line closeB;
            return out;
        }
    where
        line = sepBy word (char ’,’)
        word = many ( noneOf ",)")
```

We can test this parser using the parseTest function as follows:

```
ghci>parseTest betweenB "(sometext,somemoretext)"
["sometext","somemoretext"]
```

This function also fails on malformed input

```
ghci>parseTest betweenB "(sometextsomemoretext"
parse error at (line 1, column 22):
unexpected end of input
expecting "," or ")"
```

From here, it is only a short step to build a parser for the basic `in,out` syntax of the language. The following is a slightly untidy, but quickly built proof of concept parser to demonstrate the ease with which this can be achieved:

```
inOut :: Parser PiProcess
inOut = do {
        piIn;
        bContents <- betweenB;
        case bContents of
            [chan,message] -> return $ In chan message
            _ -> error (e "in(chan,message)")
        } <|> do {
        piOut;
        bContents <- betweenB;
        case bContents of
            [chan, message] -> return $ Out chan (TVar message)
            _ -> error (e "out(chan,message)")
        }
        where
            piIn= string "in"
            piOut=string "out"
            e x= "malformed input " ++ x ++ " expected"
```

And a demonstration of this in action (removing our custom show instance, as otherwise, because we chose to make it appear like the input, it seems like nothing is happening). Once more, malformed input throws errors depending on what exactly went wrong:

```
ghci>parseTest inOut "in(a,b)"
In "a" "b"
ghci>parseTest inOut "out(a,b)"
Out "a" b
ghci>parseTest inOut "ot(a,b)"
parse error at (line 1, column 1):
unexpected "t"
expecting "out"
ghci>parseTest inOut "Out(a,b)"
parse error at (line 1, column 1):
unexpected "O"
expecting "in" or "out"
ghci>parseTest inOut "out( a,b)"
Out " a" b
ghci>parseTest inOut "out( a ,      b)"
Out " a "       b
ghci>parseTest inOut "out  a ,      b)"
parse error at (line 1, column 4):
unexpected " "
```

```
expecting "("
ghci>parseTest inOut "out(a,b,c)"
*** Exception: malformed input out(chan,message) expected
```

2.5.3 Sockets and the Network module

Once we have parsed our input and built our syntax tree, we must then semantically analyse the input and generate the processes required. As mentioned before we will be using the Haskell Network module for our initial implementation. This provides a high-level interface to network functionality in Haskell. We create handles from incoming connections on sockets on the server side, and on the client side we generate a handle by connecting to a foreign host on a specified port. An example of this is as follows:

```haskell
import Control.Monad (forever,unless)
import Control.Concurrent (forkFinally, forkIO)
import Control.Monad.IO.Class (liftIO)
import System.IO (hGetLine, hPutStrLn)
import qualified Network as N


-- SERVER Process Code
server :: IO ()
server = do
    sock <- N.listenOn $ N.PortNumber 9000
    (handle,_,_) <- N.accept sock
    _ <- forkFinally (forever $ do
        msg <- hGetLine handle
        hPutStrLn handle msg) (\_ -> N.sClose sock)
    return ()

-- CLIENT Process Code
client :: IO ()
client = do
    handle <- N.connectTo "localhost" $ N.PortNumber 9000
    let loop = do
            line <- getLine
            hPutStrLn handle line
            msg <- hGetLine handle
            putStrLn msg
            loop
    loop

```

```
main :: IO ()
main = do
    _ <- forkIO $ server
    client
```

This small program creates two processes (which here are two GHC threads created with *forkIO*) . The client continuously reads from stdin and then sends the information over the socket, and the server receives the data from the socket and sends it back. In other words, written in our version of the applied π-calculus it does:

```
!(in(stdin,x);
out(socket(0.0.0.0,9000),x);
in(socket(0.0.0.0,9000),y);
out(stdout,y))
|
!(in(socket(0.0.0.0,9000),x);
out(socket(0.0.0.0,9000),x))
```

Which is something akin to our basic program from our introduction.

3 Package Overview

Our implementation is broken into 6 modules, we give a brief overview of each here

3.1 TypDefs

This module describes all of the data types, type synonyms and instances for types that are used in our implementation. We will discuss these in depth in section 4.1

3.2 Parser

This module contains all of the parsing logic for our program. It exports the functions *readTerm* , *readProcess* , and *readProcesses*, which parse a Term, a PiProcess, and several PiProcesses respectively. See section 4.2

3.3 Channel

This module contains the logic for channels. The most important functions this module exports are

newChan creates either the server end or client end of a socket channel (depending on which *BuildType* is passed as an argument)

stdChan creates a channel for a standard input/output (i.e. stdout,stderr,stdin)

newDummyChan creates a channel which can only communicate within a process

See section 5

3.4 Primitives

This module describes the primitives for our implementation of the language, and exports an associative list of Strings and functions for manipulating Terms : *primitives* This associative list is imported by the PiCalculus module where the strings are bound to their Term manipulating functions in the environment. We discuss the implementation and function of these primitives in section 6

3.5 PatternMatching

This module contains the logic for our pattern matching system, and exports a single function *match* We discuss how we achieve our matching in section 7

3.6 PiCalculus

This is the main module of the program. This is where the component parts are brought together, and where processes are evaluated and executed. This module is discussed in section 8

4 Types and Parsing

In this section, we will describe the data structures we use to represent the the processes, channels and data in our program. We will also briefly cover the way in which we used Parsec to parse our input and create build those structures.

4.1 Types

4.1.1 Terms

The most fundamental data type in our implementation is the *Term*. Terms are used to represent data, variables and functions.

```
data Term = TStr String
          | TNum Integer
          | TBool Bool
          | TBS ByteString
          | TData HttpData
          | TPair (Term, Term)
          | TList [Term]
          | TVar Name (Maybe Type)
          | TFun Name [Term]
            deriving (Eq)
instance Show Term where show = showTerm

type Name = String
```

We can see that terms derive Eq, meaning that two terms with the same contents, when compared with == will return *True*. We derive our own instance of Show, which appropriately unwraps each of the above. *Name* is simple a type synonym for *String*.

TStr, TNum, TBool, and TBS are just simple wrappers for their respective types (String, Integer, Bool and ByteString), to allow us to manipulate them together in Haskell's type system.

TData is a wrapper for HttpData, which is explained in 4.1.1

The next few are slightly more complicated. TPair is a wrapper around Haskell's own Tuple type, which can recursively hold two Terms. TList is a wrapper around Haskell's List type, which holds a list of Terms.

TVar is used as a variable. It is constructed using a Name (String) and a Maybe Type. Maybe is a data type in the Haskell Prelude that allows either for a value (using the constructor Just, e.g. Just 1) or no value (using the Nothing constructor). This means that we allow a variable to have no type, but it is possible to construct a variable with a type. Type itself only has two constructors:

Type

```
data Type = HttpRequest
          | HttpResponse
          deriving (Eq, Read, Show)
```

These are currently only used when receiving data in from an external channel, which is explained in section 5

Finally, a TFun is constructed with a Name and a list of Terms. They represent a function (either over terms or over processes).[1]

HttpData HttpData is itself a wrapper around the *Response* and *Request* type constructors from the Network.HTTP.Base [HTT14a] module. Again, we began our implementation with these being special cases of TFun, however we found ourselves converting to and from Responses and Requests that we cut out the middle TFun and added them to our Term structure.

```
data HttpData = Resp (Response String)
              | Req  (Request String)

instance Show HttpData where show = showHttpData
instance Eq HttpData where (==) = eqHttpData
```

We created our own instances of Show and Eq in order to unwrap the responses/requests from our constructors.

We also had to unwrap responses and requests from our constructors when creating an instance of HasHeaders [HTT14b] which is a typeclass used in many header manipulation functions of the HTTP module

```
instance HasHeaders HttpData where
    getHeaders (Resp r) = getHeaders r
    getHeaders (Req r)  = getHeaders r
    setHeaders (Resp r) = Resp . setHeaders r
    setHeaders (Req r)  = Req . setHeaders r
```

[1] It would have been possible to do away with both TPair and TList, and simply had a special case of TFun (which is in fact how they are constructed in the Parser) but we found that this quickly became tedious, so we created these special types which made manipulating them easier.

4.1.2 Processes

Processes are modelled with the *PiProcess* data type.[2]

```
data PiProcess = Null
               | In   Term Term
               | Out  Term Term
               | New  Term
               | PiProcess ‘Seq‘ PiProcess
               | Conc [PiProcess]
               | Replicate PiProcess
               | Let Term Value (Maybe PiProcess)
               | If Condition PiProcess PiProcess
               | Atom Term
                 deriving (Eq)

instance Show PiProcess where show = showPi
```

This data type derives Eq and we defined our own Show instance (the output of which is identical to our defined syntax for the language).

The constructors are described as follows:

Null Represents the null, or terminal process

In Represents a process receiving data in on a channel. The first term should be either a TFun which evaluates to a channel, or a variable pointing to one, and the second should be a variable

Out Represents a process sending data out on a channel. The first term should evaluate to a channel, as in In, and the second can be a term of any type

New Represents the reservation of a variable name

Seq Represents the sequential composition of two processes

Conc Represents the concurrent composition of a list of processes

Replicate Represents the infinite concurrent composition of a process with itself

Let Represents the assignment of a variable, or function to a value or function body. The Maybe PiProcess here represents the possibility either of locally assigning the variable in the process found within the Just constructor, or globally assigning the variable if the Maybe is Nothing.

[2] N.B. This is merely how processes are modelled, not how they are actually implemented, this is covered in section 5.2

If Represents the conditional execution of either the first process if the Condition is true, or the second process if it is false. Conditions are explained in section 4.1.2

Atom Represents a reference to a process, either as a TFun or a TVar.[3]

The Atom constructor was added to the language in order to allow calling process functions. We encountered a problem when we were implementing the language in that we had the ability to define process functions and references to processes using let, but had no way of calling them within our PiProcess type at the time. Our solution was to add this constructor, which is used by prepending "&" to a bare Term. See section 7.1 for an example of its use.

Condition Conditions are modelled with a very simple data type. They currently only allow a single constructor:

```
data Condition = Term `Equals` Term deriving (Eq)

instance Show Condition where show = showCond
```

We had thought that we would need to extend this, however we found that no other conditional types were needed. There is still the possibility to provide this extension later.

4.1.3 Channels

A Channel is modelled as follows:

```
data Channel = Channel {
                 send          :: String -> IO ()
               , receive       :: IO String
               , extra         :: [String]
               }
```

We have a single constructor, Channel, which takes two functions and a list of strings:

send Any function which takes a string and returns IO (). It is of course intended that this sends this string to a destination, although what this destination is and how this is achieved is not specified here.

[3] These must evaluate to a PiProcess, or else an error will occur

receive Any function which produces a String in the IO monad. Again, this function is intended to retrieve this string from some source, but once again this is not specified here.

extra Contains any extra data that might be associated with the channel. This is used in the serialisation of Channels which we explain in more detail in section 5.1.1.

The actual implementation of a Channel is left deliberately vague, to allow for many different types of channels to be built. In section 5 we give our implementations.

4.1.4 Values

Values are the data type we use to represent all of our types in the single environment we pass around our evaluation strategy (see section 8)

```
data Value = Proc PiProcess
           | Term Term
           | Chan Channel
           | PrimitiveFunc TermFun
           | Func {params :: [String] , body :: Value, closure :: Env}

instance Show Value where show = showVal
instance Eq Value where (==) = eqvVal
```

The first three are simply wrappers around the data types we have described so far. PrimitiveFunc models a primitive Term function, and contains a TermFun, defined in section 4.1.6 Func models a user defined function, which holds a list of parameters (as a list of Strings), a function body (itself a Value), and a closure within which the function is evaluated (as an Env, the type of which is described in section 4.1.5). The implementation of these two will be explained in section 8

We define our own instances of Show and Eq

```
showValue :: Value -> String
showValue (Proc p)  = show p
showValue (Term t)  = show t
showValue (Chan c)  = show (convert c)
    where
        convert ch = TFun "<chan>" (map TStr ex)
            where ex = extra ch
showValue (PrimitiveFunc _)  = "<primitive>"
```

```haskell
showValue (Func {})           = "<user function>"

eqvVal :: Value -> Value -> Bool
eqvVal (Proc p1)  (Proc p2) = p1 == p2
eqvVal (Term t1)  (Term t2) = t1 == t2
eqvVal _ _ = False
```

Our show implementation is uninteresting, besides for showing Channels. This is essentially a hack for serialising channels[4]

4.1.5 Environment

```haskell
type Env = IORef (Map String Value)
```

We model our environment as an IO Reference to a Map of Strings to Values. An IO Reference, or IORef [Dat14] can be thought of as a pointer to some mutable state in the IO monad. Our environment will obviously change through our implementation, and so we need to model this mutability. We chose IORefs over other possible mutability models because they are fast, and we are already operating within the IO monad for a lot of our evaluation strategy.

4.1.6 Error Handling

Finally, we have our data type representing errors in our evaluation and/or parsing.

```haskell
data PiError = NumArgs Name Integer [Value]
             | TypeMismatch String [Value]
             | Parser ParseError
             | UnboundVar String String
             | NotTerm Name Value
             | NotFunction String String
             | NotChannel String
             | NotProcess String
             | PatternMatch Term Term
             | Default String

instance Show PiError where show = showError
```

[4] When we receive a term in a phi process, we check to see whether it has the special name ”<chan>” and then try to build a channel from the data in the arguments, we go into this in more detail in section5.1.1

The purpose of these errors is fairly clear from the names of their constructors.

We represent functions which can throw errors in two ways. They can have two return types:

ThrowsError a meaning they return either a PiError (which is raised using throwError from the Control.Monad.Error module [Con14d]) or return something of type a (using return, which lifts a value into the Either monad [Con14c])

IOThrowsError a meaning they can return a PiError or a something of type a, but can also perform operations in the IO monad, internally using the ExceptT monad [Con14e] which is an instance of the MonadIO typeclass [Mon14].

The type synonyms for these are as follows

```
type ThrowsError a   = Either  PiError a
type IOThrowsError a = ExceptT PiError IO a
```

We also have a type synonym for TermFun s

```
type TermFun = [Term] -> ThrowsError Term
```

So a TermFun is a function which takes a list of Terms and can either throw an error or return a Term

4.2 Parser

As mentioned in section 2.5.2 we have used Parsec in order to build our parser. The Parser module exports the following functions:

```
readTerm :: String -> ThrowsError Term
readProcess :: String -> ThrowsError PiProcess
readProcesses :: String -> ThrowsError [PiProcess]
```

All three of which take a String and they return a Term, a PiProcess or a list or PiProcesses respectively in the ThrowsError monad.

However, these are just wrappers around the two workhorses of this module, which are

```
parseTerm :: Parser Term
```

and

```
parseProcess :: Parser PiProcess
```

4.2.1 parseTerm

As we have already said, when building parsers in Parsec, we construct a larger parser from a set of smaller parsers. The body of parseTerm is as follows:

```
parseTerm =  try parseAnonChan
         <|> try parseTFun
         <|> parseTVar
         <|> parseTNum
         <|> parseTStr
         where
            parseAnonChan = do
                _ <- char '{'
                spaces
                arg <- many parseTerm
                spaces
                _ <- char '}'
                return (TFun "anonChan" arg)
```

So when we parse a term, we first try to parse an anonymous channel, which is represented as a TFun (more on this in section 5) and then a general TFun, then a TNum, then a TVar, and finally a TStr. The final two are somewhat uninteresting, but the parseTFun and parseTVar functions contain a couple of special cases.

```
parseTFun :: Parser Term
parseTFun = do
            name <- readVar
            spaces
            args <- bracketed $ sepBy parseTerm paddedComma
            return $ case (name,args) of
                ("pair", t1:t2:_)  -> TPair (t1,t2)
                ("list", _ )       -> TList args
                _                  -> TFun  name args
```

In most cases this parser will return a TFun, but in the case that the name of the TFun we are parsing is ”pair” we return a TPair, and in the case that it is ”list” we return a TList.

```
parseTVar :: Parser Term
parseTVar = do
```

```
        v <- readVar
        case v of
            "true"  -> return $ TBool True
            "false" -> return $ TBool False
            _       -> do
                t <- myOptionMaybe parseType (paddedChar ':')
                return (TVar v t)
```

This parser has an "escape clause" for TBools. When the variable we have parsed is "true" or "false" we return the appropriate TBool. In the case that it is neither, we then attempt to parse the Type of the variable when there may or may not be one. We do this using the myOptionMaybe parser, which has the type:

```
myOptionMaybe :: Show b => Parser a -> Parser b -> Parser (Maybe a)
```

So given a parser of some type a, and a parser of something b which is an instance of Show, we return a parser of type Maybe a. Here we pass myOptionMaybe a Parser Type and a Parser Char (Char being an instance of Show) and this will give us a Parser Maybe Type, with Maybe Type being the type of the second argument to the TVar constructor.

myOptionMaybe is in fact just a specialist call to a more general function myOption

```
myOptionMaybe parser = myOption Nothing (liftM Just parser)
```

myOption being defined as:

```
myOption :: Show b => a -> Parser a -> Parser b -> Parser a
myOption opt parser sep = try (notFollowedBy sep >> return opt) <|> (sep >> parser)
```

This function is used several times throughout parseTerm and parseProcess. When given a parser for a separator, we check if the separator can be parsed using notFollowedBy. If not then we return the first argument to myOption, opt (of type a), however, if it can be parsed (i.e. notFollowedBy fails) then we parse it, ignoring its result, and then use the parser "parser" to continue parsing the rest of the input.

There is in fact a function option (and associated optionMaybe) in the Parsec library, which does nearly the same thing. However, they consume input even if the sep parser fails, which was causing havoc with our implementation, and so we wrote our own version.

4.2.2 parseTerm

```
1  parseProcess :: Parser PiProcess
2  parseProcess = liftM (\ps -> case ps of
3                           [p] -> p
4                           _   -> Conc ps) (sepBy1 parseProcess' (char '|'))
5      where
6      parseProcess'  = bracketed parseProcess'' <|> parseProcess''
7      parseProcess'' = parseNull
8                   <|> try parseIf
9                   <|> parseIn
10                  <|> parseOut
11                  <|> parseLet
12                  <|> parseReplicate
13                  <|> parseNew
14                  <|> parseAtom
```

This function is best looked at backwards. From line 7, we have the main bulk of the parser, i.e. the combination of the smaller parsers. In line 6 we see a bit of fidgeting for the parsing of bracketed processes, and in line 4 we see that we separate these (potentially bracketed) processes with a — character. The lambda function in lines 2-4 simply avoids unnecessary nesting of singleton lists in Conc constructors.

Of note here is the lack of a parser for sequentially composed processes. The reason for this is that only a subset of processes can actually be composed sequentially. That is to say, In,Out,New, and Atom. As such parseSeq is only called within the parsers for these four. Also of note with regards to the parseSeq parser is that any process that can be sequentially composed with another, can very well be composed with the Null process, which can either be written explicitly or through omission of a proceeding semicolon and process. This is achieved using the myOption parser we have previously mentioned:

```
parseSeq :: PiProcess -> Parser PiProcess
parseSeq p1 = do
            p2 <- myOption Null parseProcess (paddedChar ';')
            return (p1 `Seq` p2)
```

The final subsection of the parser we will discuss is parseLet.

```
parseLet :: Parser PiProcess
parseLet = do
            _ <- string "let"
            spaces
            name <- parseTerm
            paddedChar1 '='
            val <- try (liftM Proc parseProcess) <|> liftM Term parseTerm
            p <- myOptionMaybe parseProcess (paddedStr1 "in")
            return (Let name val p)
```

This is interesting because not only do we have the potential for a let clause without an ”in” which is handled by myOptionMaybe, but we also have the possibility of either parsing a process or a term for the value which is being assigned to the variable. Because of this we must use either the Proc or Term constructors to turn the PiProcess or Term into a Value[5]

The full code for our parser implementation can be found in appendix B

[5] 4.1.4

5 Channels and Processes

Process modelling languages are based upon the principles of processes communicating across channels. As such this part of our implementation was fairly key.

5.1 Channels

We considered what would be the best way to model this vital part of our implementation, and decided upon a structure that was as flexible as possible. As described in section 4.1, our data type representing a Channel is as follows:

```
data Channel = Channel {
                 send          :: String -> IO ()
               , receive       :: IO String
               , extra         :: [String]
               }
```

In other words, a *Channel* is modelled as an object with a *send* function, which takes a string and sends it somewhere, a *receive* function, which returns a string in the IO monad, and some other data, *extra*, as a list of strings.

As we have touched upon before, we have put no restrictions what exactly the send and receive functions do. In most cases it makes sense that the string be sent to a receiving party on the other end, an receive be receiving from a sending party. However, it is conceivable that we create channels that do nothing of the sort. Our implementation allows us to create several completely different channel types with the same interface. Having said that, using specific channels incorrectly will result in unwanted behaviour (trying to write to stdin for example, will crash the interpreter).[6] We also chose not to try and divorce our channel implementation from the rest of our modules as much as possible. It is the main program's responsibility to convert data into Strings before sending them onto a channel.

The types of channel we have implemented so far are

1. Standard Channels e.g. stdin etc.
2. Dummy Channels
3. Socket Channels

and we will explain how each of this is implemented below.

[6] We considered shielding this kind of operation from crashing, however it only really applies to the 3 standard file handles, and one should be alerted fairly swiftly if one were to mix them up

5.1.1 Serialising Channels

Before we get into the implementation of types of Channels we would like to say a brief word on serialising Channels. If a Channel has a non-empty extra section, then we say that it is serialisable. Currently the only channels which are serialisable are Socket Channels. As we will discuss in section 5.1.4, these contain information in their extra strings regarding where the server end of the socket is, which can then be connected to by whomever receives the channel. The functions which achieve this are:

```
makeExtra :: [String] -> [String] -> [String]
makeExtra = zipWith (\a b -> (a ++ dataBreak : b))
```

which zips two lists of strings together with a special character in the middle (currently it is just '#') to make our extra strings;

```
showValue (Chan c)  = show (convert c)
    where
        convert ch = TFun "<chan>" (map TStr ex)
            where ex = extra ch
```

which makes the "<chan>" TFun; and

```
getChannelData :: [String] -> Maybe (String, Integer)
getChannelData strs = do
        let ex = map (second tail . break (==dataBreak)) strs
        host         <- lookup hostSig ex
        port         <- lookup portSig ex
        return (host,read port)
```

on which breaks down the strings into their constituent parts on the receiving end. This final function will fail and return Nothing if the data is incomplete, or corrupted.

We have no qualms about admitting that the serialisation/de-serialisation mechanism we have in place for Channels is a bit of a "hack", piggy backing on our ability to parse a special "¡chan¿" TFun at the receiving end, but as it stands it works perfectly well, and it allows us to implement some interesting and exciting features.

5.1.2 Standard Channels

These are a simple wrapper around the three standard input and output channels stdin, stdout, and stderr.

A standard channel is created by passing the handle for this channel to the following function:

```haskell
stdChan :: Handle -> Channel
stdChan han = Channel write rd []
    where
        write = hPutStrLn han
        rd = hGetLine han
```

Where the write function simply becomes (hPutStrLn han), which prints a string into a handle (appending a newline) and read becomes (hGetLine han). As mentioned before, with standard channels only one of these will work at a time, but it is left to the programmer not to mix them up. In the main module these channels are created and assigned to their standard names when the program is started.

5.1.3 Dummy Channels

As it stands, there is only ever one dummy channel running at one time while phi is running. It is accessible through the variable ”localnet”. It is designed to be used to pass messages between channels running in a single process (which can itself obviously be built of several concurrent and sequential processes). We originally had dummy channels as a special subset of Socket Channels. This worked well enough, but we felt there was a lot of overhead for what could essentially be achieved with the implementation that we ended up with. That implementation is a wrapper around the Control.Concurrent.Chan [Con14a] type. This is a model of an unbounded FIFO channel, implemented as a linked list of MVars [Con14b], which are mutable variables.

```haskell
newDummyChan :: IO Channel
newDummyChan = do
    chan <- Chan.newChan
    return (Channel (Chan.writeChan chan) (Chan.readChan chan) [])
```

So in this case we write our string to the write end of the channel and then read our string from the read end of the channel.

5.1.4 Socket Channels

Where the implementation of the other two channel types was somewhat trivial, the third Channel type is where most of the meat of our implementation comes. We can build a socket channel using the newChan function.

```
newChan :: BuildType -> String -> Integer -> IO Channel
newChan t host port =
            case t of
                Init    -> newChanServer port
                Connect -> newChanClient host port
```

To this function we pass a BuildType, which is either Init or Connect, a hostname, and a port. In the case of an Init build, we ignore the host, as we are building the server end of the channel on this machine on port ”port”. If we are connecting, then we are building the client end to some foreign server.

```
newChanServer :: Integer -> IO Channel
newChanServer cp = N.withSocketsDo $ do
    hanVar <- newEmptyMVar
    _ <- forkIO $ do
        sock <- N.listenOn $ N.PortNumber $ fromIntegral cp
        (clientHandle,_,_)  <- N.accept sock
        putMVar hanVar clientHandle
    currentHost <- getHostName
    let ex = makeExtra [hostSig,portSig] [currentHost, show cp]
    return (Channel (send’ hanVar) (receive’ hanVar) ex)
```

[7]

So with a ChanServer, we create a new MVar, and then create a thread using forkIO [IO14] which listens on a socket at port ”cp”, and then accepts the first connection. Once we have accepted a connection, we place the handle generated by this and place it in the MVar we created before the call to forkIO. Next we create our extra data for this channel. We get the hostname of the current machine, and use the function makeExtra to make our serialising data ”hack”. We then pass ”hanVar” to send’ and receive’ to create the send and receive functions. We will discuss send’ and receive’ momentarily

```
newChanClient :: String -> Integer -> IO Channel
newChanClient hostName hostPort = N.withSocketsDo $ do
    hanVar <- newEmptyMVar
    _ <- forkIO $ do
        serverHandle <- waitForConnect hostName $ N.PortNumber $ fromIntegral hostPo
```

[7] N.withSocketsDo is a function for initialising sockets on Windows, on Unix systems it does nothing

```
        putMVar hanVar serverHandle
    return (Channel (send’ hanVar) (receive’ hanVar) ex)
    where
       ex = makeExtra [hostSig, portSig] [hostName,  show hostPort]
```

With a ChanClient, we create an MVar again into which we will place a handle, and again fork a thread using forkIO. This forked thread then waits until a connection is made to the given host and port number, and places the handle made from this connection into the MVar. Again we pass the MVar containing the handle to send’ and receive’ for our send and receive functions.

send’ and receive’ and emptyHandle These three functions are all helpers to socket channel construction.

send’ simply takes the handle from the given MVar and print the given string into the handle

```
send’ :: MVar Handle -> String -> IO ()
send’ hanVar msg = do
        han <- takeMVar hanVar
        hPutStrLn han msg
        putMVar hanVar han
```

receive’ reads the handle from the MVar (that is to say, it does not remove it), and then empties it using emptyHandle

```
receive’ :: MVar Handle -> IO String
receive’ hanVar = do
        han <- readMVar hanVar
        fmap unlines (emptyHandle han)
```

emptyHandle gets a line from the handle, then checks to see whether the handle contains any more input, assuming not if an error is thrown, and then returns the retrieved line in a singleton list if there is no more input or if there is we recurse on emptyHandle h. This function will return a list of strings containing the contents of a handle.[8]

```
emptyHandle :: Handle -> IO [String]
emptyHandle h = do
```

[8] There is a function hGetContents in the System.IO module [IO14], however it leaves the handle it is called on ”semi-closed” with no way of taking it out of this state, so we had to implement our own version

```
  line <- hGetLine h
  more <- hReady h `catchIOError` (\_ -> return False)
  if more
    then fmap (line:) (emptyHandle h)
    else return [line]
```

These socket handles are very versatile in and of themselves. They can be used to connect to other phi instances, or indeed to service running on any port on any machine.

5.1.5 Other Handle Types

As we mentioned at the start of this section, it is entirely possible to create a handle which at first might not seem useful, but is still compatible with our system. One could imagine creating a channel which produced random strings from a seed string sent to it, which might have some cryptographical use. Or perhaps a channel connected directly to a database of some description. Our implementation is completely open to this kind of extension.

5.2 Processes

When it came to deciding how to implement Processes, we had a couple of options. We briefly considered the System.Process module and a few other variants, but quickly came to realise that GHC threads were our best bet. They are lightweight, and provide level of concurrency we need without being overly complicated. We will discuss how these are used in section 8

6 Primitives and Term Manipulation

The Primitives module exports an associative list

```
primitives :: [(Name,TermFun)]
```

which maps the names of primitive functions to *TermFun* s. As explained in section 4.1,

```
    type TermFun = [Term] -> ThrowsError Term
```

and

```
    type ThrowsError a = Either PiError a
```

This means each of these functions will be passed a list of terms, and then has the possibility of returning a result (a *Term*) or an error (a *PiError*) As such, any malformed or incorrect inputs will fail gracefully and be handled appropriately.

We have implemented a large set of primitives with which you can manipulate Terms, and some specific to manipulating HTTP data. In this section we go over the primitive functions available to the user, and explain their purpose and implementation.

6.1 Definitions and Explanations

6.1.1 List, Pair and Number Functions

These are the basic term manipulation functions. Those familiar with Haskell should find these very familiar.

`add(num1, num2)` ⇒ `(num1 + num2)`

Returns the sum of two given numbers.

`fst(pair(a,b))` ⇒ `a`

When given a pair of terms, this returns the first element of the pair.

`snd(pair(a,b))` ⇒ `b`

When given a pair of terms, this returns the second element of the pair.

`cons(x ,list(a,b..))` ⇒ `list(x,a,b..)`

Given a term and a list of terms, we return a list with the term placed at the front (N.B. lists are heterogeneous in our implementation).

`head(list(a,b..))` ⇒ `a`

Returns the first element of a given list.

`tail(list(a,b..))` ⇒ `list(b..)`

Discards the first element of the list, and returns the remaining elements.

`append(list(a..),list(b..))` ⇒ `list(a..,b..)`

Appends list b to the end of list a

6.1.2 HTTP Data Functions

`httpReq(uri, headers(..), method)` ⇒ `request : HttpRequest`

When given a uri, which can either be generated by the *uri* function below, or given as a string (e.g. "www.google.com/index.html"), a list of headers, which again can be generated using *headers* or by passing a raw *list* of headers, and a request method (at the moment this is limited to *httpGet*(), *httpPost*(), *httpHead*() corresponding to GET, POST and HEAD requests) will generate an HTTP request.

`httpResp(responseCode, reason, headers(..), body)` ⇒ `response : HttpResponse`

When given a response code, (as a number e.g. 404) list of headers, a reason, (as a string e.g. "Not Found"), a list of headers, which can be generated using *headers* or by passing a raw *list* of headers, and a reponse body (as a string e.g. "<HTML>..</HTML>")

`headers(h1,h2..)` ⇒ `list(h1,h2..)`

headers is simply a synonym for list, but is used for clarity when generating HTTP Data.

`header(name,value)` ⇒ `header`

When given a name and a value, both strings, *header* generates a header. Internally this is simply appending the two strings with a colon inbetween the two. (i.e. $header("Content-Length","348")$ => "Content-Length: 348").

`cookies(c1,c2..)` ⇒ `list(c1,c2..)`

cookies is also a synonym for list, but is again used for clarity when generating lists of headers.

`getHeaders(httpData)` ⇒ `headers(..)`

When given a piece of HTTP data (either generated by the above *httpReq* and *httpResp*, or received from an external source), return the of headers of the data.

`insertHeader(header,httpData)` ⇒ `httpData`
`insertHeader(header,headers(h1,h2..))` ⇒ `headers(header,h1,h2..))`

This function is given a header (either generated by *header* or given as a raw string) and either a piece of HTTP data or a list of headers, and returns the data or the list of headers with the header inserted.

`getHeader(headerName, httpData)` ⇒ `headerVal`

When given a header name, such as "Content-Length", and a piece of HTTP data, returns the associated header's value. If there is no such header in the data's headers then it returns an error.

`getCookie(httpData) ⇒ cookie`

When given a piece of HTTP data returns the cookie from that data.

`setCookie(cookie, httpData) ⇒ httpData`

When given a cookie and a piece of HTTP data returns the data with the cookie inserted.

`uri(host,resource) ⇒ uri`

When given a host (as a string, e.g. "www.google.com"), and a resource (again a string, e.g. "index.html") returns a uri pointing to that location. Internally this appends the two strings with "/" inbetween (i.e. *uri("www.google.com","index.html")* => "www.google.com/index.html").

`rspCode(response) ⇒ code`

Given an HTTP response returns the response code from this response.

Several of the functions which retrieve data from lists and HTTP data can be achieved using pattern matching, which is explained in Section 7, but are provided for convenience.

6.1.3 Cryptographic Functions

The following functions are used for manipulating terms cryptographically. Currently, many of these functions are barebones implementations, and are not cryptographically sound. The reasons for this is as follows, in [RS13] it is stated that the behaviour of these functions is "captured by the smallest equational theory satisfying:"

```
sdec(x, senc(x, y))            == y
adec(x, aenc(pk(x), y))        == y
getmsg(sign(x, y))             == y
checksign(pk(x), sign(x, y))   == true
```

meaning that as long as these equalities hold true, then details of the implementation are unimportant. Many of these encryption and decryption cryptographic funtions require random seeding and other stateful operations. Ordinarily, this would not be an issue. However, Haskell is a pure language, and so stateful computations are not the norm, and would require one of two things to acheive:

1. We could place all the operations of the primitives module into the *IOThrowsErrora* Monad which is an instance of the *MonadIO* typeclass [Mon14] meaning that we can use *liftIO* to expose the *IO* monad and perform stateful computations. This would require lifting all of the *ThrowsErrora* monadic code, which seems wasteful considering that it is only these primitive functions that would require lifting.

2. We use the *System.IO.Unsafe* [Uns14] module to perform stateful computations outside of the *IO* Monad. However, this has potentially

unpredictable results, and it feels wrong to ”cheat” the purity of the language.

As neither of these are particularly appealing, we chose to make our implementation a basic one. In Section 10 we will discuss in more detail how we would handle this going future.

```
hash(msg) ⇒ hash
```

Given a message, which is either a plain string or a string of bytes, returns a hash of the message as a string of bytes. Internally, we hash using the MD5 scheme, but this choice was fairly arbitrary.

```
mac(key,msg) ⇒ mac
```

Given a key, as a string of bytes, and a message, again either a plain string or string of bytes, returns a MAC of the message. Internally, the hashing scheme is MD5 also.

```
pk(k) ⇒ pk
```

Given a private key, generates an associated public key.

```
sign(k,msg) ⇒ signed
```

Given a private key and a message, signs the message using the key.

```
getmsg(sign(k,msg)) ⇒ msg
```

Given a signed message, returns the message.

```
checksign(pk(k),sign(k,msg)) ⇒ true
checksign(pk(k),sign(_,msg)) ⇒ false
```

Checks whether a message was signed using a given public key.

```
senc(k,msg) ⇒ encryptedMsg
sdec(k,senc(k,msg)) ⇒ msg
```

Symmetrically encrypt and decode a message given a key.

```
aenc(pk(k),msg) ⇒ encryptedMsg
adec(k,aenc(pk(k),msg)) ⇒ msg
```

Asymmetrically encrypt and decrypt a message with a public key / private key system.

Hopefully it is clear that we have built this set of primitives in such a way that allows for the user to compose them easily and build their own functions. An example of this can be found in the *pilude*: $getLocation(httpRequest)$ – defined as $getHeader(”location”, httpRequest)$

7 Pattern Matching

Pattern Matching is the act of searching a given structure for a set of sub-structures and matching against them. This is best explained using an example, so in Haskell, pattern matching on a list would look like this:

```
patternMatchExample :: IO ()
patternMatchExample = do
        let list = [1,2,3]
        let first:rest = list
        print first
        print rest
```

and running this example gives the following result

```
ghci>patternMatchExample
1
[2,3]
```

So to explain:

1. We create a list with the elements 1,2 and 3
2. We pattern match the list against the pattern (*first* : *rest*) which will match *first* to 1 and *rest* to the remainder of the list (i.e. $[2, 3]$)
3. We output *first*

In phi we would like to be able to write

```
let l = list(1,2,3) in
    let list(first,rest) = l in
        out(stdout,first);
        out(stdout,rest)
```

and for it to give the result:

```
% phi patternMatch.phi
1
list(2,3)
```

When we first set out on our implementation, we had little to no idea how we were going to achieve pattern matching. It was a concept with which we were quite familiar, having used it in Haskell extensively, but we had never implemented anything like it before. Initially we tried to find any existing packages that might help us, but our search came up short. There were a few libraries which seemed like they might have been helpful, but ultimately were so badly documented that it was impossible to tell exactly what they did and how to use them. Having failed to find a library, we decided we would have to build our own system from scratch. As such we began to research how the functionality was implemented in other languages, which quickly lead us to Simon Peyton Jones’ (one of the main designers of Haskell) book The Implementation of Functional Programming Languages [Pey87]. Chapters 4 and 5 go into the semantics and efficient implementation of pattern matching in great depth. However, we quickly realised that the implementations we were reading about were far above what we currently needed for phi. This is because phi does not (currently) allow users to define their own types and structures beyond combining the ones already in the language, so realised we could get away with a fairly basic pattern matching strategy. This is described in the next section.

7.1 Implementation

Our PatternMatching module exports a single function

```
match :: Term -> Term -> ThrowsError [(Name,Value)]
```

which we use to pattern match two terms and return an associative list of variable names and values against which these names have matched (all within the *ThrowsError* monad, again meaning that this function can fail gracefully). This function is just wrapper around the main workhorse of this module

```
match’ :: Term -> Term -> ThrowsError [(Name,Term)]
```

Which does almost the same thing, but returns a list of names and terms instead.

Pattern Matching generally operates over tree-like structures. There are currently four of these in our pi-calculus implementation (not including wrappers such as *headers* and *cookies*):

1. `list(a,b..)`
2. `pair(a,b)`
3. `httpReq(uri,headers,method)`
4. `httpResp(code,reason,headers,body)`

As it stands we can handle all of these. However, if we were to add new structures to our implementation, it would require us to extend our match’ function.

The body of this function is as follows

```
match’ (TVar name _) term = case name of
                               ’_’:_ -> return []
                               _     -> return [(name,term)]
match’ (TPair (m1, m2)) (TPair (t1,t2)) =
                 liftM2 (++) (match’ m1 t1)  (match’ m2 t2)
match’ (TList (m:ms)) (TList (t:ts)) = do
               bind <- match’ m t
```

```
7                 rest <- case (ms,ts) of
8                    ([v],[t']) -> match' v t'
9                    ([v],_)    -> match' v $ TList ts
10                    _         -> match' (TList ms) (TList ts)
11                return $ bind ++ rest
12 match' l@(TList _) (TData d) = match' l $ dataToList d
13 match' t1 t2 = throwError $ PatternMatch t1 t2
```

Going through this line by line:

1. The basic principle of our matching function is that a bare variable should match against anything
2. If a variable name begins with an underscore, we ignore the binding. This is functionality we have borrowed from Haskell, allowing for wildcard pattern matching[9]
3. If the variable name does not begin with an underscore we return a singleton list containing a pair of the name and the matched term
4. A pair matches against another pair recursively
5. A list can match another list, provided they both have at least one element
6. We start by matching the first elements of both lists and getting the bindings
7. We then check what the rest of the first and second lists looks like
8. If the rest of both lists is a singleton, match the two remaining values
9. If the pattern list is a singleton, but the list we are matching against is not, we match the final element with the rest of the list
10. If neither of the lists is a singleton, we match the remaining elements recursively
11. We then return the rest of the bindings appended to the first bindings
12. A list can match an HTTP request or response. First we convert a the request/response to a list of the form given at 7.1 dependent on which it is, and then we match against that list
13. If we receive any other pattern and term combination, we throw an error

[9] We can see examples of this in this very piece of code. In line 1 for example, we match against TVar name _, so we still match against the last element, but we do not use it in this function and so do not care to give it a name

We end this section with a couple more examples of how one might use pattern matching in phi

A rather convoluted example would be

```
let ls = list(1,pair(2,list(3,4,5)),6) in
    let list(_1,pair(_2,list(_3,x,_5)),_6) = ls in
        out(stdout,x)
```

which outputs

```
    % phi eg.pi
    4
```

This more complex example describes a function takes an HTTP channel and an HTTP request, sends out the request, receives in a response ”resp”, and matches against it. If the response code is 302 (Moved) then we get the new location from the headers, make a new HTTP request, and recursively call the function. Otherwise we print out the body of the response.

```
let follow(ch,r) = (out(ch,r);in(ch,resp:HttpResponse);
    let list(c,_,_,b) = resp in
        if c = 302
            then let req = httpReq(getHeader("location",resp),headers(),httpGet()) in
                &follow(ch,req)
            else out(stdout,b))
```

8 Evaluation Strategy and Bringing it All Together

8.1 Evaluation Strategy Influence

Having never written an interpreter before, we did some research as to how one might go about doing so. We quickly found Jonathan Tang's Wikibook ”Write Yourself a Scheme” [Jon14], which breaks down the construction of an interpreter for the Scheme Lisp dialect. While obviously not entirely translatable to our own project, we undertook this particular tutorial, and found the exercise a very helpful basis for creating an interpreter of our own.

8.2 Main module

The main module of our program is the PiCalculus module. This module essentially acts to join together the previously mentioned modules and adds some further functionality. There are three main functions of this module

1. Handling user input and output
2. Evaluation of our Data structures including
   (a) Handling the Environment, which is to say variable assignment
   (b) Handling function application
   (c) Performing pattern matching
3. Handling information received from Channels appropriately

8.3 User Input and Output

In this section we describe how input from the user is brought into our evaluation strategy, and then information is handed back to the user. Many of the ideas here are inspired by [Jon14].

8.3.1 Running the program

```haskell
main :: IO ()
main = do
        name   <- getProgName
        args   <- getArgs
        case args of
            []  -> runRepl coreBindings
            [x] -> readFile x >>= runProcess coreBindings
            _   -> do
                    putStrLn           "Use:"
                    putStrLn $ name ++ " -- Enter the REPL"
                    putStrLn $ name ++ " [process] -- Run single process"
```

Our main function, which is called every time we type "phi" at the command line. When given one argument, we get the contents of the file with that name and then call runProcess on the returned contents.

```
runProcess :: IO Env -> String -> IO ()
runProcess core expr = core >>= flip evalAndPrint expr
```

runProcess accepts a set of core bindings and then calls eval and print on the string in its second argument

```
evalAndPrint :: Env -> String -> IO ()
evalAndPrint env expr = do
            res <- evalString env expr
            case res of
                "()"  -> return ()
                _     -> putStrLn res
```

evalAndPrint evaluates a string in a given environment. The case clause here is to avoid "()" being printed every time we evaluate a string to Void[10]

```
1 evalString :: Env -> String -> IO String
2 evalString env expr =
3         runIOThrows $
4         liftM show $
5         liftThrows (readProcess expr) >>= eval env
```

evalString is best understood if we break it down line by line. In line 5, liftThrows lifts its argument from ThrowsError into IOThrowsError, so the result of readProcess expr (so the result of parsing the string we pass into readProcess) is lifted. This lifted value is then passed into eval env where it is evaluated in the given environment using eval 8.4. Once this has been evaluated, we map show over the result of evaluating our parsed expression, and then call runIOThrows on that result, meaning we are now calling runIOThrows on a value of type IOThrowsError String.

```
runIOThrows :: IOThrowsError String -> IO String
runIOThrows action = liftM extractValue
                     (runExceptT (trapError action))

```

[10] ()

```haskell
trapError :: IOThrowsError String -> IOThrowsError String
trapError action = catchE action (return . show)
```

runIOThrows first calls trapError on the action passed in, which catches an exception thrown in the action and returns it.Then we call runExceptT which has type

```haskell
runExceptT :: e m a -> m (Either e a)
```

which for us means

```haskell
runExceptT :: PiError IO String -> IO (Either PiError String)
```

Then we lift a call to extractValue to retrieve the value from the Either PiError String However, because we have already caught any potential exceptions and returned them in the IOThrowsError monad, we will never encounter a Left in this Either, so we are safe to put a call to error in this function.

```haskell
extractValue :: ThrowsError a -> a
extractValue (Right v) = v
extractValue (Left  e) = error (show e)
```

So, backtracking, this now gives us a value of type IO String, which is exactly what we want.

If however we are passed no arguments, we enter the Read-Eval-Print Loop, where the only function we have not already met is until_, which simply keeps performing a monadic action until(!) a condition is met

```haskell
runRepl :: IO Env -> IO ()
runRepl core = core >>= until_ quit
               (readPrompt "phi>") . evalAndPrint
        where
            quit = flip any [":quit",":q"] . (==)
```

8.4 Evaluation of Data Structures

The two main functions in evaluation are

```haskell
eval :: Env -> PiProcess -> IOThrowsError ()
evalTerm :: Env -> Term -> IOThrowsError Value
```

each of which gets passed the Environment IORef at each call, and a PiProcess, Term, or Condition respectively.

8.4.1 Environment

There are three functions for manipulating the environment

getVar reads the environment IORef, then looks up the given variable name in the environment map, throwing an error if it does not exist.

defineVar reads the environment IORef, then inserts the value into the map with the given name. If the name already exists it is overwritten. We then write the new environment to the original IORef

bindVars reads the environment IORef, then creates a union between the current environment bindings and those in the associative list passed in. Instead of writing over the original IORef, we create a new one which is returned from this function. This allows us to create function closures, and "let..in.." bindings easily.

```
getVar :: Env -> String -> IOThrowsError Value
getVar envRef var = do
        env <- liftIO $ readIORef envRef
        maybe (throwE $ UnboundVar "Getting an unbound variable" var)
              return
              (Map.lookup var env)
defineVar :: Env -> String -> Value -> IOThrowsError ()
defineVar envRef var val = liftIO $ do
         env      <- readIORef envRef
         writeIORef envRef $ Map.insert var val env
bindVars :: Env -> [(String , Value)] -> IO Env
bindVars envRef bindings = do
                env <- readIORef envRef
                newIORef (Map.union (Map.fromList bindings) env)
```

8.4.2 eval

eval is the function which recurses through the whole structure of our processes, evaluating as it goes along. Here we go into the most interesting cases:

```
eval _ Null = return ()
```

Evaluating Null does nothing, as we would hope.

```
eval env (In a v@(TVar b t)) = do
    chan <- evalChan env a
    term <- receiveIn chan t
    bindings <- case term of
        TFun "<chan>" ex -> do
                ch <- decodeChannel ex
                return [(b,ch)]
        _ -> liftThrows $ match v term
    mapM_ (uncurry (defineVar env)) bindings
    return ()
        where
        decodeChannel e = do
            extraStrings <-  mapM extractString e
            case getChannelData extraStrings of
                Just (h,p)  -> liftM Chan $ liftIO $ newChan Connect h p
                Nothing -> throwE $ Default "incomplete data in channel"
```

Receiving a message on a channel involves first evaluating the channel. We then pass the channel and the Maybe Type of the variable to our receiveIn function ??. We then have a small section of code for our Channel serialisation hack 5.1.1, and then we attempt to match the original variable against the term we received.[11] We then monadically map defineVar over our bindings to add them to the environment.

```
eval env (Out a b) = do
                chan <- evalChan env a
                bVal <- evalTerm env b
                sendOut chan bVal
                return ()
```

When we send out a message on a channel, first we evaluate the channel in our environment, and then our message, calling sendOut. We will discuss this more in section 8.5

```
eval env (Conc procs)  = do
                var <- liftIO newEmptyMVar
                mapM_ (forkProcess var) procs
                res <- liftIO $ takeMVar var
                case res of
                    Left err -> throwE err
```

[11] The keen eyed will notice that this will always return a single binding, as we are passing a TVar as the first argument to match. The reason we have it implemented like this is that if in future we figure out how to match on an in without passing in the type of the data we expect as the type of a variable (see section ??) it won't require that much of a reshuffle

```
                    Right _  -> return ()
        where
            forkProcess var proc = liftIO $ forkIO $ do
                        res <- runExceptT (eval env proc)
                        _ <- tryPutMVar var res
                        return ()
```

When we evaluate a list of processes concurrently, we first of all create an MVar in which we will store the result of the processes. Then we monadically map forkProcess over the list. forkProcess calls forkIO on each element, and then evaluates each process in a new thread, placing its result in the MVar passed in the call to forkProcess. As it stands, the MVar only contains the result of the first process to finish executing, so if an error were to occure in a longer running process, it would not raise an exception. This is something we would address in future.

```
eval env (Replicate proc) =
    liftIO (threadDelay 100000) >> eval env (Conc [proc, Replicate proc])
```

Replicated processes are evaluated using Conc. There is a delay added because without it we quickly run out of memory. GHC threads may be lightweight, but nothing lightweight enough to spawn ad infinitum.

```
eval env (p1 `Seq` p2) = do
                eval env p1
                eval env p2
```

Evaluating two processes sequentially is handled as one would expect. We evaluate the first and then the second.

New and If are handled exactly as one would imagine. New simply reserves a name in the Environment, and If evaluates the condition and performs the first process if it evaluates to true and the second process if it evaluates to false.

Function Definition Let processes are overloaded with several purposes. We will discuss their use in pattern matching in 8.4.5 One of these is function definition which is achieved using the following three functions:

```
defineGlobalFun :: Env -> String -> [Term] ->
                    Value -> IOThrowsError ()
defineGlobalFun env name args term =
    defineVar env name $ makeFun args term env
```

```

defineLocalFun :: Env -> String -> [Term] ->
                    Value -> PiProcess -> IOThrowsError ()
defineLocalFun env name args term p = do
        clos <- liftIO $ bindVars env [(name, makeFun args term env)]
        eval clos p

makeFun :: [Term] -> Value -> Env -> Value
makeFun args = Func (map show args)
```

the first two being wrappers (for defining a function locally to the proceeding process or globally) around the third. Function definition is simply the binding of a list of parameters to a Value and, which could be a PiProcess or a Term, and a closure.

Atoms Atoms have a few special cases. The special funcion ”load” loads in a file full of let definitions separated by newlines, The special file ”pilude.pi”[12] is available to load at any time when using phi. The special function env can be used to outpute the current values in the environment. In any other case, we evaluate the given term as a process.

```
eval env (Atom (TFun "load" [TStr "pilude.pi"])) = do
    pilude <- liftIO $ getDataFileName "pilude.pi"
    eval env (Atom (TFun "load" [TStr pilude]))
eval env (Atom (TFun "load" [TStr file])) = do
    procs <- load file
    eval env $ foldl Seq Null procs
eval env (Atom (TVar "env" Nothing)) = do
    e <- liftIO $ readIORef env
    liftIO $ mapM_ (\ (k,v) -> putStrLn $ k ++ ": " ++ show v) $ Map.toAscList e
eval env (Atom p@(TFun{})) = void $ evalProcess env p
eval env (Atom p) = do
    proc <- evalProcess env p
    eval env proc
```

8.4.3 evalTerm

evalTerm is the function we use to evaluate Terms into Values

For TNums, TStrs, TBools, TBSs and TData which are already data in themselves, this is simply a case of passing them to the Term constructor. For other types of Term it is slightly more complicated.

[12] A play on the Haskell Prelude

```
evalTerm env (TVar name _) = getVar env name
```

when we evaluate a bare variable, we retrieve it's value from the environment

```
evalTerm env   (TList ls) = do
    vs <- mapM (evalTerm env) ls
    ts <- extractTerms vs
    return $ Term $ TList ts
evalTerm env (TPair (t1,t2)) = do
    a <- evalTerm env t1
    b <- evalTerm env t2
    case (a,b) of
        (Term c, Term d) -> return $ Term $ TPair (c,d)
        _                -> throwE (Default "pair not given two terms")
```

with lists and pairs we must recursively apply evalTerm to each element to fully evaluate the structure. However because evalTerm returns IOThrowsError Value, we must also check that each item returned into these structures is still a Term. extractTerms does this over a list (throwing an error if it comes across a non-term value), but with pairs we do this manually, as there are only two elements.

The only remaining kind of term is the TFun. There are a few special cases we cater for, which are the channel building functions, namely:

```
evalTerm env (TFun "anonChan" []) = do
    port <- assignFreePort env
    liftM Chan $ liftIO $ newChan Init "localhost" port
evalTerm env (TFun "anonChan" [n]) = do
    port <- evalToInt env n
    c <- liftIO $ newChan Init "localhost" port
    return $ Chan c
```

As we saw in section 4.2 an anonChan can be entered by the user using either the full name, or the shortcut syntax (a pair of curly braces, either with a number in between or empty). This is how we create the server end of a channel in phi. An anonChan with a number as an argument[13] creates the server end of a channel at the specified port. An empty anonChan

[13] This is a bit of a misnomer in this instance, but the name stuck around by association with the second usage

assigns the channel a free port[14] This might seem slightly bizarre at first - how is someone supposed to connect to a port whose number one does not know. The included sample program C.2 demonstrate how one might use such functionality.

```haskell
evalTerm env (TFun "httpChan" [a]) = do
    host <- evalToString env a
    liftM Chan $ liftIO $ newChan Connect host 80
evalTerm env (TFun "chan" [a,n]) = do
    host <- evalToString env a
    port <- evalToInt env n
    liftM Chan $ liftIO $ newChan Connect host port
```

These two functions connect to a channel at a remote location. httpChan connects to port 80 at the given hostname, and chan can connect to any given host on any given port.

The last case of our evalTerm function is:

```haskell
evalTerm env (TFun name args) = do
    fun <- getVar env name
    argVals <- mapM (evalTerm env) args
    apply fun argVals
```

which fetches the function from the environment, then monadically maps evalTerm over the arguments, and finally applies the function, which we describe below.

8.4.4 Function Application

As we saw in section 4.1.4 there are two kinds of function both of which are dealt with in our implementation by

```haskell
apply :: Value -> [Value] -> IOThrowsError Value
```

In the case of a primitive function, which acts on Terms only, we extract the terms from this list of Values passed in using extractTerms again, and then apply the function[15] , lift its result into the IOThrowsError monad and return it.

[14] This is currently also a "hack". There is a variable in the environment with a name that the parser cannot parse, and which we use to refer to the next free ephemeral port. Again, currently this starts at the lowest possible ephemeral port (2$^{[}$15]+2$^{[}$14]) and counts upwards, but it would not be too much of a stretch to have it assign a non-allocated one in the range.

[15] which is of type [Term] → ThrowsError Term

```
apply (PrimitiveFunc fun) args = do
        ts <- extractTerms args
        res <- liftThrows $ fun ts
        return $ Term res
```

For a user-defined function our task is a little more difficult. We first check if the function has been passed the correct number of arguments, then we zip the list of Values passed in with the list of parameters that Func contains, and bind those values in the closure of the function. If the function body is a Term, then we simply evaluate it in the closure, if it is a Process then we must first evaluate it and then return they body (as eval has a return type of ())

```
apply (Func parms bdy closre) args =
    if num parms /= num args
        then throwE $ NumArgs "user-defined" (num parms) args
        else do
             clos <- liftIO (bindVars closre $ zip parms args)
             case bdy of
                Term t -> evalTerm clos t
                Proc p -> eval clos p >> return bdy
                _      -> throwE (Default "this function makes no sense")
    where
        num = toInteger . length
```

8.4.5 Pattern Matching

Here we show how we use our pattern matching utility.

There are two cases here to cater for ”let..” (global) and ”let..in..” (local) bindings.

```
eval env (Let t1 (Term t2) Nothing) = do
    val <- evalTerm env t2
    case val of
        Term term -> do
            bindings <- liftThrows (match t1 term)
            mapM_ (uncurry (defineVar env)) bindings
            _          -> throwE (Default "Can only pattern match against Terms")
eval env (Let t1 (Term t2) (Just p)) = do
    val <- evalTerm env t2
    case val of
        Term term -> do
```

```
            bindings <- liftThrows $ match t1 term
            newEnv <- liftIO $ bindVars env bindings
            eval newEnv p
            _         -> throwE (Default "Can only pattern match against Terms")
```

In the first case we evaluate the term to the right of the "=", assert that it is still a term, get the bindings from our match function, and monadically map defineVar over the bindings, and in the second case we do exactly the same but call bindVars to create a new local environment.

8.5 Data from Channels

The last thing our Main module needs to do is handle data going into and coming out of channels.

```
sendOut :: Channel -> Value -> IOThrowsError ()
sendOut chan v@(Chan c) = if serialisable c
                    then liftIO $ send chan $ show v
                    else throwE $ Default "Channel not serialisable"
sendOut chan val = liftIO $ send chan (show val)
```

Sending out on channels is simple. We first check if the thing we are trying to send is a channel, and if it is, if that channel is serialisable. If not, we throw an error, in every other case we simply convert the object into a string using show and then send it out onto the channel.

```
receiveIn :: Channel -> Maybe Type -> IOThrowsError Term
receiveIn chan t = do
        str <- liftIO $ receive chan
        case t of
            Just HttpRequest  -> makeHttpRequest str
            Just HttpResponse -> makeHttpResponse str
            _                 -> liftThrows $ readTerm str
```

Receiving messages onto a channel is more complicated. We first need to check what type of message we are expecting. The reason for this is that the functions for parsing HTTP Requests and Responses are completely different, and so the only way we could think of to ascertain which one to use was to leave it up to the programmer. So, the programmer must pass in the type of the variable we are receiving into. We then match that type against HttpRequest or HttpResponse, calling the appropriate parsing function, and if it is neither, we assume it is a Term from another instance of phi.

9 Project Evaluation

When we set out on this project, we said that there were 6 things we would like to achieve, in increasing order of difficulty.

1. Basic interaction with controlled environment
2. Moderate coverage of language by compiler (in, out, parallel composition, sequential composition, limited replication, basic types and functions)
3. Models for a few more basic protocols, such as a naive handshake protocol
4. Responsive concurrency and fast interpretation
5. Complete language compilation (pattern matching, de-structuring, user defined functions and types)
6. Fully interoperable compiled models

And that we would consider the project a success if we were to achieve the first 3 of these fully, and at least one of the following three partially. As the current implementation stands, we have actually achieved 5/6 of our desired goals, but not the ones we thought we would. We had achieved items 1,2,and 5 within the first month and a bit of full-time work on the project. Item 4 was almost a given through careful design of our program. The most difficult was in fact item 3, and subsequently item 6. We did, however eventually manage to implement the handshake protocol in phi, which is in appendix C.3. Where we have failed is in our aims to really test our implementation in real life protocols. This is mostly a result of our poor implementation of cryptographic functions. It is just not possible to model real-world protocols without them, and this is a serious oversight on our part. However, having said that, the protocol definitions we have seen tend to treat all their encryption and hashing functions as impenetrable black boxes, which is partly what makes them so hard to implement. We have been able to build a very responsive interpreter for the complete language, which is built well enough that it would not be impossible to fix the issues we have eventually. Overall, we would say this project has been successful. The program is not without its issues, and we did not achieve all our goals, but it is definitely a tool which could be very useful once we've worked out a few of the kinks.

10 Further Extensions

If we were to continue with this project there would be several things we would fix before we started extending the program. First, and most importantly, we would completely redesign the cryptographic functionality of the system. We would look at various different options, either adopting the Unsafe route, or perhaps moving them into their own module with the ability to access the IO monad. Which ever way we go about it, it is a high priority, as the language is basically neutered with respect to modelling security protocols if it has poor cryptographic functionality.

Secondly, there is currently an issue whereby one cannot pass a process into a function directly, one must first create a let binding to the process, and then pass it to the function. This is not game changing, but it would help with clarity when defining recursive functions.

Thirdly, it would be nice to play around with the idea of non-standard channels (that is to say channels that generate random data, or channels connected to databases directly). Random channels could certainly help us cryptographically: we could use them to generate nonces for example.

We feel that this is a project with a lot of scope for growth in the future. Lots of small extensions, such as adding support for HTTPS, could improve it dramatically.

Ultimately, we would like to see the program become a tool to bridge the gap between protocol and implementation, which we believe it has the potential to be some day.

A Program Use

A.1 Installation

The source is available on Hackage, and can be installed using cabal:

```
cabal update
cabal install pi-calculus
```

Alternatively you can clone the source and build using cabal:

```
git clone git@github:renzyq19/pi-calculus
cd pi-calculus/pi
cabal install
```

A.2 Running

To run the interpreter, we can either enter the Read Eval Print Loop (REPL) by entering typing phi at the command line:

```
$ phi
phi>
```

or we can interpret a single file containing a process as follows:

```
$ phi file.pi
```

A.3 Language Syntax

The simplest processes in the language are

```
in(chan,msg)
```

which receives a message on the channel chan and binds it to the variable msg for subsequent processes

and

```
out(chan,msg)
```

which sends the value of variable msg across channel chan.

The most basic channels are the wrappers for stdin, stdout and stderr, all of which have the same names. So, the ubiquitous "Hello World!" program is written in pi-calculus as:

```
out(stdout,"Hello World!")
```

These can be combined sequentially using `;`, so a more complex example would be:

```
out(stdout,"What is your name?");
in(stdin,name);
out(stdout,append("Hello ",name))
```

We can define our own variables and functions using:

```
let variable = value in process
```

which binds variable to value locally in process

```
let function(..) = body in process
```

which binds the function(..) to be body locally in process

So to rewrite our second example: [16]

```
out(stdout,"What is your name?");
in(stdin,name);
let prependHello(a) = append("Hello ",a) in
    let helloName = prependHello(name) in
        out(stdout,helloName)
```

or, if we are in the REPL (or a module file, which we will cover momentarily) we can define variables and functions globally by omitting the ”in process” i.e.

```
let triple(a,b,c) = pair(a,pair(b,c))
```

If we have a file of definitions like the one above, we can load them into the current environment using the atom

```
&load("filename.pi")
```

This introduces us to the concept of atoms. An atom is a function which represents a process. We can define them just as we can define functions over terms:

```
let forever(proc) = &proc;&forever(proc)
let print(msg)    = out(stdout,msg)
```

and if we were to load this file into the REPL, we could call the functions like so:

```
phi> &load("defs.pi")
phi> let p = &print(10)
phi> &forever(&p)
10
10
10
◦ .
```

You can compose components concurrently using ’—’, which does not have spaces around it.

```
phi> out(localnet,"hello")|in(localnet,msg);out(stdout,msg)
"hello"
```

You can reserve variable names using the ”new” process:

[16] N.B. Any whitespace over a single space around ”in” is ignored, so the indentation above is optional, but nicer to read

```
phi>out(stdout,s)
Getting an undefined variable: s
phi>new s
phi>out(stdout,s)
s
```

You can use conditionals like so:

```
phi>if a = true then out(stdout,a) else out(stdout,b)
```

where the else clause is optional

Finally we can execute an infinite number of process P concurrently by calling

```
phi> !(&P)
```

B Full Parser Implementation

```
module Parser (
        readTerm,
        readProcess,
        readProcesses)
        where

import Control.Monad (liftM)
import Control.Monad.Error (throwError)
import Text.ParserCombinators.Parsec

import TypDefs

parseNull :: Parser PiProcess
parseNull = char ’0’ >> return Null <?> "parse null"

parseIn :: Parser PiProcess
parseIn = do
            _ <- string "in("
            name <- parseTerm
            paddedComma
            var  <- parseTerm
            _ <- char ’)’
            parseSeq $ In name var
            <?> "parse in"

parseOut :: Parser PiProcess
parseOut = do
            _ <- string "out("
            name <- parseTerm
            paddedComma
            term  <- parseTerm
            _ <- char ’)’
            parseSeq $ Out name term
            <?> "parse out"

parseReplicate :: Parser PiProcess
parseReplicate = do
            _ <- string "!("
            process <- parseProcess
            _ <- char ’)’
            return $ Replicate process
            <?> "parse replicate"

myOption :: Show b => a -> Parser a -> Parser b -> Parser a
myOption opt parser sep = try (notFollowedBy sep >> return opt) <|> (sep >> parser)

myOptionMaybe :: Show b => Parser a -> Parser b -> Parser (Maybe a)
myOptionMaybe parser = myOption Nothing (liftM Just parser)

parseSeq :: PiProcess -> Parser PiProcess
parseSeq p1 = do
            p2 <- myOption Null parseProcess (paddedChar ’;’)
            return $ p1 ‘Seq‘ p2

parseNew :: Parser PiProcess
parseNew = do
            _ <- string "new"
            spaces
            name <- parseTerm
            parseSeq $ New name
            <?> "parse new"

parseIf :: Parser PiProcess
parseIf = do
            _ <- string "if"
            spaces
            cond <- parseCondition
            paddedStr "then"
            p1 <- parseProcess
            p2 <- myOption Null parseProcess (paddedStr1 "else")
            return $ If cond p1 p2
            <?> "parse if"

parseLet :: Parser PiProcess
parseLet = do
            _ <- string "let"
            spaces
            name <- parseTerm
            paddedChar1 ’=’
            val <- try (liftM Proc parseProcess) <|> liftM Term parseTerm
            p <- myOptionMaybe parseProcess (paddedStr1 "in")
            return $ Let name val p
            <?> "parse let"

parseAtom :: Parser PiProcess
parseAtom = do
            _ <- char ’&’
            atom <- parseTerm
            parseSeq $ Atom atom

padded :: Parser a -> Parser ()
padded p = do
         spaces
         _ <- p
         spaces

paddedChar :: Char ->  Parser ()
paddedChar ch = padded $ char ch

paddedStr :: String -> Parser ()
paddedStr str = padded $ string str

paddedComma :: Parser ()
paddedComma = paddedChar ’,’

padded1 :: Parser a -> Parser ()
padded1 p = do
         skipMany1 space
         _ <- p
         skipMany1 space

paddedChar1 :: Char ->  Parser ()
paddedChar1 ch = padded1 $ char ch

paddedStr1 :: String -> Parser ()
paddedStr1 str = padded1 $ string str

parseCondition :: Parser Condition
parseCondition = do
            t1 <- parseTerm
            paddedChar1 ’=’
            t2 <- parseTerm
            return $ t1 ‘Equals‘ t2

parseTVar :: Parser Term
parseTVar = do
        v <- readVar
        case v of
            "true"  -> return $ TBool True
            "false" -> return $ TBool False
            _       -> do
                t <- myOptionMaybe parseType (paddedChar ':')
                return $ TVar v t

parseTFun :: Parser Term
parseTFun = do
            name <- readVar
            spaces
            args <- bracketed $ sepBy parseTerm paddedComma
            return $ case (name,args) of
                ("pair", t1:t2:_)  -> TPair (t1,t2)
                ("list", _ )       -> TList args
                _                  -> TFun  name args

parseTStr :: Parser Term
parseTStr = do
        _ <- char '"'
        x <- many $ noneOf "\""
        _ <- char '"'
        return $ TStr x

parseTNum :: Parser Term
parseTNum = liftM (TNum . read) (many1 digit)

readVar :: Parser Name
readVar = do
        frst <- letter <|> symbol
        rest <- many $ letter <|> digit <|> symbol
        return $ frst:rest

symbol :: Parser Char
symbol = oneOf "'._<>"

parseTerm :: Parser Term
parseTerm =  try parseAnonChan
         <|> try parseTFun
         <|> parseTNum
         <|> parseTVar
         <|> parseTStr
         where
            parseAnonChan = do
                _ <- char '{'
                spaces
                arg <- many parseTerm
                spaces
                _ <- char '}'
                return $ TFun "anonChan" arg

parseProcesses :: Parser [PiProcess]
parseProcesses = sepEndBy parseProcess newline

parseProcess :: Parser PiProcess
parseProcess = liftM (\ps -> case ps of
                        [p] -> p
                        _   -> Conc ps) $ sepBy1 parseProcess' (char '|')
    where
    parseProcess'  = bracketed parseProcess'' <|> parseProcess''
    parseProcess'' = parseNull
                 <|> try parseIf
                 <|> parseIn
                 <|> parseOut
                 <|> parseLet
                 <|> parseReplicate
                 <|> parseNew
                 <|> parseAtom

parseType :: Parser Type
parseType =  try (str HttpRequest)
         <|> try (str HttpResponse)
         where
         str t = string (show t) >> return t


bracketed :: Parser a -> Parser a
bracketed = between (char '(') (char ')')

readOrThrow :: Parser a -> String -> String -> ThrowsError a
readOrThrow parser name input = case parse parser name input of
                        Left  err -> throwError $ Parser err
                        Right val -> return val

readProcess :: String -> ThrowsError PiProcess
readProcess = readOrThrow parseProcess "single process"

readProcesses :: String -> ThrowsError [PiProcess]
readProcesses = readOrThrow parseProcesses "multiple-processes"

```

```haskell
readTerm :: String -> ThrowsError Term
readTerm = readOrThrow parseTerm "term"
```

C Code Samples

C.1 Simple One-Shot Chat Server

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

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

C.2 Chat Server with Anonymous Channels

Here we create an anonymous channel, send it across to a remote phi, and then communicate along it.

```
let server =
    out(stdout,"Enter port:"); in(stdin,port);
    let ch = {port} in
        in(ch,sch);in(stdin,msg);out(sch,msg)

let client =
    out(stdout,"Enter host:"); in(stdin,host);
    out(stdout,"Enter port:"); in(stdin,port);
    let ch = chan(host,port) in
        let sch = {} in
            out(ch,sch);in(sch,msg);out(stdout,msg)
```

C.3 Handshake Protocol

Here is an implementation of the Handshake protocol written in phi. To run it, open the REPL, &load(”handshake.pi”, and you should see an endless string of s’s appearing

```
new s
let c = {9000}
let c’ = chan("localhost",9000)
let clientA(pkA,skA,pkB) =
    (out(c,pkA); in(c,x);
    let y = adec(skA,x) in
    let pair(pkD,j) = getmsg(y) in if pkD = pkB then let k = j in
    out(c,senc(k,s)))
let serverB(pkB,skB) =
    in(c’,pkX);
    new k;
    out(c',aenc(pkX,sign(pkB,pair(pkB,k))));
    in(c',x);
    let z = sdec(k,x) in out(stdout,z)
let process =
    (new skA;
    new skB;
    let pkA = pk(skA) in
    let pkB = pk(skB) in
    !(&clientA(pkA,skA,pkB))|!(&serverB(pkB,skB)))
```

C.4 Following Redirects and Getting a Resource

Here we load the follow file, containing the follow function we saw earlier and demonstrate its use.

```
let follow(ch,r) = (out(ch,r);in(ch,resp:HttpResponse);
    let list(c,_,h,b) = resp in
        if c = 302
            then let req = httpReq(getHeader("location",resp),headers(),httpGet()) in &follo
            else &print(b))

&load("pilude.pi");
&load("follow.pi");
&print("Host:");in(stdin,site);
&print("Resource:");in(stdin,res);
let siteChan = httpChan(site) in
    let req = httpReq(uri(site,res),headers(),httpGet()) in &follow(siteChan,req)
```

References

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL'01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115. ACM Press, 2001.

[Con14a] Control.Concurrent.Chan. Control.Concurrent.Chan – Hackage, 2014. [Online; accessed 13-June-2014].

[Con14b] Control.Concurrent.MVar. Control.Concurrent.MVar – Hackage, 2014. [Online; accessed 13-June-2014].

[Con14c] Control.Monad.Either. Control.Monad.Either – Hackage, 2014. [Online; accessed 12-June-2014].

[Con14d] Control.Monad.Error. Control.Monad.Error – Hackage, 2014. [Online; accessed 12-June-2014].

[Con14e] Control.Monad.Trans.Except. Control.Monad.Trans.Except – Hackage, 2014. [Online; accessed 12-June-2014].

[Dat14] Data.IORef. Data.IORef – Hackage, 2014. [Online; accessed 12-June-2014].

[HTT14a] HTTP.Base. Network.HTTP.Base – Hackage, 2014. [Online; accessed 10-June-2014].

[HTT14b] HTTP.Headers. Network.HTTP.Headers – Hackage, 2014. [Online; accessed 10-June-2014].

[IO14] IO. System.IO – Hackage, 2014. [Online; accessed 10-June-2014].

[Jon14] Jonathan Tang. Write Yourself a Scheme in 48 hours – Wikibooks, 2014. [Online; accessed 20-January-2014].

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mon14] MonadIO. Control.Monad.IO.Class – Hackage, 2014. [Online; accessed 10-June-2014].

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts i and ii). *Information and Computation*, 100:1–77, 1992.

[Net14a] Network. Network – Hackage, 2014. [Online; accessed 10-June-2014].

[Net14b] Network Sockets. Network sockets — Wikipedia, the free encyclopedia, 2014. [Online; accessed 10-June-2014].

[Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International (UK) Ltd, Campus 400, Maylands Avenue, Hemel Hempstead, Hertfordshire, HP2 7EZ, 1987.

[Pro14] Process Calculus. Process calculus — Wikipedia, the free encyclopedia, 2014. [Online; accessed 20-February-2014].

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Indiana University, 1997.

[RS13] Mark D. Ryan and Ben Smyth. Applied pi calculus, 2013. 2013 revision.

[Uns14] Unsafe. System.IO.Unsafe – Hackage, 2014. [Online; accessed 10-June-2014].