Imperial College London

3RD YEAR COMPUTING

INTERIM REPORT

Implementation of a New Web Language

Author:
William DE RENZY-MARTIN

Supervisor:
Sergio MAFFEIS

February 21, 2014

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 a compiler 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 $\bar{x}\langle y\rangle.P|x\,(v)\,.Q$ 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 $P\backslash\,[x]$.

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 Compilation

Trying to compile a process based language presents several difficulties from the offset. Such a compiler 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 compiler 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
$if M = N then P else Q$	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
$A \mid B$	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.

$P \mid Q$	`P | Q`	parallel composition
$P.Q$	`P ; Q`	sequential composition
$!P$	`!P`	replication
vn	`new x`	new
$if M = N then P else Q$	`if p(M) then P else Q`	conditional
$u(x)$	`in(u,x)`	input
$\bar{u}\langle N \rangle.P$	`out(u,N)`	output
$\left[\frac{M}{x}\right]$	`let X = M in P`	active substitution (i.e. pattern matching)

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

2.4.3 Starting Restrictions

The first build of our compiler 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 WebSockets [Web14]. The reasoning here is that attempting a completely generic channel would involve implementing a huge range of different server and client-like processes, which is not feasible. By starting with WebSockets, we can familiarise ourselves with some of the concepts of channels and cross-process communication, and later on we can expand to more generic sockets once we are sure 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 a compiler.

2.5.1 Abstract Data Type

Haskell makes it trivial to create abstract 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.

module *PiProcess* **where**

import *Data.List*
data *PiProcess* =

Our abstract data type

In Name Variable |

Wait for message in on Channel and assign it to Variable

Out Name Term |

Send out Message on Channel

PiProcess ‘*Conc*‘ *PiProcess* |

Perform Processes concurrently

PiProcess ‘*Seq*‘ *PiProcess* |

Perform Processes sequentially

Repl PiProcess |

Replicate process

If Condition PiProcess PiProcess |

Conditional selection of process

New Name |

Creation of new / reserved name in proceeding processes.

Let Name Term PiProcess

Pattern match name with a term in process

deriving (*Eq*)

We can define our terms as follows:

data *Term* = *TName Name* |

Plain name

TVar Variable |

Plain variable (we keep the distinction for now to be in keeping with [AF01] and [RS13])

TFun Name [*Term*] *Int*

Functions over a list of terms, holding information about their arity

deriving (*Eq*)

To begin with, for the purposes of getting a basic parser up and running, we will simply have Variabls and Names be type synonyms for String.

type *Variable* = *String*
type *Name* = *String*

This will change in future, as both will eventually need to contain some type information. This should be achievable by making them data types of their own.

Condition currently only holds information about two terms being equal

data *Condition* = *Term* ‘*Equals*‘ *Term* **deriving** (*Eq*, *Show*)

This may change

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 ++ "," ++ show m ++ ")"
  show (Repl proc) = "!(" ++ show proc ++ ")"
  show (p1 `Conc` p2) = show p1 ++ "|\n" ++ show p2
  show (p1 `Seq` p2) = show p1 ++ ";\n" ++ show p2
  show (New n) = "new " ++ n
  show (If c p1 p2) = "if " ++ show c ++ " then " ++ show p1 ++ " else " ++ show p2
  show (Let n t p2) = "let " ++ n ++ " = " ++ show t ++ " in\n" ++ show p2


instance Show Term where
  show (TVar x) = x
  show (TName n) = n
  show (TFun n [] 0) = n
  show (TFun n ts _) = n ++ "(" ++ (concat (intersperse "," (map show ts))) ++ ")"
```

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:

```
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
    ghci>parseTest inOut "out (a,b)"
    parse error at (line 1, column 4):
    unexpected " "
    expecting "("
```

In the last case it is interesting to note that whitespace is not handled very well. This can be solved by first tokenising the input. The ParsecToken module can be used to generate a tokeniser.

2.5.3 WebSockets

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 WebSockets for our initial implementation.

Here we can demonstrate some basic WebSockets functionality. We can use the Haskell WebSockets module in the following way:

```
import Control.Monad (forever,unless)
import Control.Concurrent (forkIO)
import Control.Monad.IO.Class (liftIO)
import qualified Data.Text as T
import qualified Data.Text.IO as T
import qualified Network.WebSockets as WS


-- SERVER Process Code
server :: IO ()
server = do
    WS.runServer "0.0.0.0" 9000 $ pong

pong :: WS.ServerApp
pong pending = do
    conn ← WS.acceptRequest pending
    forever $ do
        msg ← WS.receiveData conn
        WS.sendTextData conn (msg :: T.Text)
```

```
-- CLIENT Process Code
client :: IO ()
client = WS.runClient "0.0.0.0" 9000 "/" ping

ping :: WS.ClientApp ()
ping conn = do
    _ ← forkIO $ forever $ do
        msg ← WS.receiveData conn
        liftIO $ T.putStrLn msg
    let loop = do
            line ← T.getLine
            unless (T.null line) $WS.sendTextData conn line
            loop
    loop

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

This small program creates two processes. 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 Plan

Currently, we have completed most of our background research, and have started building a small web server in Haskell. The purpose of the latter is firstly to learn more about web frameworks in general, and secondly, once we have built a basic implementation, we can start to see how easy it will be to interact with conventional servers.

3.1 Milestone Dates

There are roughly 16 weeks between now and 17^{th} June, the preliminary archive submission deadline. We therefore aim to split that time into eight periods of two weeks, and at the end of each of those periods we would like to achieve the following:

3.1.1 7^{th} March

By this stage we would ideally have a small framework set up against which we could start testing some of the basic features of the language. We will start to write up the implementation details for this framework. Basic interaction with small servers i.e. `in(a,message);out(a,message)`

3.1.2 21^{st} March

We would like to be able to compile a subset of the language, definitely `in, out, !, |, if then else` and be able to test some basic interactions. If this is not possible at this stage, we may have to rethink the restrictions we have set for the language. Continue to write up implementation details for whatever we have achieved so far.

3.1.3 4^{th} April

Ideally we should be able to model and execute a basic handshake protocol by this stage, this will have required the addition of `new` and `let` to our compile-able subset, and also a few functions. Continue to add any further implementation details.

3.1.4 18^{th} April

If all goals met so far, begin trying to speed up compilation times and responsiveness of compiled programs. If not then work out what is causing difficulties, maybe rethink strategy/implementation. Write up anything relevant on either changing the implementation or increasing responsiveness.

3.1.5 2^{nd} May

Hopefully have an acceptably responsive (comparable to similar implementations of models in existing languages) model. Start polishing the compiler and working towards complete language compilation. Begin writing evaluation, by this stage how much is possible in the time left will be very clear.

3.1.6 16^{th} May

If the compiler is not yet complete continue working on it. If it is, then continue work on making the implementation more responsive.

3.1.7 19^{th} - 23^{rd} May : Health Check-Up

3.1.8 30^{th} May

At the very least 70% of the report should be done, and any remaining parts should have a clear structure laid out. Continue tinkering if necessary, if not then keep writing the report.

3.1.9 13^{th} June

Address any final issues with the compiler. If in working order, attempt some complex models. Assess any major issues faced during the course of the project Finish evaluation and conclusion

3.1.10 17^{th} June - Project submission deadline

3.1.11 23^{rd} June - Preliminary Archive Submission Deadline

3.1.12 30^{th} June - Final Project Archive Submission Deadline

4 Evaluation

There are 6 points of success we would like to achieve over the course of this project. In increasing order of difficulty they are:

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 compilation
5. Complete language compilation (pattern matching, de-structuring, user defined functions and types)
6. Fully interoperable compiled models

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.

Overall the aim of this project is to create something which is easy to use, and which can interact with a number of different processes and systems. As we have demonstrated with some of our background research, interaction on a single kind of channel seems feasible, and this makes us optimistic for final product which can achieve our goal.

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.

[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.

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

[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.

[Web14] WebSockets. Websockets— Wikipedia, the free encyclopedia, 2014. [Online; accessed 20-February-2014].