

The π -calculus and the Pict Programming Language

Owen McGrath

owen.mcgrath@uky.edu

Jan 24

Background: Turing Machines

Turing machines were one of the first mathematical descriptions of general computation.

Their method of operation could be described as a calculus, in that it provides a definition of calculation.



Background: Lambda Calculus

The lambda calculus also provides a background for computation, but describes computation as application of functions to other functions.

It is possible to derive everything in a programming language as an abstraction of the lambda calculus.

I will talk about the π -calculus, which is yet another calculus.

$$\begin{aligned} & (\lambda p. \lambda q. p p q) (\lambda a. \lambda b. a) (\lambda a. \lambda b. b) = (\lambda a. \lambda b. a) (\lambda a. \lambda b. b) \\ & (\lambda p. \lambda a. \lambda b. p b a) (\lambda a. \lambda b. a) = \lambda a. \lambda b. (\lambda a. \lambda b. a) \\ & (\lambda p. p (\lambda a. \lambda b. b) (\lambda a. \lambda b. a)) (\lambda a. \lambda b. a) = \end{aligned}$$

Example lambda expressions

Background: Imperative vs. Functional languages

Imperative languages : Turing Machines :: Functional Languages : Lambda calculus



```
int get_integer_argument(const char * command)
{
    const char *num = afterspace(command);
    //t will be advanced until a non-integer is reached
    const char *t = num;
    while (t[0] != '\0')
    {
        if (t[0] > '9' || t[0] < '0')
        {
            printf("Not numeric: ");
            printf(num);
            printf("\n");
            return -1;
        }
        t++;
    }

    int n = atoi(num);
    return n;
}
```



```
(defmacro defopcode (name (&key (docs "") raw (track-pc t)) modes &body body)
  "Define a Generic Function NAME with DOCS if provided and instructions,
  i.e. methods, via DEFINS for each addressing mode listed in MODES. If RAW is
  non-nil, MODE can be funcalled with a cpu in BODY to retrieve the byte at MODE's
  address. Otherwise, funcalling MODE will return the computed address itself."
  ` (progn
    (eval-when (:compile-toplevel :load-toplevel)
      ,@(loop for (op cycles bytes mode) in modes
        collect `(setf (aref *opcodes* ,op) '(,name ,cycles ,bytes ,mode))))
    (defgeneric ,name (,(intern "OPCODE") &key cpu
      ,(intern "MODE") ,(intern "SETF-FORM"))
      (:documentation ,docs))
    ,@(loop for mode in modes for mname = (fourth mode) with x = (intern "X")
      do (setf (fourth mode) (if raw
        `` ,mname
        `(lambda (cpu) (get-byte (,mname cpu))))))
    collect `(defins (,name ,@mode)
      (:setf-form (lambda (,x) (setf (,mname cpu) ,x))
        :track-pc ,track-pc
        ,@body)))))
```

Motivation

Turing Machines can help us model and understand computation.

⇒ This naturally leads to structured programming and many features found in C and Algol.

The Lambda calculus can help us understand functions and higher-order functions.

⇒ This leads to constructs that can be useful like easy recursion (via the fixed point combinator), lazy evaluation, and functions with no side effects.

Motivation

However, there are still some problems that neither functional nor imperative paradigms help with.

The major problem I will address here is **concurrency**.

Concurrency is hard: C++ example

`std::thread` - Only available since C++11

`std::atomic` - Hard / impossible to create atomic user-defined type

We also have `std::mutex`, `std::promise` and `std::future`...

Many new features in C++20.




These may work well, but they are not a natural language construct in the way functions or variables are.

How can we improve the situation?

Lambda calculus derives everything (numbers, typed values) as abstractions around functions, and this makes it easier to work with functions in programming languages.

Therefore, is it possible to define everything as an abstraction of concurrent communication to make concurrency easier?

The π -calculus attempts to do that.

	Turing Machines	Lambda Calculus	π -Calculus	...
Computation can be thought of as...	Symbol manipulation	Functions and function application	Concurrent Processes	...
This makes it easy to...	Model and understand computation	Reason about higher-order functions	Efficiently use and reason about concurrency?	...
Languages inspired	C++  Algol	Haskell  Lisp 	Pict	...

Introduction to the π -Calculus

- Everything is an agent or a link
- No Types

Agents can be thought of as processes, and links can allow communication between agents.

Introduction to the π -Calculus

Computation happens by separate concurrent agents communicating over links.

Links can be thought of as busses or pipes: data is sent in one end, and it is sent out the other.

I will discuss agent semantics first.

π -calculus: agents

$P \mid Q$ This agent executes P and Q simultaneously.

0 The zero agent does nothing.

We need a way to describe processes and how they interact.

We will use agents to be an abstraction of processes or threads, and we will have rules for executing them in parallel and for sending messages between them.

Agent examples

$$(0 \mid 0 \mid 0) \rightarrow (0 \mid 0) \rightarrow 0$$

Because the zero agent does nothing, we can eliminate it without changing the program.

This would be analogous to the C++ code on the right.

```
void zero() { }  
  
int main()  
{  
    std::thread a(zero), b(zero), c(zero);  
  
    a.join();  
    b.join();  
    c.join();  
}
```

Links between agents

Say we have agents P and Q running in parallel.

$$(P \mid Q)$$

As an example, say P is a thread that looks polls the internet for data, and Q is a thread that processes that data. In this example, we cannot have P finish before beginning Q . Therefore, we need some way to communicate between the threads.

Links between agents

$$(P \mid Q)$$

In our example, P must communicate information to Q . An important aspect of this communication is that the communication is one-way.

The π -calculus models this as a link from P to Q .

Because P sends information across the link, it has the input prefix.
 Q will have the output prefix.

We now need a way of handling links.

π -calculus: links

$\bar{x}(y).P$

This agent sends y along the link x and continues as the agent P .
This is known as the negative prefix or the input prefix.

$x(z).P$

This agent receives z along link x and continues as P , with the variable z replaced with what is received.
Also known as the positive prefix or output prefix.

In our example, P needs to give information to Q . This means that we will need a link between them. Let this link be x ; then, P will have the input prefix and Q will have the output prefix.

Links are also known as channels. Often, the π -calculus concerns links while the Pict language concerns channels.

Example: passing information between processes

$$(\bar{x}(y).P \mid x(z).Q)$$

This is two processes running in parallel, where x is a link between them. The left hand side of the composition (\mid) sends y along the link x . The right hand side receives anything on that link as z . Therefore, this can be reduced to

$$(P \mid Q\{y/z\})$$

The act of receiving y as argument z can be thought of as calling a function, and $Q\{y/z\}$ can be thought of as replacing the parameters with the arguments.

Example: passing information between processes

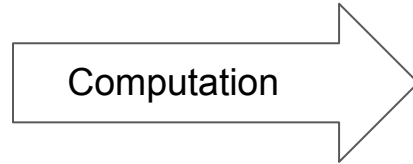
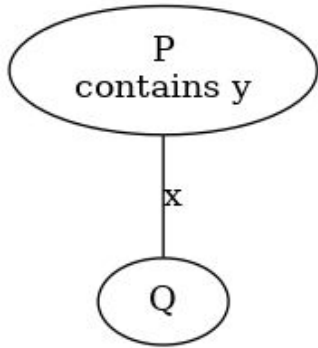
$$(\bar{x}(y).P \mid x(z).Q)$$

Because x is a link on which information can be exchanged between P and Q , it makes sense to think of x as a link between P and Q .

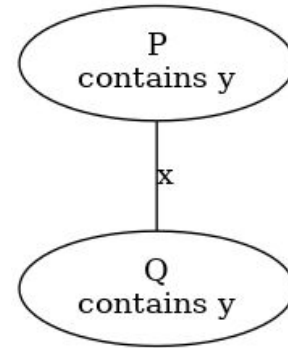
We can represent agents as nodes and links between them as edges in a graph that models the program.

Network topology

$(\bar{x}(y).P \mid x(z).Q)$



$(P \mid Q\{y/z\})$



The name y is passed from P to Q .
 y could be a link here, but no agent is waiting on a signal from y .

Passing links between agents

$$(\bar{x}(y).P \mid x(z).Q)$$

We have been passing around y between two processes. So far, y has just been a variable that represents information.

We could imagine that y itself is a link. To accomplish this, we must introduce a destination to be at the positive end of y .

We will call this new process R , and it must have the output prefix for y .

Passing links between agents

$$(\bar{x}(y).P \mid x(z).Q)$$

Will then become

$$(\bar{x}(y).P \mid x(z).Q \mid y(s).R)$$

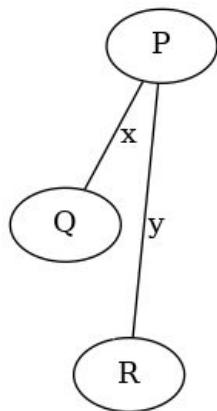
We have introduced an arbitrary destination agent R . This makes y a link that ends in R .

y can still be passed around along links, but now it is also a link itself.

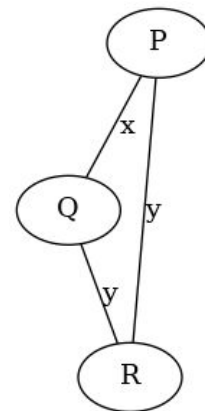
We will now look at how this affects the network topology.

Passing links between agents

$$(\bar{x}(y).P \mid x(z).Q \mid y(s).R)$$



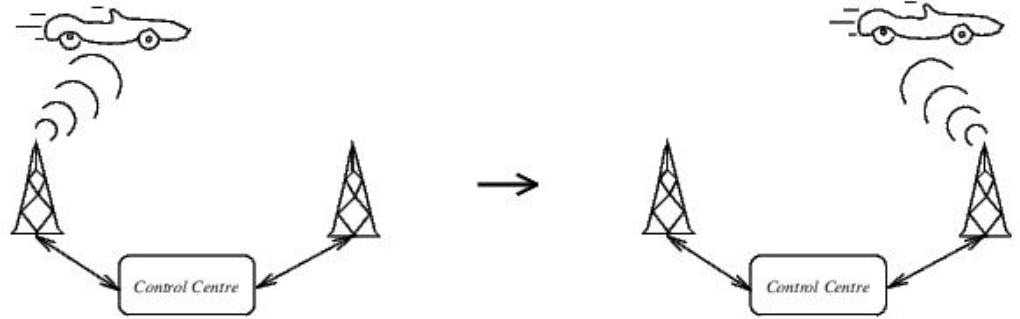
$$(P \mid Q\{y/z\} \mid y(s).R)$$



P has passed *y* (a link to *R*) to *Q* (via *x*).
By doing so, the network changes.

Changing network topology

The ability to represent computation over a changing network topology is an important aspect of the π -calculus.



A common example of a use case is the communication between a car radio and radio towers.

The car will come within range of a tower and may leave the range of others.

π -calculus in distributed computing

In the π -calculus so far, we have only defined communication over links as passing other links. We have not allowed passing agents themselves over links.

As a demonstration of the versatility of the π -calculus, I will show that this is almost possible.

Goal: pass an agent over a link

Take the following program:

$$(t(\varepsilon).P \mid Q)$$

Let P be an agent that does not use ε at all. In other words, P is waiting for some value over link t but does not use the data at all. ($\varepsilon \notin fn(P)$)

If Q does not send data over t , then P will never execute.

If Q does send data over t , regardless of the actual data, P can immediately begin execution.

Goal: pass an agent over a link

$$(t(\varepsilon).P \mid Q)$$

In this program, t can be called a trigger for P . Any other agent that has access to t can send data through it begin execution of P .

In this way, transmission of the link t over some other link is mostly identical to transmission of the agent P .

(P can only be executed once though, although this can be fixed through the special agent replication operator $*P$.)

The π -calculus can model any program

Although I will not show it here, if we add the following agent rule:

$*P$

Agent replication.
Identical to $(P|*P)$.

It is possible to embed any lambda term in the π -calculus. It is therefore possible to embed a program in any turing-complete language in the π -calculus.



π -calculus

Overall, I have showed that:

- The π -calculus is effective at modeling concurrent programs
- It can represent computation over a changing network landscape
- It can model sending processes themselves over links

I will now show how the π -calculus can actually be used in the real world.

The Pict Programming Language

The Pict programming language

I have chosen to talk about the Pict language because of how close it is to the π -calculus. Of course, some concessions must be made in order to function as a real programming language.

One of these is the translation of the π -calculus to ASCII:

π -calculus	Pict	
$\bar{x}y.0$	$x!y$	asynchronous output
$x(y).e$	$x?y = e$	input prefix
$e_1 \mid e_2$	$(e_1 \mid e_2)$	parallel composition
$(\nu x)e$	$(\text{new } x \ e)$	channel creation
$!x(y).e$	$x?*y = e$	replicated input

Simple programs similar to the π -calculus

We are used to creating concurrent agents and the zero agent:

$$(0 \mid 0 \mid 0)$$

```
run ( () | () | () )
```

In either case, no output is produced.

In Pict, the special channel *print* is introduced. Anything sent along this channel is sent to stdout. Using the input prefix notation $x!y$, we can create the hello world program:

```
run print!"Hello World"
```

Types in Pict

In Pict, every variable has a type, including channels. In the pure π -calculus, there are no types.

Some common types are `Bool`, `String`, and `Int`, and tuples.

Value	Type
32	<code>Int</code>
<code>[1 2 3]</code>	<code>[Int Int Int]</code>
<code>[]</code>	<code>[]</code>
<code>"Hello World"</code>	<code>String</code>

Channel Types in Pict

In Pict, channels have a distinct type. Every channel can carry values of only one type. Therefore, the type of a channel is defined by the type it carries.

We will denote the type of a channel carrying X's to be X .

Value	Type
Channel that carries numbers	$^{\text{Int}}$
Channel that carries pairs of numbers	$^{\text{[Int, Int]}}$
Channel that sends and receives Strings	$^{\text{String}}$

We will often use a channel of type $^{\text{[]}}$ to act as a trigger or a function that takes no arguments.

Usage of a channel

Unlike the π -calculus as I have introduced it, all channels must be declared before use. To declare a channel q of type K , we use the syntax:

```
new q : K
```

To send the integer 3 along channel x , we use:

```
x!3
```

To receive a value y along channel x and send y to print, we use:

```
x?y = print!y
```

Usage of a channel

Putting this all together, we obtain the following program.

(`{- this -}` is a comment)

```
new x : ^String      {- x is a channel sending and receiving Strings. -}  
run ( x?str = print!str {- when x receives str, send str to print -}  
    | x!"Hello" )    {- send "Hello" along x -}
```

Output: Hello

Usage of a channel: similar to a variable

In this Pict program, we are:

- Declaring a name (x) which can be used to interact with information
- “Giving” x the information “Hello”
- Retrieving the information and sending it to print.

All of this might seem familiar to the following C++ program:

This is evidence that channels in Pict can act like mutable variables in other languages.

```
new x : ^String  
  
run ( x?str = print!str  
      | x!"Hello" )
```

```
int main()  
{  
    string x;  
    x = "Hello";  
    printf(x);  
}
```

Procedures

If we have a channel declared as:

```
new x : ^[]
```

(a channel receives any value of the type [] -- which can only be [])

and we define the following:

```
run x?e = print!"X was executed"
```

Then, anything we send along x will cause “X was executed” to be sent to the console. This may feel similar to the following π -calculus term:

$$(t(\varepsilon).P \mid Q)$$

Procedures

This program defines a “procedure” x:

```
new x : ^[]  
run x?e = print!"X was executed"
```

...and this will “execute” it:

```
run x![]
```

Before, we made a comparison between Pict’s channels and mutable variables. We can now also think of them as procedures.

Literals in Pict

In the pure π -calculus, there are no literals such as “strings” or 12. However, Pict gives us these “for free.”

However, it is possible to define booleans without the need for special literals. I will show how this is possible, then show how if-then-else statements arise naturally out of them.

This will appear familiar to the construction of bools in the lambda calculus.

Usage example of a bool

Normally, when using a bool, we will have some construction similar to the following:

```
if (condition)
  then
    true_case
  else
    false_case
```

We want this to be equivalent to `true_case` if `condition` is true, and `false_case` otherwise.

Bools in Pict

In Pict, we will define the values:

```
new TRUE : ^[^[] ^[]]  
run TRUE?[true_case false_case] = true_case![]
```

Here, we have defined TRUE to be a channel that takes pairs of procedures. When TRUE is given a pair of two procedures, it will execute (by sending the empty list along) the first one.

Bools in Pict

We will also define false:

```
new FALSE : ^[^[] ^[]]  
run FALSE?[true_case false_case] = false_case![]
```

FALSE acts the same as TRUE, but it will execute the second argument it is given instead of the first.

FALSE will essentially select the second argument, while TRUE will select the first.

Conditional statements in Pict

Now, we can have statements that produce different output depending on FALSE or TRUE:

```
1 new TRUE : ^[^[] ^[]]
2 run TRUE?[true_case false_case] = true_case![]
3
4 new FALSE : ^[^[] ^[]]
5 run FALSE?[true_case false_case] = false_case![]
6
7 new true_case : ^[]
8 run true_case?e = print!"It was TRUE."
9
10 new false_case : ^[]
11 run false_case?e = print!"It was FALSE."
12
13 run TRUE![true_case false_case]
14
```

Conditional statements in Pict

In our program, the conditional statement is on line 13:

```
13 run TRUE![true_case false_case]
```

Because TRUE selects the first argument, this will trigger the true_case, which was defined:

```
8 run true_case?e = print!"It was TRUE."
```

So the program will output:

```
It was TRUE.
```

Conditional statements in Pict

If we had however written:

```
13 run FALSE![true_case false_case]
```

FALSE would have selected the second case, which would then print:

```
It was FALSE.
```

In this way, we can simulate conditional statements in Pict.

From the π -calculus to Pict

The π -calculus only provides facilities for agents and inter-agent communication. However, Pict can be developed from these basic definitions. We can define:

- Mutable variables
- Procedures and Functions
- Conditional statements
- Output to stdout.

Applications of the π -calculus and Pict

- Modeling business processes (see Business Process Modeling Language (BPML), XLANG)
- Simulation of the signalling pathway in a cell
 - It is possible to simulate the entire metabolic network of a simple cell using a variant of π -calculus
- Pict as a compilation target
- Reasoning about highly concurrent problems.

OPEN ACCESS Freely available online

PLoS COMPUTATIONAL BIOLOGY

A Computational Approach to the Functional Screening of Genomes

Davide Chiarugi¹, Pierpaolo Degano², Roberto Marangoni^{2*}

¹ Dipartimento di Matematica e Informatica, Università di Siena, Siena, Italy, ² Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Comparative genomics usually involves managing the functional aspects of genomes, by simply comparing gene-by-gene functions. Following this approach, Mushegian and Koonin proposed a hypothetical minimal genome, Minimal Gene Set (MGS), aiming for a possible oldest ancestor genome. They obtained MGS by comparing the genomes of two simple bacteria and eliminating duplicated or functionally identical genes. The authors raised the fundamental question of whether a hypothetical organism possessing MGS is able to live or not. We attacked this viability problem specifying *in silico* the metabolic pathways of the MGS-based prokaryote. We then performed a dynamic simulation of cellular metabolic activities in order to check whether the MGS-prokaryote reaches some equilibrium state and produces the necessary biomass. We assumed these two conditions to be necessary for a living organism. Our simulations clearly show that the MGS does not express an organism that is able to live. We then iteratively proceeded with functional replacements in order to obtain a genome composition that gives rise to equilibrium. We ruled out 76 of the original 254 genes in the MGS, because they resulted in duplication from a functional point of view. We also added seven genes not present in the MGS. These genes encode for enzymes involved in critical nodes of the metabolic network. These modifications led to a genome composed of 187 elements expressing a virtually living organism, Virtual Cell (ViCe), that exhibits homeostatic capabilities and produces biomass. Moreover, the steady-state distribution of the concentrations of virtual metabolites that resulted was similar to that experimentally measured in bacteria. We conclude then that ViCe is able to “live *in silico*.”

Citation: Chiarugi D, Degano P, Marangoni R (2007) A computational approach to the functional screening of genomes. *PLoS Comput Biol* 3(9): e174. doi:10.1371/journal.pcbi.0030174

Introduction

The search for LUCA, the Last Universal Common

consist in synthesizing this genome, in cloning it in a ghost bacterium, and in evaluating the overall cell viability.

Cryptographic applications

- ProVerif: automatic crypto protocol verification
- ProVerif uses a variant of the π -calculus, the Spi calculus

```
(* Secrecy assumptions *)

not attacker(new skA).
not attacker(new skB).
not attacker(new skS).

(* 2 honest host names A and B *)

free A, B: host.

(* the table host names/keys
   The key table consists of pairs (host, public key) *)
table keys(host, pkey).

(* Queries *)

free secretANa, secretANb, secretBNa, secretBNb: bitstring [private].
query attacker(secretANa);
      attacker(secretANb);
      attacker(secretBNa);
      attacker(secretBNb).
```


Questions?

π -calculus:

- motivation
- syntax
- agents
- channels
 - network topology
 - channels over channels
 - changing network
 - agents over channels

Pict Language:

- types
 - channel types
- agent syntax
- channel syntax
 - channels as variables
 - channels as procedures
- procedures
- boolean literals
 - conditional statements