A π-Calculus Internal Domain-Specific Language for Scala

Pedro Matiello
pmatiello@gmail.com

Ana C. V. de Melo (advisor)
acvm@ime.usp.br

Contents

1 Introduction

The π-calculus is one of many approaches to concurrent computation by the means of formal modeling. It provides a simple, yet expressive, language capable of describing process interaction and reconfiguration uniquely through communication in the form of message passing.

Scala is a general purpose programming language that compile to Java bytecodes. It integrates features both from object-oriented programming and functional programming. Also, it provides a sophisticated static type system, but still manages to conciliate this with a flexible and consise syntax.

A Domain Specific Language (DSL) a programming language designed to a specific purpose. A DSL is said to be internal (or embedded) when it is built within a host language (usually as a library or framework).

This document presents Pistache[1]: an implementation of the π-calculus as a domain specific language hosted in Scala.

The realm of sequential programs has been positively and significantly impacted by the introduction of several abstractions, often packed in paradigms such as structured and object-oriented programming, and in collections of data types. These abstractions facilitated the reasoning on these types of programs by encapsulating low-level details behind more understandable symbols and operations.

A similar breakthrough is yet to happen for concurrent programs. Still, this project has been developed under the understanding that the π-calculus might contribute to this goal.

[1] http://code.google.com/p/pistache/

2 The π-Calculus

The π-calculus, introduced by Milner, Parrow and Walker in [MPW89], is a process calculus for concurrent computation with dynamic reconfiguration. Agents (also called processes) communicate by the exchange of names through channels (links). Since channels are names, the interconnections may change as they are passed and shared.

This section is a brief introduction to an orthodox version of the calculus, which is synchronous and monadic. For a more detailed description, one should refer to [MPW89], [Par01] and [Mil99].

2.1 Definitions and Syntax

Given a set of *names* $\{x, y, z, ...\}$, a set of *agent identifiers* $\{A, B, C, ...\}$, each having an arity, and a set of *agents* $\{P, Q, R, ...\}$. The syntax of the agents is defined as follows:

Prefixes	$\alpha ::=$	$\bar{y}x$	Output
		$y(x)$	Input
		τ	Silent
Agents	$P ::=$	0	Nil
		$\alpha.P$	Prefix
		$P + P$	Sum
		$P \mid P$	Parallel
		$(\nu x)P$	Restriction
		$[x = y].P$	Match
		$[x \neq y].P$	Mismatch
Definitions		$A(x_1, ..., x_n) \stackrel{\text{def}}{=} P$	

Therefore, agents can be of the following forms:

- The empty agent **0**.
- An *output prefix* $\bar{y}x.P$ that sends the name x through the channel y and then continues as P.

- An *input prefix* $y(x).P$ that receives some name along the channel y, places it on x, and then continues as P.
- A *silent prefix* $\tau.P$ that continues as P after executing a silent action τ, which does not interacts with the environment.
- A *sum* $P + Q$ that can behave either as P or as Q.
- A *parallel composition* $P|Q$ that behaves as P and Q running in parallel.
- An *restriction* $(\nu x).P$ that behaves as P having the name x in its local scope.
- A *match* $[x = y].P$ that behaves as P if x and y are the same name.
- A *mismatch* $[x \neq y].P$ that behaves as P if x and y are not the same name.
- A *defined agent* $A(y_1, ..., y_n)$, from an *defining equation* $A(x_1, ..., x_n) \stackrel{\text{def}}{=} P$, that behaves like $P\{y_i/x_i\}$.

2.2 Semantics

Traditionally, the operational semantics of a process algebra is given by a labelled transition system, where transitions are of kind $P \xrightarrow{\alpha} Q$, for agents ranging over $\{P, Q, ...\}$ and transitions ranging over $\{\alpha, ...\}$. In $P \xrightarrow{\alpha} Q$, it is understood that P is subject to a labelled transition α leading to Q.

Regarding the π-calculus, the transition rules are:

Prefix	$\frac{}{\alpha.P \xrightarrow{\alpha} P}$	Restriction	$\frac{P \xrightarrow{\alpha} P', x \notin \alpha}{(\nu x).P \xrightarrow{\alpha} (\nu x).P'}$
Summation	$\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$	Summation	$\frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$
Match	$\frac{\alpha.P \xrightarrow{\alpha} P}{[x=x].P \xrightarrow{\alpha} P}$	Mismatch	$\frac{\alpha.P \xrightarrow{\alpha} P, x \neq y}{[x \neq y].P \xrightarrow{\alpha} P}$
Parallel	$\frac{P \xrightarrow{\alpha} P', bn(\alpha) \cap fn(Q) = \emptyset}{P\|Q \xrightarrow{\alpha} P'\|Q}$	Parallel	$\frac{Q \xrightarrow{\alpha} Q', bn(\alpha) \cap fn(P) = \emptyset}{P\|Q \xrightarrow{\alpha} P\|Q'}$
Communication	$\frac{P \xrightarrow{\alpha(x)} P', Q \xrightarrow{\bar{\alpha}u} Q'}{P\|Q \xrightarrow{\alpha} P'\{u/x\}\|Q'}$		

2.3 An Example

The following example, inspired by [Par01], shall illustrate the calculus.

Let C, P and S be agents for a client, a printer and a print server, respectively. The print server and the client share a communication channel a, and the print server and the printer share another communication channel b. The intended interaction is to have the S to share access to P with C, and then to have C to send a message to P. Also, after performing their tasks, the agents P and S should return to their starting state and the agent C should stop.

The agents can be defined as:

$C = a(p).\bar{p}x$
$P = b(y).P$
$S = \bar{a}b.S$

And the entire situation happens by the parallel composition of these three agents:

$C|P|S$

2.4 Variants

The π-calculus can be varied in many ways, for achieving many different effects (including, often, simpler mechanics that are friendlier to concrete implementations). Some of the common variations are:

Guarded sums The standard π-calculus summation performs a global choice synchronously, which is cumbersome in many implementation scenarios. In order to preserve the operator, many presentations of the calculus use guarded sums, which is a summation such that that every agent P_i in $P_1 + ... + P_n$ is preceded (guarded) by either an input or by an output prefix (i.e. $P = y(x).P'$ or $P = \bar{y}x.P'$).

Polyadic In the standard π-calculus, channels transmit a single name at time. The polyadic π-calculus, on the other hand, allows channels of arbitrary arities. Therefore, in this variant, outputs of the form $\bar{y}\langle x_1, ..., x_n\rangle$ and inputs of the form $y(x_1, ..., x_n)$ are valid.

Asynchronous The standard π-calculus determines that every communication between two agents is synchronous: the emission and the reception of a message always happen at the same time (and agents too advanced in the computation and trying to send a message will have to wait until some other agent catches up to receive it). The asynchronous variant of the calculus does not have this requirement, allowing an process to emit a name through a channel and continue its computation even if no other agent is listening for messages from this channel.

2.5 Implementations

Being such a simple and expressive language for expressing concurrence, a set of concrete implementations of the π-calculus were developed. They implement different fragments of the calculus and target different uses. A small survey follows.

2.5.1 PiLib

PiLib is an implementation of the π-calculus for Scala introduced in [CO03] by Vincent Cremet and Martin Odersky. It is part of Scala's standard library and provides a monadic, synchronous and typed version of the calculus, without match nor mismatch, and with guarded sums.

The same paper provides as example a specification of a two-place buffer:

$$Buffer(put, get) = B_0(put, get)$$
$$B_0(put, get) = put(x).B_1(put, get, x)$$
$$B_1(put, get, x) = \overline{get}x.B_0(put, get) + put(y).B_2(put, get, x, y)$$
$$B_2(put, get, x, y) = \overline{get}x.B_1(put, get, y)$$

The same specification is written as Scala with PiLib code as bellow.

```
def Buffer[a](put: Chan[a], get: Chan[a]):unit = {
    def B0:unit = choice(put * { x => B1(x) });
    def B1(x: a):unit = choice (
        get(x) * B0,
        put * { y => B2(x, y) } );
    def B2(x: a, y: a):unit = choice(get(x)
                                     * B1(y));
    B0
}
```

This table summarises PiLib's syntax:

π-calculus	PiLib
$P = ...$	`def P {...}`
$\bar{y}x.P$	`y(x)*P`
$y(x).P$	`y*{x=>P(x)}`
$P + Q$	`choice(P, Q)`
$P \vert Q$	`spawn < P | Q >`
$(\nu x)P$	`def proc {val x; P}`

2.5.2 Pict

Pict [PT97] is a programming language based on the π-calculus computation model. Due to implementation concerns, the calculus here is polyadic, asynchronous, typed and without summation. It provides a syntax similar to the original calculus and some predefined processes and links for convenience.

The following program spawns two processes and creates a link called `ch`. Then, the first process sends the string `"Something"` through the link to the second process, which prints it.

```
run (new ch:^String (ch!"Something"
                     | ch?msg = print!msg))
```

This table summarises Pict's syntax:

π-calculus	Pict
$P = ...$	`def P = ...`
$\bar{y}x.P$	`y!x = P`
$y(x).P$	`y?x = P`
$P \vert Q$	`P | Q`
$(\nu x)P$	`(new x P)`

2.5.3 Kroc

Kroc (Kent Retargetable occam-pi Compiler) [WB05] is an occam-pi compiler for Intel 386 and compatible processors. The occam-pi language itself is a variation of the original occam language built on the π-calculus instead of the Communicating sequential processes (CSP) [Hoa78].

The occam-pi language implements a typed, synchronous and monadic (including data structures) variant of the calculus. Guarded sums are supported. Process mobility (between different machines) is also supported.

The following example illustrates message passing, process definition, composition and sequencing in occam-pi.

```
PROC example1 (CHAN BYTE out!)
  -- (Private) Channel creation
  CHAN INT link:

  -- Process definition
  PROC print()
    INT value:
    SEQ  -- Sequential process
      link ? value  -- input prefix
      out.int (value, 0, out!)
  :

  -- Process with arguments
  PROC sumProc(VAL INT x, y)
    link ! x + y  -- output prefix
  :
  PAR  -- Parallel composition
    sumProc(2, 3)
    print()
:
```

Bellow, a deliberately complicated example for introducing guarded processes in occam-pi. The producer processes sends even numbers through one link and odd numbers through the other. The consumer process then watches both links and reacts accordingly when one of them is filled.

```
PROC example2 (CHAN BYTE out!)
  CHAN INT even:
  CHAN INT odd:

  PROC consumer()
    WHILE TRUE
      INT value:
      ALT  -- Guarded process (summation)
        even ? value
          out.int (value, 0, out!)
        odd ? value
          out.int (value, 0, out!)
  :
```

```
  PROC producer()
    INITIAL INT n IS 0:
    WHILE TRUE
      SEQ
        even ! n
        odd ! n+1
        n := n + 2
  :

  PAR
    producer()
    consumer()
:
```

This table summarises occam-pi's syntax:

π-calculus	**occam-pi**

$P = ...$	`PROC P`
	`  ...`
	`:`

$\bar{y}x.P$	`SEQ`
	`  y ! x`
	`  P()`

$y(x).P$	`SEQ`
	`  y ? x`
	`  P()`

$y(x).P + z(x).Q$	`ALT`
	`  y ? x`
	`  P()`
	`  z ? x`
	`  Q()`

$P\|Q$	`PAR`
	`  P()`
	`  Q()`

$(\nu x)P$	`PROC proc`
	`  TYPE x`
	`  P`
	`:`

3 The Scala Programming Language

Scala[2] is a general purpose programming language designed to integrate features of object-oriented programming and functional programming. It is one of the many new languages that compile to Java bytecodes. Although it is a statically typed language (i.e. values types are known and checked at compile time), it offers some mechanisms to bring as much as possible of the conveniences of dynamically typed languages without sacrificing the safety provided by static typing.

This chapter should work as an brief introduction to this language. The interested reader is advised to refer to [OSV08] for an extensive guide. Those who are familiar with Java might also find [SH09] useful.

3.1 The Basics

Variable declaration Scala supports two types of variables: `val`s, which can be assigned only once, and `var`s, which can be reassigned.

```
val string:String = "Not reassignable"
var string:String = "Reassignable"
```

In most cases, the compiler can infer the type of the variables. Therefore, some of the typing can be avoided:

```
val string = "Not reassignable"
var string = "Reassignable"
```

Method definition Methods are defined with the `def` keyword.

```
def max(x:Int, y:Int):Int = {
    if (x > y) x else y
}
```

Methods containing a single statement, like the one above, can omit the curly braces:

```
def max(x:Int, y:Int):Int = if (x > y) x else y
```

And, in many cases, the return type can also be inferred by the compiler:

```
def max(x:Int, y:Int) = if (x > y) x else y
```

[2] http://www.scala-lang.org/

Class definition Classes, defined by the `class` keyword, can be used to wrap variables and methods.

```
class SpecialInt(int:Int) {
    def isPositive = int >= 0
    def isNegative = int <= 0
}
```

The arguments in the first line belong to the class constructor and are visible to contained methods. Instances of classes are built using the `new` keyword:

```
val number = new SpecialInt(10)
```

3.2 Inheritance and Traits

Scala provides two mechanisms of inheritance. The first is subclassing:

```
class VerySpecialInt(int:Int) extends SpecialInt(int:Int) {
    def isZero = int == 0
}
```

The second mechanism is a construct called `trait`, which can be understood as a Java `interface` supporting method implementations.

```
trait Person {
    def sleep { Thread sleep 1000 }
    def talk:Unit
}

class NicePerson extends Person {
    def talk { println("Hello") }
}
```

The `NicePerson` class is forced by the compiler to provide an implementation to the `talk` method. The method `sleep` can be optionally provided.

```
class LazyPerson extends Person {
    def talk { println("Zzzz") }
    override def sleep { Thread sleep 2000 }
}
```

3.3 Implicit Conversions

Given a situation where an expression E of type T is expected to be of type S, and T does not extends S, the Scala compiler will try to implicitly convert

E to type S by using a predefined conversion rule.

The simplest use case is to convert a value to an expected type on a method call:

```
implicit def Int2String(int:Int) = int.toString

def len(str:String) = str.size
```

With the conversion above in scope, the method `len` can be called with an integer as argument (and will return the length of its decimal representation).

This mechanism can also be used to new methods to an existing class:

```
class SpecialInt(int:Int) {
    def isPositive = int >= 0
    def isNegative = int <= 0
}

implicit def Int2SpecialInt(int:Int) = new SpecialInt(int)
```

When the conversion above is in scope, code like `3.isPositive` will compile just like the `Int` class actually had a method called `isPositive`.

3.4 Pattern Matching

Scala supports a construct called *case classes*. A class with a `case` modifier will always export all its constructor parameters as public class attributes and allow recursive decomposition by another feature of the language called *pattern matching*.

Pattern matching is one more flow control mechanism. The mechanism will match values of any type and will execute different branches of the code depending of the type of the matched object and of the values enclosed by it.

This example illustrates the use of pattern matching on case classes:

```
trait Human
case class Man(name:String, age:Int) extends Human
case class Woman(name:String, age:Int) extends Human

def whoIs(human:Human) {
    human match {
        case Man(name, age) => println("His name is " + name)
        case Woman(name, age) => println("Her name is " +
            name)
    }
}
```

4 Application Programming Interface

Pistache provides an internal domain-specific language for writing π-calculus programs in Scala. Among the usual pi-Calculus features and operations, the following are supported:

- Agent definition, including agents with arguments
- Prefix and Prefix-Agent concatenation (for sequential execution)
- Agent composition (for parallel execution)
- Guarded agent summation
- Scope restriction for names
- Links for sending and receiving names (including links)
- If structure for matching
- Silent transitions

4.1 Syntax

This section explains the supported syntax.

4.1.1 Agent Definition

The common idiom for defining agents is:

```
val agent = Agent(...)
```

Self-referential agents, used to implement recursive behaviour, require some boilerplate code. In order to satisfy Scala's type checker in compile time, the `Agent` type can not be omitted. Also, to avoid runtime errors due to references to an still non-instantiated object, lazy evaluation[3] must be used.

```
lazy val recursiveAgent:Agent = Agent(...)
```

[3]Lazy evaluation can be defined by two main traits: "First, the evaluation of a given expression is delayed, or *suspended*, until its result is needed. Second, the first time a suspended expression is evaluated, the result is *memoized* (i.e., cached) so that, if it is ever needed again, it can be looked up rather than recomputed." [Oka98].

Agents with arguments can be defined as below:

```
def agentWithArgs(arg1:Type1, ..., argN:TypeN):Agent =
    Agent {
        ...
    }
```

4.1.2 Names

Names can be used as references to objects. For instance:

```
val name = Name(some_object)
```

It is also possible to specify a name holding no reference:

```
val name = Name[Type]
```

The referred object then can be set or changed:

```
name := other_object
```

And retrieved:

```
referred_object = name.value
```

4.1.3 Links

Links are the mechanism provided to address the communication between processes. Although it is not standard in π-calculus, the links in Pistache are typed (so the values transmitted must be instances of the specified type). As expected, they are created by calling `Link`:

```
val link = Link[Type]
```

The syntax for sending a name through a link is:

```
link~name
```

The sent reference can then be received and bound to another name on another agent:

```
link(another_name)
```

4.1.4 Scope Restriction

It is often necessary to have names restricted to a single agent. This is also possible:

```
val agentWithRestrictedName = Agent {
    val restrictedName = Name(...)
    ...
}
```

4.1.5 Silent Transitions

Silent transitions do not communicate with the environment, but actually perform an action within the context of the process. These actions are an ordinary closures, without any arguments nor return type, wrapped as an agent by calling `Action`:

```
val silentTransition = Action{ ... }
```

It is possible to have an agent containing a single silent transition. For instance:

```
val silent = Action{ doSomething() }
val agent = Agent(silent)
```

4.1.6 Prefix Concatenation

Agents can be made by sequencing the prefixes:

```
val sequentialAgent = Agent(prefix1*...*prefixN)
```

A recursive agent can be defined along the same lines:

```
lazy val recursiveAgent:Agent =
    Agent(prefix1*...*prefixN*recursiveAgent)
```

As silent transitions are understood as prefixes, they can be concatenated:

```
val silent1 = Action{ doSomething() }
val silent2 = Action{ doSomethingElse() }
val agent = Agent(silent1*silent2)
```

4.1.7 Agent Composition

It is also possible to a agent to be composed of other agents running in parallel:

```
val composedAgent = Agent(agent1 | ... | agentN)
```

And having more than one agent running makes communication possible:

```
val square = Link[Link[Int]]

val C = Agent {
    val link = Link[Int]
    val reply = Name[Int]
    val print = Action { println(reply.value) }
    square~link*link~5*link(reply)*print
}

val S = Agent {
    val link = Name[Link[Int]]
    val number = Name[Int]
    val calculate = Action {
        number := number.value * number.value
    }
    square(link)*link(number)*calculate*link~number
}
```

When `C | S` is executed, `C` will send a channel named `link` to `S`, which will be used by `C` to send an integer to `S`. Then, `S` will use this same link to send back to `C` to square of the integer sent.

4.1.8 Agent Summation

Agents can be guarded by prefixes (input, output or silent transitions). Although in π-calculus the ordinary concatenation operator is used to guard an agent, Pistache requires the use of a semantically equivalent guard operator.

```
val guardedAgent = Agent {
    guardPrefix :: Agent
}
```

Guarded agents can be used in summations. When a summation is executed, only one of its terms will be selected and executed.

```
val summation = Agent {
    val t = Action { ... }
    (y~(x) :: P1) + (y(x) :: P2) + (t :: P3)
}
```

4.1.9 Matching

In order to direct the execution flow of the programs, the `If` match structure is provided.

The accepted syntax is:

```
val agent = Agent(If (condition) {thenAgent})
```

When executed, if `condition` evaluates to `true`, the agent `thenAgent` is to be executed. Otherwise, the agent just halts.

The composition operator can be used if some agent is also to be executed when the given condition evaluates to `false`:

```
If (condition) {P} | If (!condition) {Q}
```

4.2 Internal Representation

Pistache types are crafted to enforce π-calculus' syntactic rules. Therefore, code written in the syntax presented above is checked at compile time for type errors. This checking is provided by the Scala compiler and prevents the programmer from writting meaningless code like:

```
val agent = Agent(p1*45.3)
```

(The compiler will rightfuly output something like: `type mismatch; found : Double(45.3) required: pistache.picalculus.Agent`.)

Avoiding type errors like this is the duty of the type system[4], but this can only be done properly by having the types to reflect the domain model as closely as possible. Therefore, in order to provide both π-calculus' restrictions and flexibilities, a number of types were implemented.

The following traits are present:

[4]"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." [Pie02].

Type Name	Description
PiObject	A trait for tagging all π-calculus' objects as such. Provides no behaviour.
Concatenation	A trait providing the prefix concatenation operation.
Composition	A trait providing the parallel composition operation.
Summation	A trait providing the summation operation.
Guard	A trait providing the guard operation.
Prefix	A trait for all π-calculus' prefixes.
Agent	A trait for all π-calculus' agents.

```
trait PiObject

trait Concatenation{
    def *(other: => Prefix):ConcatenationPrefix
    def *(other: => Agent):ConcatenationAgent
}

trait Composition {
    def |(other: => Agent):CompositionAgent
}

trait Summation {
    def +(other: => GuardedAgent):SummationAgent
}

trait Guard {
    def ::(other: => Prefix):GuardedAgent
}


trait Prefix extends PiObject with Concatenation

trait Agent extends PiObject with Composition with Guard
```

The prefixes are provided by the following case classes:

Type Name	Description
ActionPrefix	A case class representing silent transitions.
ConcatenationPrefix	A case class representing the concatenation of two prefixes.
LinkPrefix	A case class representing the an action (input or output) through a link.

```
case class ActionPrefix(val procedure: () => Unit) extends
   Prefix

case class ConcatenationPrefix(val left: () => Prefix, val
   right: () => Prefix) extends Prefix

case class LinkPrefix[T](val link:Link[T], val action:
   ActionType, val name:Name[T]) extends Prefix
```

And agents are provided by the following case classes:

Type Name	Description
ConcatenationAgent	A case class representing the concatenation of a prefix and an agent.
GuardedAgent	A case class representing an agent guarded by a prefix.
CompositionAgent	A case class representing the parallel composition of two agents.
SummationAgent	A case class representing the summation of two agents.
MatchAgent	A case class representing an agent conditioned by a match.
NilAgent	A case class representing the null agent.
RestrictedAgent	A case class representing agents with support for restricted names.

```
case class ConcatenationAgent(val left: () => Prefix, val
   right: () => Agent) extends Agent

case class GuardedAgent(val left: () => Prefix, val right: ()
    => Agent) extends Agent with Summation

case class CompositionAgent(val left: () => Agent, val right:
    () => Agent) extends Agent
```

```
case class SummationAgent(val left: () => Agent, val right:
    () => Agent) extends Agent with Summation

case class MatchAgent(val condition: () => Boolean, val then:
     () => Agent) extends Agent

case class NilAgent() extends Agent

case class RestrictedAgent(val agent: () => Agent) extends
    Agent
```

Assuming that the programmer made no syntax mistakes, the compiler should be satisfied. Then, at runtime, full π-calculus agents will be built from their atomic parts, as specified by the programmer. The following examples ilustrates how agents and prefixes are built from concatenation:

```
// suppose already defined prefixes p1, p2, p3
// and an agent Q
val P = Agent(p1*p2*p3*Q)
```

This is the agent $P = p_1.p_2.p_3.Q$ written as a Pistache object. The method `*` belongs to the prefix on its left side, takes as argument the prefix on its right side, and returns a specific type of prefix, called `ConcatenationPrefix` if the right side is a prefix, and called `ConcatenationAgent` if the right side is an agent.

So, the first invocation of `*` has `p1` on the left side, `p2` on the right side, and produces `ConcatenationPrefix(p1, p2)` as a result. The second invocation has `ConcatenationPrefix(p1, p2)` on the left side, `p3` on the right side, and produces `ConcatenationPrefix(ConcatenationPrefix(p1, p2), p3)` as a result. And that's how it goes.

The same happens for process composition. The only difference is that the type that wraps the agents on the left and on the right is called `CompositionAgent`. So, the parallel agent `Agent(P | Q | R)` is built as `CompositionAgent(CompositionAgent(P, Q), R)`.

Now, a last case must be considered:

```
// suppose already defined prefixes p1, p2, p3
// and an agent Q
val P = Agent(p1*p2*p3 | Q)
```

As seen before, `p1*p2*p3` is of type `ConcatenationPrefix`, which does not provide a `|` method for parallel composition. The whole assignment accounts for the agent $P = p_1.p_2.p_3|Q$ in pen-and-paper π-calculus. As this

is informally accepted as a valid expression in the calculus (as abbreviation of $P = p_1.p_2.p_3.0|Q$), one would expect that the code above would compile fine. In order to do so, Pistache uses an implicit conversion to produce an agent from the concatenated prefixes.

More precisely: from `p1*...*pN` it will produce `ConcatenationAgent( p1*...*pN, NilAgent)` which accounts for the explicitly terminated agent $p_1...p_n0$ in π-calculus.

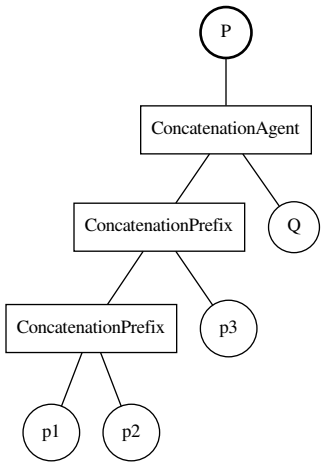

Figure 1: Internal representation of a sequential process

5 Execution Model

As the agent objects are just simple data structures, devoid of any behaviour, an external mechanism is needed to turn all these objects into an actual program. This task is performed by an specification runner, which takes an agent as argument, interprets its structure and executes the appropriate actions.

This approach provides some flexibility, allowing for the possibility of implementation of different runners, tailored for different execution environments or variants of the π-calculus.

5.1 The Thread-Based Runner

The thread-based runner, named `ThreadedRunner` in Pistache, uses ordinary system threads to execute π-calculus agents concurrently (basically, every agent has its own thread, and their execution can be carried by as many cores are available in the machine). The communication between agents is synchronous and it is prepared to handle any expression accepted by the API (i.e. any compilable expression).

In order to understand this implementation, both the general execution algorithm and the message passing mechanism must be observed.

5.1.1 Message Passing

As agents are concerned, two operations are possible on channels: output and input. In the present case the communication is synchronous, therefore agents attempting to send a message are blocked until a message is actually received by some other agent, and agents attempting to receive a message are blocked until a message is actually sent.

The procedure is described bellow. It is assumed in the description that, for every channel, both the output and input sides share a lock for mutual exclusion in order to keep at most one thread active at the procedure at any moment. When an thread is set to wait, its execution will pause until it is awaken by a signal. Different channels use different lock objects in order to avoid interference.

Output

Wait until the channel is empty
Put the message in the buffer
Signal the channel not empty
Wait until the channel is empty

Input

Wait until the channel is not empty
Get the message from the buffer
Signal the channel empty

This is very simple and works properly for agents in a parallel composition, but not for agents in a summation. Because at most one term of a summation is to be executed, the execution of the guard prefixes must be attempted sequentially (and not concurrently) until one of them succeeds. Still, the algorithm above is not prepared for polling and will lock the thread until the first (input or output) prefix actually performs a communication, which might not happen at all.

The following version is prepared for this situation. Non-guarded I/O uses the modified *Input* and *Output* algorithms bellow, while guarded I/O, not surprisingly, uses the *Guarded Input* and *Guarded Output* algorithms. These guarded variants are responsible for enforcing that the appropriate preconditions are satisfied before calling the basic input or output algorithms.

The same locking pattern used in the previous version applies here. Also, the guarded variants report their success status as return value, as this information is required by the runner in order to determine the execution flow.

Output

Wait until the channel is empty *and*
    (there is no writer *or* the writer is me)
Set the writer as myself
Put the message in the buffer
Signal the channel not empty
Wait until the buffer is empty
Unset the writer
Signal the channel unblocked

Guarded Output

If there is no writer *or* the channel is empty
    Then set the writer as myself
If the channel is empty *and* there is a reader which is not me:
    Then set the writer as myself and start `Output`
Otherwise, fail

Input

Set the reader as myself
Wait until the channel is not empty
Get the message from the buffer
Signal the channel empty
Unset the reader
Signal the channel blocked

Guarded Input

If the channel is not blocked *and* there is a writer which is not me:
- Then, if the channel is not empty: Start `Input`
- Otherwise, set the reader as myself and fail
Otherwise, fail

In this version, there are three new variables. Both `writer` and `reader` are used to announce the intent of communicating through a channel and to avoid that terms of the same summation attempt to communicate to each other. The variable `blocked` is used to avoid a race condition where some potential reader is mislead into expecting a message in a channel where the former reader have already left but the former writer have not.

5.1.2 Algorithm Overview

The algorithm for concurrent execution of π-calculus specifications is described bellow. Relevant comments are inserted within code fragments in order to properly explain the fragment in question.

```
function start(agent):
    initialize the list of threads
    run(agent) in new thread
    wait until all threads executing agents are finished
```

(This function will start the computation. It will only return when all threads executing agents terminate.)

```
private function run(node):

    node match:

        case ActionPrefix(procedure):
            execute procedure
```

(Execute the closure wrapped in a ActionPrefix object.)

```
        case ConcatenationAgent(left, right):
            run(left)
            run(right)

        case GuardedAgent(left, right):
            run(left)
            run(right)
```

(Both expressions above match agents like $l.R$, where l is a prefix (or a non-terminated sequence of prefixes) and R is a agent. Their execution is performed sequentially, with the agent following the prefix.)

```
        case ConcatenationPrefix(left, right):
            execute(left apply)
            execute(right apply)
```

(Execute a sequence $l.r$ of prefixes, one after another. A sequence of prefixes is also considered a prefix here.)

```
        case CompositionAgent(left, right):
            run(left) in new thread
            run(right) in new thread
```

(This matches expressions like $L|R$. New threads are produced to execute L and R concurrently.

```
        case SummationAgent(left, right):
            agents = shuffle([terms(left) terms(right)])
            continue = null
            while (continue is null):
                for each in agents:
                    if not done:
                        each.left match:
                            case ActionPrefix(procedure):
                                execute procedure
                                continue = each.right
```

```
                        case LinkAgent(link, Send, name):
                            if guardedSend(link, name)
                                continue = each.right
                        case LinkAgent(link, Receive,
                           name):
                            if guardedRecv(link, name)
                                continue = each.right
        run(continue)
```

(This starts producing a list of terms $\alpha_1.P_1, ..., \alpha_n.P_n$ for a $\alpha_1.P_1 + ... + \alpha_n.P_n$. This list is shuffled to ensure non-determinism. Then, every prefix α_i will have its execution attempted until one of them succeeds. Once this is the case for α_k, P_k will be executed and every other $\alpha_j.P_j, j \neq k$ will be abandoned.)

```
case IfAgent(condition, branch):
    if (condition is true): run(branch)
```

(This matches expressions like $If(condition)B$. The agent B will be executed if $condition == true$.)

```
case LinkAgent(link, Send, name):
    send(link, name)

case LinkAgent(link, Receive, name):
    receive(link, name)
```

(The two matches above relate to expressions of the form $\bar{y}x$ and $y(x)$, respectively. Their behaviour is described in the previous section.)

The first thing to notice is the use of pattern matching for both selecting the execution branch based on the subtype of the `node` object and extracting the relevant values within it.

Second, from the algorithm one can notice that nodes wrapping silent transitions and message passing are always leaf nodes (the recursion stops on them). And concatenations, compositions and conditionals are always non-leaf nodes (the recursion might continue from them). The former are actual actions and the latter mostly provide structure and order to the execution.

5.1.3 Thread Management

In order to prevent the application from terminating while some agents are still being processed, the runner must delay its own termination until all

threads executing agents are finished. This is mentioned in the algorithm overview above, but a more detailed description of the mechanism is provided here.

Let T be a list of threads, which is empty when `start()` is called. The execution of an agent on a new thread is described below:

```
private function runInNewThread(agent):
    synchronized { numberOfThreads++; notify }
    t = new Thread {
        run(agent)
        synchronized { numberOfThreads--; notify }
    }
    T = T + t
    start t
```

The termination is postponed until the variable $numberOfThreads$ reaches zero. In order to avoid unnecessary CPU usage, the thread responsible for verifying this condition only executes when the number of threads is changed.

```
private function waitThreads():
    synchronized {
        while (numberOfThreads != 0) wait;
    }
```

5.1.4 Thread Spawning

Producing threads is an expensive process. Although this cost is negligible in long-lived agents, it takes a quite significant share of the life time of processes that exist for only a few hundred milliseconds. If too many of these short-lived agents are spawned in a short amount of time, the runner's performance may be severelly harmed.

The cost of new agents, still, can be minimized by the reuse of threads. The Java platform provides a cached thread pool[5], which can be used to start threads. When a new thread is requested, the pool will reuse one of the cached threads, if one is available; otherwise it will produce a new one. Finished threads are cached and, if not used within 60 seconds, discarded (so memory can be released).

In a small experiment, 100,000 agents were produced, both with and without thread reuse. These produced agents were composed of a single, empty, silent action. The following table summarizes the obtained results[6].

[5] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Executors.html

[6] On a Core2 Duo 1.8Ghz with 3GB of RAM.

Mechanism	Consumed Time
new Thread	23.743 seconds
CachedThreadPool	2.089 seconds

5.1.5 Regarding Composition

Processes in a parallel composition perform can evolve both independently of other processes and by interacting with other processes.

Communication In a situation where two agents P and Q execute in parallel, attempting to communicate to each other through a common channel, the message passing mechanism described ahead will take care of the thread locking and name exchanging. Once this is performed, the agents are allowed to procced. In short: $\bar{y}x.P|y(x).Q \xrightarrow{\tau} P|Q$.

Parallel Execution Any agent P attempting to perform a silent transition will perform it without interference from any other agents that may be executing in parallel. In short: $\tau.P|Q \xrightarrow{\tau} P|Q$.

5.1.6 Usage

Actual usage of the runner is very simple and, once the desired agent is produced, it requires no more than a line a code.

```
val agent = Agent { ... }
new ThreadedRunner(agent).start
```

6 Common Patterns

This chapter introduces some patterns that can be used in order to perform common tasks.

6.1 Function calls

Functions can be encoded by processes with arguments augmented with an extra parameter: a channel for returning the computed value. The function call is performed by executing, in parallel, both this process (filled with the relevant arguments) and the process that needs the computed value (guarded by the link through which the return value will be sent).

The following code illustrates this technique.

```
def sum(p1:Int, p2:Int, returnLink:Link[Int]):Agent = Agent {
    returnLink~Name(p1+p2)
}

val P = Agent {
    val link = Link[Int]
    val retv = Name[Int]
    val prnt = Action{println(retv.value)}
    link(retv)*prnt | sum(5, 10, link)
}
```

6.2 Mutual Exclusion

Mutual exclusion can be implemented by using an extra process to control the access to the critical section. This process can be of the form:

$C \stackrel{\text{def}}{=} m(n).n().C$

The entry protocol for clients is to send any link n through m. The exit protocol is to send any message (even an empty one) through n.

This works because π-calculus links are synchronous. The controller waits for agents trying to enter the critical section in $m(n)$. When one comes in, the controller will wait in $n()$ until the agent releases the lock by sending a message. Meanwhile, any other agents trying to get in will be blocked.

The following code illustrates this technique.

```
var shared = 0
val mutex = Link[Link[Any]]
```

```
lazy val controller:Agent = Agent {
    val link = Name[Link[Any]]
    mutex(link)*link()*controller
}

lazy val P:Agent = Agent {
    val act = Action { shared += 1 }
    val pvt = Link[Any]
    mutex~pvt*act*pvt~()*P
}
```

6.3 Synchronization Barrier

A synchronization barrier for an arbitrary number of processes can be implemented by using an extra controller agent. Given a shared link `barrier` of type `Link[Link[Any]]`, this controller process, listed below, will receive and store links until the desired number of held agents is reached. Then, it will a send empty messages through each of these links in order to notify the opening of the barrier.

```
def controller(n:Int, channel:Link[Link[Any]]) =
   controllerHold(n, channel, Nil)

def controllerHold(n:Int, channel:Link[Link[Any]], agentList:
   List[Link[Any]]):Agent = Agent {
    val newAgent = Name[Link[Any]]

    If (agentList.size < n) {channel(newAgent) *
       controllerHold(n, channel, newAgent :: agentList) } |
    If(agentList.size >= n) {controllerRelease(agentList) }
}

def controllerRelease(agentList:List[Link[Any]]):Agent =
   Agent {
    agentList.first~() * If (!agentList.tail.isEmpty) {
       controllerRelease(agentList tail) }
}
```

Clients of this barrier must then send an private link to the controller (through the `barrier` channel) and, then, wait on this same link until they receive an empty message, which signals the opening of the barrier.

7 Proof of Concept

For the purpose of demonstrating the viability of Pistache as a platform for developing concurrent applications, some small programs were written. These programs are presented in this chapter.

7.1 Client-Server

This is the implementation of the client-server example presented in section 2.3, in which three agents (C, S and P) communicate in order to print a message. The pen-and-paper π-calculus version is repeated here for convenience:

$C = (\nu p)(\nu x)a(p).\bar{p}x$
$S = \bar{a}b.S$
$P = (\nu y)b(y).P$

The Pistache version below presents the same mechanics. The `Server` agent (S) will send to the `Client` agent (C) a link to the `Printer` agent (P). The client will then send a message to the printer, which will display it on the screen.

```
object Printserver {
    def main (args:Array[String]) {

        val sl = Link[Link[String]]
        val pl = Link[String]

        val Client = Agent {
            val l = Name[Link[String]]
            sl(l) * l~"Hello, world!"
        }

        lazy val Server:Agent = Agent {
            sl~pl*Server
        }

        lazy val Printer:Agent = Agent {
            val msg = Name[String]
            val act = Action { println(msg.value) }
            pl(msg) * act * Printer
        }
```

```
        new ThreadedRunner(Client | Server | Printer) start
    }
}
```

It's worth noting the explicit declaration and typing of names (including the communication links), which isn't required in pure π-calculus.

7.2 HTTP Server

A tiny HTTP Server, capable of serving files in the execution directory, was written and is presented here in a simplified form.

First, this application needs to perform communication through TCP sockets. The class below takes, as constructor arguments, a socket and two π-calculus links. It has a method that returns an agent which will behave like an interface between the application and the socket.

```
class PiSocket(socket:Socket, send:Link[String], recv:Link[
   String]) {

    private val wBuf = Name[String]
    private val rBuf = Name[String]

    private val writeToSocket = Action { send value of wBuf
       through socket }

    private val readFromSocket = Action { receive a line from
        socket and store on rBuf }

    private val closeSocket = Action { close socket }

    def agent() = Agent {
        lazy val writer:Agent = send(wBuf) * (If (wBuf !=
           null) {writeToSocket * writer} | If (wBuf == null)
            {closeSocket})
        lazy val reader:Agent = readFromSocket * recv~rBuf *
           reader
        writer | reader
    }
}
```

Given a socket and two links (e.g. `send` and `recv`), the call bellow will return the desired agent:

```
new PiSocket(socket, send, recv).agent
```

This agent will read a message from the `send` link, and send this message through the socket if it is not null (otherwise, the socket will be closed). Similarly, it will receive a message from the socket and send it through `recv`. Therefore, an second agent can send a message through the socket by sending it through the `send` link, and receive messages from the socket through the `recv` link.

Now, having this class, the actual server can be presented. It is composed of a few agents:

`serverAgent`	Accepts incoming connections and spawns other agents to handle them
`handlerAgent`	Spawns an agent for interfacing with the socket and an agent to receive the HTTP request
`loop`	Receives the request and, depending on its validity, continues as one of the agents below
`okAgent`	Sends a header reporting success followed by the requested file
`errAgent`	Sends a header reporting failure

```
def main(args:Array[String]) {

    val serverSocket = new ServerSocket(8080)

    lazy val serverAgent:Agent = Agent {
        var requestSocket:Socket = null
        val accept = Action { requestSocket = serverSocket.
            accept }
        accept * (serverAgent | handlerAgent(requestSocket))
    }

    def handlerAgent(socket:Socket) = {
        val send = Link[String]
        val recv = Link[String]
        var fileData:String = null
        val requestSocketAgent = new PiSocket(socket, send,
            recv).agent

        lazy val loop:Agent = Agent {
            val buffer = Name[String]

            val parse = Action {
```

```
                parse request and fill the contents of the
                    requested file in fileData
            }

            recv(buffer) * parse * (
                If (buffer.value != null) {loop} |
                If (buffer.value == "" && fileData != null) {
                    okAgent(send, fileData)} |
                If (buffer.value == "" && fileData == null) {
                    errAgent(send)})
        }

        requestSocketAgent | loop
    }


    def okAgent(send:Link[String], extension:String, data:
        String) = Agent {
        send~HEADER_OK * send~data * send~null
    }

    def errAgent(send:Link[String]) = Agent {
        send~HEADER_ERR * send~null
    }

    new ThreadedRunner(serverAgent).start
}
```

Of course, the code above is lacking many details, including error handling and the relevant imports. The full code, still, is available online[7].

[7] http://bitbucket.org/pmatiello/pihttpd

8 Conclusion

Process calculi were developed to express and reason about sets of independent processes and their interactions through mechanisms of message-passing.

The π-calculus is a relatively recent member of this family. Although originally developed as a specification language, a number of implementations were created (a few of them being presented earlier in this document). These implementations have demonstrated the feasibility of the π-calculus concurrency model, not only as a formal specification tool, but also in actual software programming.

Pistache is, then, yet another π-calculus implementation. It is very similar to PiLib, being written as a domain-specific language hosted in the general-purpose language Scala. Still, both implementations play different roles as PiLib offers a more fluid and comfortable interface for programming while in Pistache these conveniences are sacrificed to keep it syntactically closer to the pen-and-paper calculus.

In regard to Kroc and Pict, there are differences beyond the programming interface. Pistache is hosted in Scala and, in consequence, exists within the Java ecosystem. This is a very broad and rich environment, and allows for the luxury of integrating with a number of other libraries.

The small HTTP Server presented as proof of concept application upholds the stated perception that the π-calculus can contribute to the development of real programs. The simple and expressive mechanics of the calculus, arguably, made the task easier by providing adequate abstractions for reasoning about the problems in question.

Also, the treatment of formal expressions as executable code narrows the gap between specifications and implementations, and does so without the use of automatic code generation tools.

References

[CO03] Vincent Cremet and Martin Odersky. PiLib: A Hosted Language for Pi-Calculus Style Concurrency. In *Proceedings of Domain-Specific Program Generation: International Seminar*, Lecture Notes in Computer Science (LNCS), 2003.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 1978.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[MPW89] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I. *I and II. Information and Computation*, 100, 1989.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.

[Par01] Joachim Parrow. An Introduction to the pi-Calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1997.

[SH09] Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers. `http://www.scala-lang.org/node/198`, 2009.

[WB05] P.H. Welch and F.R.M. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*. Springer Verlag, 2005.