Pilib: using π-calculus in Scala

Vincent Cremet

4th March 2004

Abstract

pilib is a library written in Scala that implements the concurrency constructs of the π-calculus. The “extensible” syntax of Scala makes it possible to use almost the same syntax as in the π-calculus. The huge advantage of this library with respect to any simple π-calculus interpreter is that we can transmit any value along names, we can get control over them using the type system, and we can use any existing Java class.

1 Getting started

Here is how you can write in Scala a generic two-place buffer based on π-calculus primitives using the library pilib.

```
def Buffer[a](in: Chan[a], out: Chan[a]): Unit = {

  def B0: Unit = {
    val x = in.read;
    B1(x)
  }

  def B1(x: a): Unit = choice (
    out(x) * (B0) +
    in * (y => B2(x, y))
  );

  def B2(x: a, y: a): Unit = {
    out.write(x);
    B1(y)
  };

  B0
}
```

If we have now a producer agent and a consumer agent that have the following signature:

```
def Producer(l: Chan[String]): Unit;
def Consumer(r: Chan[String]): Unit;
```

We can link them together and execute them in parallel with the expression:

```
spawn < Producer(in) | Buffer(in, out) | Consumer(out) >
```


where *in* and *out* are fresh channels previously created using:

```
val in = new Chan[String];
val out = new Chan[String];
```

2 Primitive constructs for concurrency in Scala

The concurrency model of Scala is inspired by Java. It consists of a class Monitor and a function fork defined in the module concurrency.

Here is the interface of class Monitor:

```
class Monitor with {
  def synchronized[a](def p: a): a;
  def wait(): Unit;
  def wait(timeout: Long): Unit;
  def notify(): Unit;
  def notifyAll(): Unit;
}
```

And here is the signature of function fork:

```
def fork(def process: Unit): Unit;
```

The module concurrency contains also a construction spawn that allow to fork several threads at the same time. For instance if you want to execute concurrently the expressions P, Q and R, each one of type Unit, you just have to type

```
spawn < P | Q | R >
```

3 π-calculus encoding in Scala

3.1 Syntax and semantics

Action prefix	π	$::=$	$x(y)$	receive y along x
		$\mid$	$\bar{x}\langle y\rangle$	send y along x
Guarded process	G	$::=$	$\pi.P$	
Process	P	$::=$	$\sum_{i>0} G_i$	summation
		$\mid$	0	empty sum
		$\mid$	$P_1 \mid P_2$	composition
		$\mid$	$\nu a.P$	restriction
		$\mid$	$A\langle x_1, ..., x_n\rangle$	agent
Agent definition	D	$::=$	$A(x_1, ..., x_n) = P$	
Term	t	$::=$	$D_1, ..., D_n \vdash P$	

The semantics is given by means of a structural congruence relation together with reduction rules (see Milner's book page 89). The only difference is that we use recursive agents instead of the replication construct. So we have to add the equivalence rules $A\langle y_1, ..., y_n\rangle \equiv P[\bar{y}/\bar{x}]$ for each agent definition $A(x_1, ..., x_n) = P$ and names $y_1, ..., y_n$.

3.2 Translation scheme of a π-term in Scala.

The translation of a π-term is a Scala expression which has type Unit. It is compositional.

Guarded process

$$
\begin{array}{lcl}
[\![x(y).P]\!] & = & x \ * \ (y \Rightarrow [\![P]\!]) \\
[\![\bar{x}\langle v\rangle.P]\!] & = & x(v) \ * \ ([\![P]\!])
\end{array}
$$

Process

$$
\begin{array}{lcl}
[\![\pi_1.P_1 + ... + \pi_n.P_n]\!] & = & \text{choice (} [\![\pi_1.P_1]\!] + ... + [\![\pi_n.P_n]\!] \text{)} \\
[\![0]\!] & = & () \\
[\![P_1 \mid ... \mid P_n]\!] & = & \text{spawn} < [\![P_1]\!] \mid ... \mid [\![P_n]\!] > \\
[\![\nu a.P]\!] & = & \{ \text{ val } a = \text{new Chan[T]; } [\![P]\!] \ \} \\
[\![A\langle x_1, ..., x_n\rangle]\!] & = & A(x_1, ..., x_n)
\end{array}
$$

Agent definition

$$
\begin{array}{lcl}
[\![A(x_1, ..., x_n) = P]\!] & = & \text{def } A(x_1, ..., x_n)\text{: Unit} = [\![P]\!]
\end{array}
$$

Term

$$
\begin{array}{lcl}
[\![D_1, ..., D_n \vdash P]\!] & = & \{ \ [\![D_1]\!]; \ ... \ [\![D_n]\!]; \ [\![P]\!] \ \}
\end{array}
$$

Remarks: As we allow in the Scala implementation to transmit any value along channels and that we want to give types to channels we introduce some changes w.r.t. the π-calculus. First the argument of an output can be any expression which evaluates to a value whose type is the same as the channel's one. Finally when we create a new name we give explicitly its type.

Also we add a timeout in the unobservable action τ

This translation requires only to import the module concurrency which contains the definition of the method spawn, and the module pilib which contains definitions for the method choice and the class Chan. So it is necessary to add the following lines at the beginning of your Scala file.

```
import pilib;
import concurrency;
```

Shortcuts

There are several short-cuts which can be used in order to get more readable code. They are used mainly when we want to translate a process which is a summation consisting of a unique guarded process.

$$
\begin{array}{lcl}
[\![x(y).P]\!] & = & \{ \text{ val } y = \text{x.read; } [\![P]\!] \ \} \\
[\![\bar{x}\langle v\rangle.P]\!] & = & \{ \ x.\text{write}(v); \ [\![P]\!] \ \}
\end{array}
$$

The expressions `x.read` and `x.write(v)` are respectively synchronous input and output calls. The former will wait for an output on the name x then returns the transmitted value. The latter will wait for an input on the name x and then will transmit the value v.

Note that a call to the function choice in a thread will also block the thread as long as one prefix is not triggered and the associated continuation is not finished.

Unobservable action

In π-calculus compared to pilib there is an additional prefix τ called unobservable action. In fact it is only useful as a technical tool for proving equivalence between two processes using bisimulation. From a programming point of view we know that $\tau.P + Q$ is equivalent to $\nu a.(a \mid \bar{a}.P + Q)$ and we can use the latter form.

In pilib there is the construction tau(timeout) * ($[\![P]\!]$) that can be used in a summation and which is based on the latter form. A timeout of 0 gives a guarded process equivalent to $\tau.P$, otherwise it prevents the guarded process from executing during a certain amount of time.

Remark on non-determinism

If we want to use the π-calculus as a programming language we must not try to simulate non-determinism.

In the current implementation of the π-calculus, a summation like $\pi_1.P_1 + ... + \pi_n.P_n$ continues with one of the processes P_i as soon as there comes another summation with which it can performs a communication step.

In order to have real non-determinism, i.e. to have the same probability for each branch $\pi_i.P_i$ to be chosen, we would have to wait an infinite amount of time in order to give to each branch the chance of a matching summation to arrive.

To compensate this lack, there exists in the module pilib a function called either which can be used instead of choice. The expression $either(\pi_1.P_1 + ... + \pi_n.P_n)$ is simply equivalent to some branch $\pi_i.P_i$ where i has been chosen randomly.

Remark on guards [x = y]

In some variants of the π-calculus there are guarded processes like $[x = y].P$ and the new reduction rule

$$[x = x].P + Q \quad \rightarrow \quad P$$

Usually it is used to express a kind of switch statement where each case is exclusive, like $[x = c_1].P_1 + [x = c2].P_2$. In this case, we can simply use in Scala the conditional construct:

```
if (x == c1) P1 else if (x == c2) P2
```

4 Value passing CCS encoding in Scala

Value passing CCS is a process calculus which has the same synchronization model as the π-calculus, based on a synchronous communication between a process that is doing an input and another process that is doing an output. But in CCS contrary to the π-calculus the channels are not allowed to transmit channels.

CCS being a subset of π-calculus we can use the library pilib as it is for encoding CCS. Of course we can then not rely on the Scala type system to signal the use of a channel as a transmitted value.