# PiLib: A Hosted Language for Pi-Calculus Style Concurrency

Vincent Cremet, Martin Odersky

École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
{vincent.cremet, martin.odersky}@epfl.ch

**Abstract.** PiLib is a library written in Scala that implements the concurrency constructs of the $\pi$-calculus. Some features of the programming language Scala, examined in the paper, make it possible to use almost the same syntax as in the $\pi$-calculus. The advantages of this library with respect to a simple $\pi$-calculus interpreter are that we can transmit any value along names, we can get control over them using the type system, and we have access to the full power of Scala in terms of expressiveness and libraries.

## 1 Introduction

Support for concurrency is now almost universal in general purpose programming languages. But the supported concepts vary widely from one language to the next. For example, Concurrent ML [1] is based on an event-based model for process synchronization. Occam [2] is a programming language based on CSP [3]. In a similar way, Pict [4] is based on the $\pi$-calculus [5]. Facile [6] also uses $\pi$-calculus style channel-based communication. JoCaml [7], Funnel [8], and Polyphonic C# [9] use synchronization patterns from join calculus [10]. Erlang [11] is based on an actor model where processes interpret messages in their mailboxes by pattern matching. Oz [12] uses logic variables for process communication. Id [13] and Concurrent Haskell [14] use a form of mutable variables called M-structures [15].

In contrast, most mainstream languages are based on a shared memory thread model. Recent popular languages such as Java [16] or the .NET common language runtime [17] augment this model with thread synchronization based on the Monitor concept [18,19].

In spite of this confusing variety, some trends can be observed. A large body of theoretical research in concurrency is based on CCS or its successor, the $\pi$-calculus. The same holds for many specifications of concurrent systems. A $\pi$-calculus specification can be an executable program. Nevertheless, most concurrent systems are still implemented with shared-memory threads, which are synchronized with semaphores or monitors. Such programs are often much harder to reason about and verify than $\pi$-calculus specifications. But they are also often more efficient to implement.

Given this situation, it seems desirable to have a wide spectrum of programming solutions in which one can move easily from a high-level specification and prototype to a (thread-based) low-level implementation. One way to achieve this is with a programming language which has both low-level and high-level concurrency constructs. However, such a language would tend to be quite large. Moreover, in the light of the great variety of high-level solutions that have been proposed it seems difficult to pick a concrete high-level syntax with confidence that it is the "right one".

A better solution is to express the high-level language as a library abstraction in terms of the low-level one. Then the high-level language becomes, in effect, a domain-specific language where the domain is process coordination. This approach is similar to the use of skeletons in parallel programming [20,21].

Being implemented as a library, the high-level language is embedded (or: hosted) in the low-level language. This has the advantages that no separate compiler and run-time environment needs to be designed, and that all of the facilities of the host-language can be used from the high-level language. However, if the high-level language is to be convenient to use, the host language needs to have a fair degree of expressiveness.

In this paper, we describe our experience with such an embedding. We have developed a process coordination language that is closely modeled after the $\pi$-calculus and that is implemented as a library in the host language Scala [22]. Scala is a new functional/object-oriented language that interacts smoothly with Java and C#. Compared to these environments, Scala has several additional language features, which make it suitable as a host language for domain specific languages. Among others, it supports the following concepts.

- A rich type system, with generics as well as abstract and dependent types.
- Object composition using mixin-style multiple inheritance.
- Named as well as anonymous functions as first-class values that can be nested.
- Pure object orientation, in the sense that every value is conceptually an object and every operator is a method call.

Scala is designed to operate in a JVM or .NET environment, so it can be regarded as an extension language for Java or C#. Scala does not have any constructs dealing with concurrency in the language proper. Instead, it re-uses the concurrency constructs of the underlying environment. These constructs are almost the same for Java and .NET – they consist in each case of a class-based thread model with monitors for synchronization.

The need to define a high-level process model for Scala came up when we taught a class on concurrency theory and concurrent programming. We wanted to be able to have students program concurrent systems that closely follow specifications in CCS and the $\pi$-calculus. At the same time, we wanted to relate this to Java's classical thread/monitor model of concurrent programming. Embedding the high-level concurrent language in Scala provided an elegant means of going from specification to (classical) implementation and at the same time made available the rich facilities of the Java environment.

Implementing a high-level concurrent language as a library abstraction poses some challenges for the syntactic and type system abilities of the host language. In this paper we show how a convenient syntax and type discipline for the high-level language can be achieved with the help of Scala's constructs for abstraction and composition. Particularly important in this respect are Scala's generic type system, its ability to have arbitrary operators as methods, the dynamic, receiver based interpretation of such operators, as well as its light-weight syntax for nestable anonymous functions (i.e., closures).

The rest of this paper is organized as follows. Section 2 is a short introduction to the $\pi$-calculus. Section 3 allows the reader to appreciate the close link between PiLib and the $\pi$-calculus by comparing how a small example can be expressed in both formalisms. In Section 4 we present the elements of the PiLib language in detail. Section 5 shows how these elements are mapped to Scala constructs. Section 6 recapitulates the languages features necessary for the embedding. Section 7 gives a brief description of PiLib's implementation, and Section 8 concludes.

## 2 The $\pi$-calculus in short

The $\pi$-calculus is a model of concurrent computation in which processes can connect and disconnect dynamically by exchanging the names of the channels they use to communicate. A calculus can be seen as a programming language with limited essential concepts, in order to permit formal reasoning. So, as for programming languages, the precise definition of the $\pi$-calculus includes a description of the syntax of the terms that compose the language as well as a definition of their meaning.

### 2.1 Syntax

There are various versions of the $\pi$-calculus. The one we present here corresponds to our implementation in PiLib. For the readers that know the other versions, we consider in our case a monadic $\pi$-calculus without *match* nor *mismatch* operator and where each process in a sum is guarded by an input or output prefix.

Here is the inductive definition of our set of $\pi$-calculus processes.

**Processes**

| | | |
|---|---|---|
| $P, Q ::= \sum_{i=0}^{n} G_i$ | | Sum ($n \geq 0$) |
| | $\nu x.P$ | Restriction |
| | $P \mid Q$ | Parallel |
| | $A(y_1, \ldots, y_n)$ | Identifier |

**Guarded processes**

| | | |
|---|---|---|
| $G ::= x(y).P$ | | Input |
| | $\overline{x}\langle y \rangle.P$ | Output |

**Definitions**

$$A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P \text{ (where } i \neq j \Rightarrow x_i \neq x_j\text{)}$$

A *process* in the $\pi$-calculus is either a choice $\sum G_i$ of several guarded processes $G_i$, or a parallel composition $P \mid Q$ of two processes $P$ and $Q$, or a restriction $\nu x.P$ of a private fresh name $x$ in some process $P$, or a process identifier $A(y_1, \ldots, y_n)$ which refers to a (potentially recursive) process definition. In a sum, processes are guarded by an input action $x(y)$ or an output action $\overline{x}\langle y \rangle$, which receives (respectively sends) a channel $y$ via the channel $x$ before the execution of the two communicating processes continues.

### 2.2 Structural equivalence

A first way of assigning a meaning to processes is to define an equivalence relation between them. For instance identifying processes which differ only by the order of parallel sub-processes is a means of giving a meaning to the $\mid$ operator. The equivalence that we define in this section is structural in the sense that it is based only on the syntax of processes. This is in contrast to behavioral equivalences which try to identify programs that have similar behaviors.

The structural equivalence is defined as the smallest congruence that contains a set of equalities. The more interesting equalities are the scope extrusion rules (Extr-Par, Extr-Sum) that allow to pull a restriction at the top-level of a process, and the rule (Unfold) which replaces the occurrence of an agent name with the body of its definition:

$$\textsc{Extr-par}\ \frac{}{P \mid \nu x.Q \equiv \nu x.(P \mid Q)}\ x \notin \mathrm{fn}(P)$$

$$\textsc{Extr-sum}\ \frac{}{P\ +\ \nu x.Q \equiv \nu x.(P\ +\ Q)}\ x \notin \mathrm{fn}(P)$$

$$\textsc{Unfold}\ \frac{A(\widetilde{x}) \stackrel{\text{def}}{=} P}{A(\widetilde{y}) \equiv P[\widetilde{y}/\widetilde{x}]}$$

In the first two equalities, $\mathrm{fn}(P)$ is the set of free names of the process $P$, i.e. the names that are not bound through the parameter of an input guarded process or through the restriction operator. In the third equality, $\widetilde{x}$ represents a sequence $x_1, \ldots, x_n$ of names and $P[\widetilde{y}/\widetilde{x}]$ denotes the substitution of $y_i$ for every occurrence of $x_i$ in $P$.

Beside these three equalities, there are also rules which identify processes that differ only in the names of bound variables, that declare $+$ and $\mid$ to be associative and commutative, and that allow permuting restriction binders $\nu x$.

### 2.3 Operational semantics

The standard semantics of the $\pi$-calculus is given by a labelled reduction relation. Labels associated to reduction steps are needed to define bisimulation, an equivalence relation that identifies processes that can simulate themselves

mutually. But to understand the computational content of a $\pi$-calculus term a reduction relation without labels is simpler. We therefore define $\rightarrow$ to be the smallest relation that satisfies the following inference rules:

$$\textsc{Par}\ \frac{P\ \rightarrow\ P'}{P \mid Q\ \rightarrow\ P' \mid\ Q}$$

$$\textsc{Nu}\ \frac{P\ \rightarrow\ Q}{\nu x.P\ \rightarrow\ \nu x.Q}$$

$$\textsc{Struct}\ \frac{P \equiv P' \qquad P'\ \rightarrow\ Q' \qquad Q' \equiv Q}{P\ \rightarrow\ Q}$$

$$\textsc{Com}\ \frac{}{a(x).P + P' \mid \overline{a}\langle y\rangle.Q + Q'\ \rightarrow\ P[y/x] \mid Q}$$

The rule Par tells that a process can evolve independently of other processes in parallel. The rule Nu is a contextual rule that makes it possible to reduce under a restriction. The rule Struct is the one that allows to identify equivalent processes when considering reduction. Finally, rule Com is the key to understand the synchronization mechanism of the calculus: two parallel processes that can perform complementary actions, i.e. send and receive the name of some channel along the same channel can communicate. They proceed next with the guarded processes associated to the involved send and receive actions. Note that the name $x$ potentially free in the continuation $P$ of the input guarded process $a(x).P$ is bound to the transmitted name $y$ thanks to the substitution $[y/x]$ applied to $P$.

## 3 Example: The Two-place Buffer

### 3.1 Specification in $\pi$-calculus

The specification of a two-place buffer in $\pi$-calculus consists of a set of mutually recursive process definitions:

$$\begin{aligned}
&\mathit{Buffer}(\mathit{put}, \mathit{get}) = B_0(\mathit{put}, \mathit{get})\\
&B_0(\mathit{put}, \mathit{get}) = \mathit{put}(x).B_1(\mathit{put}, \mathit{get}, x)\\
&B_1(\mathit{put}, \mathit{get}, x) = \overline{\mathit{get}}\langle x\rangle.B_0(\mathit{put}, \mathit{get})\ +\ \mathit{put}(y).B_2(\mathit{put}, \mathit{get}, x, y)\\
&B_2(\mathit{put}, \mathit{get}, x, y) = \overline{\mathit{get}}\langle x\rangle.B_1(\mathit{put}, \mathit{get}, y)
\end{aligned}$$

The definitions $B_0$, $B_1$ and $B_2$ represent respectively the state of the buffer when it contains zero, one or two elements, the buffer is initially empty. All the definitions are parameterized by the names of the channels used to put ($\mathit{put}$) or get ($\mathit{get}$) an element. Additionally definitions $B_1$ and $B_2$ are parameterized by the names of the elements stored in the buffer.

The logic of these definitions is then very intuitive: if the buffer is empty (state $B_0$) it can only accept the input of an item on channel $\mathit{put}$ and move to

state $B_1$. If it contains exactly one item (state $B_1$), it can either output this item and go back to state $B_0$ or accept a new item and move to state $B_2$. Finally when the buffer contains two items it can only output the first stored item and go to state $B_1$.

$put(x).B_1(put, get, x)$ and $\overline{get}\langle x\rangle.B_0(put, get)$ are two instances of *guarded processes*. The first one is an input guarded process which binds an input item on channel $put$ to the name $x$ in the continuation process $B_1(put, get, x)$. The second one is an output guarded process which outputs the item $x$ on channel $get$ before continuing with process $B_0(put, get)$.

### 3.2 Implementation with PiLib

Because PiLib is modeled after $\pi$-calculus concurrency primitives, it is possible to reproduce the specification above almost as it is:

```
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
}
```

Process definitions are replaced by function definitions (introduced by **def** in Scala) with result type unit. This type contains just one value and corresponds to void in C or Java. The buffer is parameterized by the type $a$ of the stored items. Chan[$a$] is the type of channels carrying objects of type $a$. The $\pi$-calculus input guarded process $put(x).B_1(put, get, x)$ is now written put $*$ { x ⇒ B1(x)}, and the output guarded process $\overline{get}\langle x\rangle.B_0(put, get)$ is now written get(x) $*$ B0 (the character $*$ replaces the dot symbol). A $\pi$-calculus summation + between several alternatives is now a sequence of guarded processes separated by a comma and enclosed in choice ( ... ).

Compared to the specification in $\pi$-calculus, the implementation using PiLib has several advantages. First the channels have a type and the typechecker phase of the compiler ensures that only objects of the right type are passed to channels. Second, since processes in the $\pi$-calculus are modeled as functions, it is possible to hide the internal states $B_0$, $B_1$ and $B_2$ by nesting the corresponding functions inside the function Buffer. Because the scope of parameters $put$ and $get$ extends to these local functions they can refer to these parameters implicitly.

### 3.3 Using the buffer

Here is an example of using the buffer in $\pi$-calculus, where $Producer$ is a process that repeatedly creates a new channel and put it in the buffer, and $Consumer$

is a process that tirelessly gets an item from the buffer and discards it.

$$Producer(put, get) = \nu\, x.\, \overline{put}\langle x\rangle.Producer(put, get)$$
$$Consumer(put, get) = get(x).Consumer(put, get)$$

$$\nu\, put, get.\ Producer(put, get) \mid Buffer(put, get) \mid Consumer(put, get)$$

The three processes are put in parallel using the operator | and are linked together through the sharing of the fresh channels *put* and *get* introduced by the restriction operator $\nu$.

Contrary to PiLib whose channels can carry arbitrary objects, in the $\pi$-calculus everything is a name. So both channels and the values they carry are taken from the same domain. A way of typing such channels in PiLib is to use a recursive type definition Channel to represent $\pi$-calculus channels:

```
class Channel extends Chan[Channel];
```

Such a definition can be read: "A $\pi$-calculus channel is a channel that can carry other $\pi$-calculus channels".

Using this type definition, the $\pi$-calculus code above has now an exact counterpart in PiLib:

```
def Producer(put: Channel, get: Channel): unit = {
  val x = new Channel;
  choice ( put(x) * Producer(put, get) )
}
def Consumer(put: Channel, get: Channel): unit =
  choice ( get * { x => Consumer(put, get) } );

val put = new Channel, get = new Channel;
spawn < Producer(put, get) | Buffer(put, get) | Consumer(put, get) >
```

New features of PiLib appear only in the way of creating new channels and executing several processes in parallel using spawn.

As we have seen, the implementation using PiLib is very close to the specification in $\pi$-calculus. It seems as if Scala had special syntax for $\pi$-calculus primitives, however PiLib is nothing more than a library implemented using low-level concurrency constructs of Scala inherited from Java. In the rest of the paper we will try to demystify the magic.

Of course we can also write a two-place buffer using monitors as in Java and the implementation would certainly have been more efficient. But then relating the implementation to the specification would have been completely not trivial. With PiLib we can closely relate the specification language and the implementation language, and thereby gain immediate confidence in our program.

## 4 Description of the PiLib grammar

In this section we briefly describe the original concurrency interface of Scala, directly inherited from Java, and then introduce the PiLib interface.

### 4.1 Original concurrency interface

The original concurrency interface of Scala consists mainly of a class Monitor and a function fork with the following signatures.

```
class Monitor {
  def synchronized[a](def p: a): a;
  def wait(): unit;
  def wait(timeout: long): unit;
  def notify(): unit;
  def notifyAll(): unit;
}

def fork(def p: unit): unit;
```

The method synchronized in class Monitor executes its argument computation p in mutual exclusive mode - at any one time, only one thread can execute a synchronized argument of a given monitor.

A thread can suspend inside a monitor by calling the monitor's wait method. At this point, the thread is suspended until a notify method of the same monitor is called by some other thread. Calls to notify with no threads waiting on this monitor are ignored.

The fork method creates a new thread for the execution of its argument and returns immediately.

### 4.2 PiLib interface

Now we present the grammar of PiLib. How this grammar is actually implemented as a library in Scala is the topic of the next section. Here we just present the syntax of PiLib constructions and give an informal description of their associated semantics.

**Channel creation** At the basis of the PiLib interface is the concept of *channel* (also called "name" in the $\pi$-calculus). A channel represents a communication medium. To get an object which represents a fresh channel that can carry objects of type $A$, one simply writes **new** Chan[$A$].

**Guarded processes** Assuming an expression $a$ that evaluates to a channel carrying objects of type $A$, the term $a * \{ x \Rightarrow c\}$ is an *input guarded process* with continuation $c$. References to $x$ in $c$ are bound to the value of the transmitted

object. The type of a guarded process whose continuation has result type $B$ is GP[$B$]. Note that instead of $\{\ x \Rightarrow c\ \}$ we could have used any expression of type $A \Rightarrow B$ (the type of functions from $A$ to $B$).

Similarly an *output guarded process* is written $a(v) * c$ where $v$ is the value to be sent and $c$ is any continuation expression. If the type of $c$ is $B$, the type of the entire guarded process is GP[$B$].

As in the $\pi$-calculus on which it is based, communications in PiLib are synchronous: when a thread tries to output (resp. input) a value on a given channel it blocks as long as there is no other thread that tries to perform an input (resp. output) on the same channel. When two threads are finally able to communicate, through input and output guarded processes on the same channel, the thread that performs the input proceeds with the continuation of the input guarded process applied to the transmitted value, and the thread that performs the output proceeds with the continuation of the output guarded process.

**Summation** The next specific ingredient of PiLib is the method choice which takes an arbitrary number of guarded processes as arguments and tries to establish communication using one of these guarded processes with another thread. The choice method blocks until communication takes place, as explained above. Once the communication takes place, the guarded processes among the arguments to the function choice that do not take part in the communication are discarded.

**Forking multiple threads** The construction spawn is used to fork several threads at the same time. For instance to execute concurrently some expressions $p$, $q$, $r$, each one of type unit, one writes

spawn $< p \mid q \mid r >$

The figure 1 is a summary of the concurrency primitives provided by PiLib with their typing rules. All these typing rules are admissible in the host type system.

### 4.3 Derived constructs

The class Chan[$A$] also contains methods to perform synchronous read and synchronous write. These constructs can be derived from the more primitive constructs presented so far.

If $a$ is an expression of type Chan[$A$], an example of a synchronous read is:

**val** x = $a$.read;

It is equivalent to (and implemented as)

**var** x: $A$ = **null**;
choice ($a$ * { y ⇒ x = y });

$$\textsc{Nu}\ \frac{}{\text{new Chan}[A] : \text{Chan}[A]}$$

$$\textsc{Input}\ \frac{a : \text{Chan}[A] \qquad x \Rightarrow c : A \Rightarrow B}{a * \{x \Rightarrow c\} : \text{GP}[B]}$$

$$\textsc{Output}\ \frac{a : \text{Chan}[A] \qquad v : A \qquad c : B}{a(v) * c : \text{GP}[B]}$$

$$\textsc{Choice}\ \frac{g_i : \text{GP}[B]}{\text{choice}\ (g_1, \ldots, g_n) : B}$$

$$\textsc{Spawn}\ \frac{p_i : \text{unit}}{\text{spawn}\ < p_1 | \ldots | p_n >: \text{unit}}$$

**Fig. 1.** PiLib constructs

A synchronous write a.write(v) is equivalent to choice (a(v) $*$ {}).

## 5 Desugarization

In this section we will explain how it is possible to implement PiLib as an hosted language, i.e. as a Scala library. We consider each PiLib construction in turn and see how it is interpreted by the language.

### Channel creation

In PiLib, a fresh channel carrying objects of type $A$ is created using the syntax **new** Chan[A]. Indeed, this is the normal Scala syntax for creating instances of the parameterized class Chan.

### Input guarded process expression

Scala allows a one-argument method to be used as an infix operator , i.e. the Scala parser will replace any expression e f g by e.f(g). Furthermore, $*$ is a valid identifier. So an input guarded process

$$a * \{\ x \Rightarrow c\ \}$$

is recognized by the parser as

$$a.*(\{\ \text{x} \Rightarrow \text{c}\ \})\ .$$

In the code above, $a$ is an expression of type Chan[$A$], and $\{\ x \Rightarrow c\ \}$ is an anonymous function of type $A \Rightarrow B$. It works because the class Chan[$T$] contains a polymorphic method named $*$ with the following signature:

**def** $*[U]$(f: $T \Rightarrow U$): GP[$U$];

The anonymous function is itself desugarized as the creation of an anonymous object of type Function1[$A$, $B$] (if $A$ is the type inferred for the parameter $x$ and $B$ the type inferred for the body $c$) with a method apply whose unique parameter is $x$ and whose body is $c$. That is, $\{x \Rightarrow c\}$ is translated to

```
new Function1[A,B] {
  def apply(x: A): B = c;
}
```

### Output guarded process expression

In Scala an expression $a(v)$ where $a$ has type Function1[$T$,$U$] is replaced by $a$.apply($v$). So an expression describing an output guarded process $a(v) * c$ is recognized as $a$.apply($v$).$*$($c$). It implies that the class Chan[$A$] implements the interface Function1[$A$,Product[$A$]] where $A$ is the type of the transmitted value and Product[$a$] is an helper class which contains the method $*$ which is necessary for simulating the dot symbol of the natural syntax.

### Guarded process

The type of an input or output guarded process expression is GP[$A$] where $A$ is the result type of the continuation.

The class GP[$A$] encapsulate the elements that compose a guarded process: the name of the channel and the continuation function for an input guarded process, the name of the channel, the transmitted value and the continuation function for an output guarded process.

### Summation

A summation is written choice ($g_1$, ..., $g_n$) in PiLib. It is just a call to the polymorphic function choice with signature:

**def** choice[$A$](gs: GP[$A$]$*$): $A$:

As specified by the star symbol at the end of the argument type, the argument can be a sequence of guarded processes of arbitrary length.

### Forking of multiple threads

The syntax to fork several threads is spawn $< p \mid q \mid r >$. As we have seen previously, this expression is recognized as spawn.$<$($p$).|($q$).|($r$).$>$. Here is the implementation of this construct:

```
abstract class Spawn {
  def <(def p: unit): Spawn;
  def |(def p: unit): Spawn;
  def > : unit;
}

object spawn extends Spawn {
  def <(def p: unit): Spawn = { fork(p); this }
  def |(def p: unit): Spawn = { fork(p); this }
  def > : unit = ()
}
```

In the code above, fork is the low-level function to fork a sub-process.

The **def** modifier applied to parameters in the methods of the abstract class Spawn specifies that the argument must be passed without being evaluated. It will be evaluated each time it is used in the body of the function. This strategy of evaluation is called *call-by-name*. Without this indication, the different threads would be executed sequentially in the order of evaluation of the arguments of the method.

## 6 Scala constructs used for the embedding

In this section we summarize the constructs that proved to be useful for hosting PiLib in Scala.

### Polymorphism

Type genericity allowed us to parameterize a channel by the type of the objects it can carry. The Scala type system makes sure that a channel is always used with an object of the right type.

In Scala, methods can also be polymorphic. As $\pi$-calculus agents are represented by methods in PiLib, such agents can be polymorphic, like the two-place buffer in the example of section 3.

Type parameters may have bounds in Scala, and these bounds can be recursive. This feature known under the name of *F-bounded polymorphism* can be used to express recursive channels. For instance to define a kind of channel that can carry pairs of integers and channels of the same kind, we would write

```
class C extends Chan[Pair[int, C]]
```

### Higher-order functions and anonymous functions

We already saw that an input guarded process is expressed as a call of the method $*$ with a continuation function as argument. This is an example of a higher-order function. Furthermore it is natural to express the continuation as

an anonymous function because most of the time it is not recursive and does not need to be referred elsewhere in the program.

The type inference performed by the compiler allows to omit some type annotations, like the type of the formal parameter in the continuation of an input guarded process.

#### Syntactic sugar

It is convenient to write x(v) $*$ c instead of x.apply(v).$*$(c). This is permitted by the SCALA parser which perform simple but powerful transformations.

The fact that all functions are members of a class (i.e. methods) allows us to overload operators such as $*$ easily without introducing ambiguity.

#### Call-by-name

The parameter modifier **def** makes it possible in SCALA to explicitly specify that some parameters must be called by name.

For instance it allows us to write an output guarded process as x(v) $*$ c instead of x(v) $*$ { () $\Rightarrow$ c } because the parameter of method $*$ has a **def** modifier. And the construction spawn $<$ p | q $>$ would not have been possible either. We would have been required to replace this by the more cumbersome spawn(() $\Rightarrow$ p, () $\Rightarrow$ q).

#### Sequence type

Another feature of SCALA, which is used in the definition of the choice method, is the possibility for a function to pass an arbitrary number of arguments to methods.

## 7 Overview of the implementation

To explain the implementation we first describe the involved data structures and then present the means by which processes interact.

Two guarded processes are called *complementary* if one is an input and the other is an output on the same channel. For the sake of simplicity, the implementation unifies the concepts of input and output guarded processes: at the time where communication takes place between two complementary guarded processes, each of them will provide a value to the other (the dummy value () for the input guarded process) and apply its continuation to the value it receives (always the dummy value () for the output guarded process). The data structure for a guarded process consists then of four components: a channel, a kind (input or output), the value it provides, and the continuation function to apply to the received value.

A *sum* is an ordered list of guarded processes, together with a state that represents the continuation to execute after communication takes place. At the

creation of a sum, it is not known which of its guarded process will react and which value this guarded process will receive, so it is impossible to give a value to the sum continuation. Two sums are said *complementary* if they contain complementary guarded processes.

The implementation keeps a global list of pending sums, i.e. sums that are waiting to communicate. We call it the *pending list*. An invariant of this list is that the sums it contains are pairwise not complementary. The pending list behaves as a queue, that is, new entries are added at its end.

Now we will explain what happens when a thread calls the function `choice`. The argument of the function choice is a sequence of guarded processes. This sequence is first turned into a sum with an undefined continuation. From now this sum is designated as the *arriving sum*. The pending list is scanned to find a complementary sum. If there is none, the arriving sum is appended at the end of the pending list, the choice function waits until the sum continuation gets a value and then execute this continuation. If there is a compatible sum, it is extracted from the pending list, both communication sums get the value for their continuation and their respected threads are woken up.

The implementation consists of about 150 lines of Scala code with a large part dedicated to the syntactic sugar. The complete source code is available on the web [23].

## 8 Conclusion

We have presented PiLib, a language for concurrent programming in the style of the $\pi$-calculus. The language is hosted as a library in Scala. The hosting provides "for free" a highly higher-order $\pi$-calculus, because all parts of a PiLib construct are values which can be passed to methods and transmitted over channels. This is the case for channels, output prefixes, continuations of input and output guarded processes, and guarded processes themselves. In the following example several guarded processes share the same continuation in a summation:

```
val c = ...;
choice (a(v) * c, b(w) * c)
```

A PiLib program corresponds closely to a $\pi$-calculus specification. Of course, any run of that program will only perform one of the possible traces of the specification. Moreover, this trace is not necessarily fair, because of PiLib's method of choosing a matching sum in the pending list. A fair implementation remain topics for future work.

There is a result of [24,25] showing that it is not possible to implement the mixed choice $\pi$-calculus into an asynchronous language in a distributed (deterministic) way. Our current implementation is centralized and in this case the mixed choice is not problematic. A possible continuation of this work is to implement a distributed version of PiLib, which would force us to abandon the mixed choice.

Our experience with PiLib in class has been positive. Because of the close connections to the $\pi$-calculus, students quickly became familiar with the syntax and programming methodology. The high-level process abstractions were a big help in developing correct solutions to concurrency problems. Generally, it was far easier to develop a correct system using PiLib than using Java's native thread and monitor-based concurrency constructs.

## References


1. John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.
2. INMOS Ltd. *OCCAM Programming Manual*. Prentice-Hall International, 1984.
3. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. Reprinted in "Distributed Computing: Concepts and Implementations" edited by McEntire, O'Reilly and Larson, IEEE, 1984.
4. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
5. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
6. Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
7. Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, October 1999.
8. Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, March 2000.
9. Nick Benton, Luca Cardelli, and Cdric Fournet. Modern concurrency abstractions for C$^\sharp$. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 415–440. Springer-Verlag, 2002.
10. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Principles of Programming Languages*, January 1996.
11. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
12. Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, 1995.
13. Arvind, K.P. Gostelow, and W. Plouffe. The ID-Report: An Asynchronous Programming Language and Computing Machine. Technical Report 114, University of California, Irvine, California, USA, 1978.
14. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 295–308, New York, NY, USA, 1996. ACM Press.

15. Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In John Hughes, editor, *Proceedings Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA*, pages 538–568. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.
16. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Java Series, Sun Microsystems, 2000. ISBN 0-201-31008-2.
17. Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley, 2002.
18. C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
19. Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(6):199–207, 1975.
20. Murray I. Cole. Library for skeletal parallel programming. In Batory, Consel, Lengauer, and Odersky, editors, *Dagstuhl proceedings: Domain-Specific Program Generation*, 2003.
21. Herbert Kuchen. A skeleton library. In Batory, Consel, Lengauer, and Odersky, editors, *Dagstuhl proceedings: Domain-Specific Program Generation*, 2003.
22. Martin Odersky. *Report on the Programming Language Scala*, 2002. `http://lampwww.epfl.ch/scala/`.
23. Vincent Cremet. Pilib. http://lampwww.epfl.ch/∼cremet, February 2003.
24. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–265. ACM, 1997.
25. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. *Mathematical Structures in Computer Science*, To appear.