Pict: A Programming Language based on the Pi-Calculus

Janus Dam Nielsen

[1]Department of Computer Science
University of Aarhus

Mobile computing fall 2004

Outline

π vs. λ

$$\frac{ML, Haskel, Scheme, ...}{\lambda} = \frac{?}{\pi - calculus} \tag{1}$$

The main goals are to implement a high level concurrent language purely in terms of the π - calculus primitives, and communication as the sole mechanism of computation. Furthermore to design a practical type system, combining sub-typing and higher order polymorphism.

Outline

1. Introduction
2. The Language
   - Pict and π
   - Core language
   - The High level
   - An example
3. The type system
   - Types
   - Type safety

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free ($e_1 + e_2$)
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free $(e_1 + e_2)$
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free $(e_1 + e_2)$
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free $(e_1 + e_2)$
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free $(e_1 + e_2)$
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free $(e_1 + e_2)$
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison

Pict is based on π.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressivenes.
- Following restrictions
  - asynchronous
  - choice free $(e_1 + e_2)$
  - no match
  - replicated input
  - No importance for the practical programmer.

A comparison cont'

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

Structural congruence

- Structural congruence

$$(e_1 \mid e_2) \equiv (e_2 \mid e_1) \tag{2}$$

$$((e_1 \mid e_2) \mid e_3) \equiv (e_1 \mid (e_2 \mid e_3)) \tag{3}$$

$$\frac{x \notin FV(e_2)}{((new\ x : T\ e_1) \mid e_2) \equiv (new\ x : T\ (e_1 \mid e_2))} \tag{4}$$

Reduction

- Reduction

$$\frac{\{p \rightarrow v\}\ \textit{defined}}{(x!v \mid x?p = e) \rightarrow \{p \rightarrow v\}(e)} \tag{5}$$

And likewise for replicated input.

$$\frac{\{p \rightarrow v\}\textit{defined}}{(x!v \mid x?*p = e) \rightarrow (\{p \rightarrow v\}(e) \mid x?*p = e)} \tag{6}$$

Reduction proceeds under declaration and parallel composition

$$\frac{e_1 \rightarrow e_2}{(d\ e_1 \rightarrow d\ e_2)} \qquad \frac{e_1 \rightarrow e_3}{(e_1|e_2) \rightarrow (e_1|e_2)} \tag{7}$$

if true then e_1 else $e_2 \rightarrow e_1$ if false then e_1 else $e_2 \rightarrow e_2$

Outline

Values

Val	=	id	variable
		[Label Val . . . Label Val]	record
		Type Val	Polymorphic package
		(rec : T Val)	Rectype value
		String	String Constant
		Char	
		Int	
		bool	
Label	=	empty	anonymous label
		id	Explicit label

Patterns

Pat	=	id : Type	variable
		_ : Type	Wildcard
		id : Type @ Pat	Layered
		[Label Pat ... Label Pat]	record
		{ id < Type } Pat	Package
		(rec : T Pat)	Rectype

Processes

Abs	=	Pat = Proc	Process abstraction
Proc	=	Val ! Val	output atom
		Val ? Abs	input prefix
		Val ?* Abs	ymorphic package
		(Proc | Proc)	Parallel composition
		(Dec — Proc)	Local declaration
		if Val then Proc else Proc	Conditional
Dec	=	new id : Type	Channel creation

Outline

Simple transformations

- Declaration
  (new x_1 ... (new x_n e))
  (d_1 ... d_n e) $\Rightarrow$ (d_1 ... (d_n e))
- parallel composition
  (run e_1 e_2) $\Rightarrow$ (e_1 | e_2)

Simple transformations

- Declaration
  (new x_1 ...(new x_n e))
  $(d_1 \ldots d_n \text{ e}) \Rightarrow (d_1 \ldots (d_n \text{ e}))$
- parallel composition
  $(\text{run } e_1\ e_2) \Rightarrow (e_1 \mid e_2)$

Abstraction

- Process abstraction:
  def f [x,y] = ($x!y$ | $x!y$)
  (def x p = e_1 e_2) $\Rightarrow$ (new x (x?*p = e_1 | e_2))
- Mutually recursive definitions:
  (def $x_1 a_1 \ldots$ and $x_n a_n$) $\Rightarrow$
  (new $x_1 \ldots$ (new x_n ($x_1?{*}a_1 \mid \ldots \mid x_n?{*}a_n \mid$ e)))
- Function abstraction
  def f [a_1 a_2 a_3 r] = r!v
  def f [a_1 a_2 a_3] = v $(|X_1 < T_1 \ldots X_n < T_n | l_1 p_1 \ldots l_n p_n)$:T = v
  $\Rightarrow X_1 < T_1 \ldots X_n < T_n [l_1 p_1 \ldots l_n p_n \; r :! T]$ = r!v
- Anonymous functions:
  $\backslash a \Rightarrow$ (def x a x)

Abstraction

- Process abstraction:
  def f [x,y] = $(x!y \mid x!y)$
  (def x p = e_1 e_2) $\Rightarrow$ (new x (x?*p = e_1 | e_2))
- Mutually recursive definitions:
  (def $x_1 a_1$ ... and $x_n a_n$) $\Rightarrow$
  (new x_1 ... (new x_n (x_1?*a_1 | ... | x_n?*a_n | e)))
- Function abstraction
  def f [a_1 a_2 a_3 r] = r!v
  def f [a_1 a_2 a_3] = v $(|X_1 < T_1 \ldots X_n < T_n | l_1 p_1 \ldots l_n p_n)$:T = v
  $\Rightarrow X_1 < T_1 \ldots X_n < T_n[l_1 p_1 \ldots l_n p_n\ r :!T]$ = r!v
- Anonymous functions:
  $\backslash a \Rightarrow$ (def x a x)

Abstraction

- Process abstraction:
  def f [x,y] = $(x!y \mid x!y)$
  (def x p = e_1 e_2) $\Rightarrow$ (new x (x?*p = e_1 | e_2))
- Mutually recursive definitions:
  (def $x_1 a_1$ ... and $x_n a_n$) $\Rightarrow$
  (new x_1 ... (new x_n (x_1?*a_1 | ... | x_n?*a_n | e)))
- Function abstraction
  def f [a_1 a_2 a_3 r] = r!v
  def f [a_1 a_2 a_3] = v $(|X_1 < T_1 \ldots X_n < T_n | l_1 p_1 \ldots l_n p_n)$:T = v
  $\Rightarrow X_1 < T_1 \ldots X_n < T_n [l_1 p_1 \ldots l_n p_n\ r :! T]$ = r!v
- Anonymous functions:
  $\backslash a \Rightarrow$ (def x a x)

Abstraction

- Process abstraction:
  def f [x,y] = $(x!y \mid x!y)$
  (def x p = e_1 e_2) $\Rightarrow$ (new x (x?*p = e_1 | e_2))
- Mutually recursive definitions:
  (def $x_1 a_1$ ... and $x_n a_n$) $\Rightarrow$
  (new x_1 ... (new x_n (x_1?*a_1 | ... | x_n?*a_n | e)))
- Function abstraction
  def f [a_1 a_2 a_3 r] = r!v
  def f [a_1 a_2 a_3] = v $(|X_1 < T_1 \ldots X_n < T_n|l_1 p_1 \ldots l_n p_n)$:T = v
  $\Rightarrow X_1 < T_1 \ldots X_n < T_n[l_1 p_1 \ldots l_n p_n \ r :!T]$ = r!v
- Anonymous functions:
  $\backslash a \Rightarrow$ (def x a x)

Complex values

(new n c!n)

$$
\begin{array}{rcl}
[[x \rightarrow c]] & = & \text{c!x} \\
[[k \rightarrow c]] & = & \text{c!k} \\
[[(dv) \rightarrow c]] & = & (\text{d } [[v \rightarrow c]]) \\
[[(rec : Tv) \rightarrow c]] & = & (\text{new } c' \ ([[v \rightarrow c']] \mid c'?\text{x} = \text{c!(rec:T x)}))) \\
[[\{T\}v \rightarrow c]] & = & (\text{new } c' \ ([[v \rightarrow c']] \mid c'?\text{x} = c!\{T\} \text{ x}))) \\
[[[l_1 v_1 \ldots l_n v_n] & = & (\text{new } c_1 \ ([[v_1 \rightarrow c_1]] \mid c_1 x_1 = \ldots \\
 & & (\text{new } c_n \ ([[v_n \rightarrow c_n]] \mid c_n x_n = \\
 & & \text{c!}[l_1 x_1 \ \ldots l_n x_n)) \ \ldots))
\end{array}
$$

Named values and application

Named value declaration:

(val p=v e) $\Rightarrow$ (new c ([[$v \rightarrow c$]] | c?p = e))

Application:

$$(\text{v}\ v_1 \ldots v_n)\ [[(v\ |T_1 \ldots T_n|\ l_1 v_1 \ldots l_n v_n) \rightarrow c]]$$

$$\begin{aligned} = \quad & (\text{new}\ c'\ ([[v \rightarrow c']]\ |\ c'?\text{x} = \ldots \\ & (\text{new}\ c_1\ ([[v_1 \rightarrow c_1]]\ |\ c_1?x_1 = \ldots \\ & (\text{new}\ c_n\ ([[v_n \rightarrow c_n]]\ |\ c_n?x_n = \\ & \text{x}!T_1 \ldots T_n[l_1 x_1 \ldots l_n x_n\ \text{c}])) \ldots)))) \end{aligned}$$

Outline

Hello world.

run (print!"hello" — print!"world")

Polymorphism.

```
def print2nd [#X l: (List X) p:/[X /String]] =
    if (null l) then
        print!"Null list"
    else if (null (cdr l)) then
        print!"Null tail"
    else
        print!(p (car (cdr l)))
run print2nd![#Int (cons ¿ 6 8 9 nil) int.toString]
run print2nd![#String (cons ¿ "A" "B" "C" nil) \(s:String) = s]
```

Fibo

```
def fibo[n:Int r:!Int] =
  if (—— (== n 0) (== n 1)) then
    r!1
  else
    r!(+ (fibo (- n 1)) (fibo (- n 2)))
run printi!(fibo 4)
```

Outline

The basics

- Types of channels and of the values they carry.
- Why types?
- Types are useful at ensuring consistent use of channel names and eliminating pattern matching failures.

The basics

- Types of channels and of the values they carry.
- Why types?
- Types are useful at ensuring consistent use of channel names and eliminating pattern matching failures.

The basics

- Types of channels and of the values they carry.
- Why types?
- Types are useful at ensuring consistent use of channel names and eliminating pattern matching failures.

Subtyping

Subtyping on channel types

- Refinements of the channel type ^T.
  !T for output only
  ?T for reading only.
- Natural subtype relation since ^T can be used anywhere one of the other two is used.

Subtyping

Subtyping on channel types

- Refinements of the channel type ^T.
  !T for output only
  ?T for reading only.
- Natural subtype relation since ^T can be used anywhere one of the other two is used.

Recursive types

- Types for recursive data structures like lists and trees.
- Some alternatives.
- We go for the simple one, where “folding” and “unfolding” of recursion must be handled explicitly by the programmer.

Recursive types

- Types for recursive data structures like lists and trees.
- Some alternatives.
- We go for the simple one, where "folding" and "unfolding" of recursion must be handled explicitly by the programmer.

Recursive types

- Types for recursive data structures like lists and trees.
- Some alternatives.
- We go for the simple one, where “folding” and “unfolding” of recursion must be handled explicitly by the programmer.

Polymorphism

- Polymorphic types are supported by means of package values and patterns.
- Polymorphic functions are represented as output channels carrying package values.
- Polymorphism and subtyping is combined by providing an upper bound on each bound type variable in a package value.

Polymorphism

- Polymorphic types are supported by means of package values and patterns.
- Polymorphic functions are represented as output channels carrying package values.
- Polymorphism and subtyping is combined by providing an upper bound on each bound type variable in a package value.

Polymorphism

- Polymorphic types are supported by means of package values and patterns.
- Polymorphic functions are represented as output channels carrying package values.
- Polymorphism and subtyping is combined by providing an upper bound on each bound type variable in a package value.

Type inference

- The core language is explicitly typed, but some type information can be derived from the context.
- The x in c?x=e has type int if c has type^int
- The inferennce algorithm is local in that it only uses the immediate surrounding context to determine the type.

Outline

Type safety

- Conjecture: Evaluation can not fail in well-typed processes.
- Conjecture: Reduction preserves typing.

No proofs, but nice features!

Type safety

- Conjecture: Evaluation can not fail in well-typed processes.
- Conjecture: Reduction preserves typing.

No proofs, but nice features!

Type safety

- Conjecture: Evaluation can not fail in well-typed processes.
- Conjecture: Reduction preserves typing.

No proofs, but nice features!

Conclusion

- Pict a programming language based on π.
- A typesystem for Pict.
- Pict can be implemented efficiently.