Pict: A Programming Language based on the Pi-Calculus

Janus Dam Nielsen

1Department of Computer Science
University of Aarhus

Mobile computing fall 2004
Outline

1 Introduction

2 The Language
   - Pict and $\pi$
   - Core language
   - The High level
   - An example

3 The type system
   - Types
   - Type safety
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.
1. Introduction

2. The Language
   - Pict and $\pi$
     - Core language
     - The High level
     - An example

3. The type system
   - Types
   - Type safety
A comparison

Pict is based on $\pi$.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- Following restrictions
  - asynchronous
  - choice free ($e_1 + e_2$)
  - no match
  - replicated input
  - No importance for the practical programmer.
Pict is based on $\pi$.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- 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 $\pi$.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- Following restrictions
  - asynchronous
  - choice free ($e_1 + e_2$)
  - no match
  - replicated input
  - No importance for the practical programmer.
Pict is based on $\pi$.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- 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 $\pi$.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- Following restrictions
  - asynchronous
  - choice free ($e_1 + e_2$)
  - no match
  - replicated input
  - No importance for the practical programmer.
Pict is based on $\pi$.

- **Extended with primitive values:**
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- **Following restrictions**
  - asynchronous
  - choice free ($e_1 + e_2$)
  - no match
  - replicated input
  - No importance for the practical programmer.
Pict is based on $\pi$.

- Extended with primitive values:
  - booleans
  - integers
  - etc.
  - no change of expressiveness.

- Following restrictions
  - asynchronous
  - choice free ($e_1 + e_2$)
  - no match
  - replicated input
  - No importance for the practical programmer.
### A comparison cont’

<table>
<thead>
<tr>
<th>$\pi$</th>
<th>Pict</th>
<th>Desc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{x}y.0$</td>
<td>$x!y$</td>
<td>asynchronous output</td>
</tr>
<tr>
<td>$x(y).e$</td>
<td>$x?y = e$</td>
<td>input prefix</td>
</tr>
<tr>
<td>$e_1 \mid e_2$</td>
<td>$(e_1 \mid e_2)$</td>
<td>parallel composition</td>
</tr>
<tr>
<td>$\nu(x)e$</td>
<td>(new $x$ e)</td>
<td>channel creation</td>
</tr>
<tr>
<td>!$x(y).e$</td>
<td>$x?^*y = e$</td>
<td>replicated input</td>
</tr>
</tbody>
</table>
Structural congruence

\[ (e_1 \mid e_2) \equiv (e_2 \mid e_1) \]  \hspace{1cm} (2)

\[ ((e_1 \mid e_2) \mid e_3) \equiv (e_1 \mid (e_2 \mid e_3)) \]  \hspace{1cm} (3)

\[ x \notin FV(e_2) \]

\[ ((\text{new } x : T e_1) \mid e_2) \equiv (\text{new } x : T (e_1 \mid e_2)) \]  \hspace{1cm} (4)
Reduction

- Reduction

\[
\{ p \rightarrow v \} \text{ defined } \quad \frac{(x!v \mid x?p = e) \rightarrow \{ p \rightarrow v \}(e)}{(x!v \mid x?\ast p = e) \rightarrow (\{ p \rightarrow v \}(e) \mid x?\ast p = e)}
\]

And likewise for replicated input.

\[
\{ p \rightarrow v \text{ defined } \quad \frac{(x!v \mid x?\ast p = e) \rightarrow (\{ p \rightarrow v \}(e) \mid x?\ast p = e)}{(x!v \mid x?^2 p = e) \rightarrow \{ p \rightarrow v \}(e) \mid x?^2 p = e)}
\]

Reduction proceeds under declaration and parallel composition

\[
\begin{align*}
e_1 \rightarrow e_2 & \quad \frac{e_1 \rightarrow e_3}{(d \ e_1 \rightarrow d \ e_2)} \\
(e_1|e_2) \rightarrow (e_1|e_2)
\end{align*}
\]

if true then \(e_1\) else \(e_2\) \(
\rightarrow \)
if false then \(e_1\) else \(e_2\) \(
\rightarrow \)
Outline

1 Introduction

2 The Language
   - Pict and π
   - Core language
     - The High level
     - An example

3 The type system
   - Types
   - Type safety
Values

\[
\text{Val} \quad = \quad \text{id} \\
\quad \quad \quad \quad [\text{Label} \ \text{Val} \ldots \text{Label} \ \text{Val}] \\
\quad \quad \quad \quad \text{Type} \ \text{Val} \\
\quad \quad \quad \quad (\text{rec} : \ T \ \text{Val} ) \\
\quad \quad \quad \quad \text{String} \\
\quad \quad \quad \quad \text{Char} \\
\quad \quad \quad \quad \text{Int} \\
\quad \quad \quad \quad \text{bool} \\
\text{Label} \quad = \quad \text{empty} \\
\quad \quad \quad \quad \text{id} \\
\text{variable} \\
\text{record} \\
\text{Polymorphic package} \\
\text{Rectype value} \\
\text{String Constant} \\
\text{anonymous label} \\
\text{Explicit label}
\]
Patterns

\[\begin{align*}
\text{Pat} & = \quad \text{id : Type} & \quad \text{variable} \\
_ & : \text{Type} & \quad \text{Wildcard} \\
\text{id : Type} & @ \text{Pat} & \quad \text{Layered} \\
[\text{Label Pat} \ldots \text{Label Pat}] & \quad \text{record} \\
\{ \text{id} < \text{Type} \} & \text{Pat} & \quad \text{Package} \\
(\text{rec : T Pat}) & \quad \text{Rectype}
\end{align*}\]
### Processes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs</td>
<td>Pat = Proc</td>
<td>Process abstraction</td>
</tr>
<tr>
<td>Proc</td>
<td>Val ! Val</td>
<td>output atom</td>
</tr>
<tr>
<td></td>
<td>Val ? Abs</td>
<td>input prefix</td>
</tr>
<tr>
<td></td>
<td>Val ?* Abs</td>
<td>ymorphic package</td>
</tr>
<tr>
<td></td>
<td>(Proc</td>
<td>Proc)</td>
</tr>
<tr>
<td></td>
<td>(Dec — Proc)</td>
<td>Local declaration</td>
</tr>
<tr>
<td></td>
<td>if Val then Proc else Proc</td>
<td>Conditional</td>
</tr>
<tr>
<td>Dec</td>
<td>new id : Type</td>
<td>Channel creation</td>
</tr>
</tbody>
</table>

Janus Dam Nielsen

*Pict: A Programming Language based on the Pi-Calculus*
Outline

1. Introduction

2. The Language
   - Pict and $\pi$
   - Core language
   - The High level
   - An example

3. The type system
   - Types
   - Type safety
Simple transformations

- **Declaration**
  
  \[
  \begin{align*}
  &\text{(new } x_1 \ldots \text{ (new } x_n \text{ e)}) \\
  &\text{(} d_1 \ldots d_n \text{ e) } \Rightarrow \text{(} d_1 \ldots (d_n \text{ e))}
  \end{align*}
  \]

- **Parallel composition**

  \[
  \begin{align*}
  &\text{(run } e_1 \text{ e}_2) \Rightarrow (e_1 | e_2)
  \end{align*}
  \]
### Simple transformations

- **Declaration**
  
  \[(\text{new } x_1 \ldots (\text{new } x_n \text{ e}))\]
  \[(d_1 \ldots d_n \text{ e}) \Rightarrow (d_1 \ldots (d_n \text{ e}))\]

- **Parallel composition**
  
  \[(\text{run } e_1 e_2) \Rightarrow (e_1 | e_2)\]
Abstraction

- Process abstraction:
  \[
  \text{def } f \ [x,y] = (x!y \mid x!y) \\
  (\text{def } x \ p = e_1 \ e_2) \Rightarrow (\text{new } x \ (x?*p = e_1 \mid e_2))
  \]

- Mutually recursive definitions:
  \[
  (\text{def } x_1 a_1 \ldots \text{and } x_n a_n) \Rightarrow \\
  (\text{new } x_1 \ldots (\text{new } x_n \ (x_1?*a_1 \mid \ldots \mid x_n?*a_n \mid e)))
  \]

- Function abstraction
  \[
  \text{def } f \ [a_1 \ a_2 \ a_3 \ r] = r!v \\
  \text{def } f \ [a_1 \ a_2 \ a_3] = v \ (|X_1 < T_1 \ldots X_n < T_n| l_1 p_1 \ldots l_n p_n):T = v \\
  \Rightarrow X_1 < T_1 \ldots X_n < T_n[l_1 p_1 \ldots l_n p_n \ r :! T] = r!v
  \]

- Anonymous functions:
  \[
  \backslash a \Rightarrow (\text{def } x \ a \ x)
  \]
Abstraction

- **Process abstraction:**
  \[
  \text{def } f [x,y] = (x!y \mid x!y)
  \]
  \[
  \text{(def } x \mathord{p} = e_1 e_2) \Rightarrow (\text{new } x (x?*p = e_1 \mid e_2))
  \]

- **Mutually recursive definitions:**
  \[
  (\text{def } x_1 a_1 \ldots \text{ and } x_n a_n) \Rightarrow
  (\text{new } x_1 \ldots (\text{new } x_n (x_1?*a_1 \mid \ldots \mid x_n?*a_n \mid e)))
  \]

- **Function abstraction**
  \[
  \text{def } f [a_1 a_2 a_3 r] = r!v
  \]
  \[
  \text{def } f [a_1 a_2 a_3] = v (|X_1 < T_1 \ldots X_n < T_n|l_1 p_1 \ldots l_n p_n):T = v
  \Rightarrow
  X_1 < T_1 \ldots X_n < T_n[l_1 p_1 \ldots l_n p_n \ r :! T] = r!v
  \]

- **Anonymous functions:**
  \[
  \lambda a \Rightarrow (\text{def } x \ a \ x)
  \]
Abstraction

- Process abstraction:
  
  \[
  \text{def } f [x,y] = (x!y | x!y) \\
  (\text{def } x \ p = e_1 \ e_2) \Rightarrow (\text{new } x (x?*p = e_1 | e_2))
  \]

- Mutually recursive definitions:
  
  \[
  (\text{def } x_1 a_1 \ldots \text{ and } x_n a_n) \Rightarrow \ \\
  (\text{new } x_1 \ldots (\text{new } x_n (x_1?*a_1 | \ldots | x_n?*a_n | e)))
  \]

- Function abstraction
  
  \[
  \text{def } f [a_1 \ a_2 \ a_3 \ r] = r!v \\
  \text{def } f [a_1 \ a_2 \ a_3] = v (|X_1 < T_1 \ldots X_n < T_n|l_1p_1 \ldots l_np_n):T = v \ \\
  \Rightarrow X_1 < T_1 \ldots X_n < T_n[l_1p_1 \ldots l_np_n \ r :! T] = r!v
  \]

- Anonymous functions:
  
  \[
  \backslash a \Rightarrow (\text{def } x \ a \ x)
  \]
Abstraction

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

\[(\text{new } n \ c!n)\]
\[
\begin{align*}
[[x \to c]] &= c!x \\
[[k \to c]] &= c!k \\
[[((d v) \to c)]] &= (d [[v \to c]]) \\
[[((\text{rec} : T v) \to c)]] &= (\text{new } c' ([[v \to c']] | c'?x = c!(\text{rec}:T x)))) \\
[[\{T\} v \to c]] &= (\text{new } c' ([[v \to c']] | c'?x = c!\{T\} x))) \\
[[[l_1 v_1 \ldots l_n v_n]]] &= (\text{new } c_1 ([[v_1 \to c_1]] | c_1 x_1 = \ldots (\text{new } c_n ([[v_n \to c_n]] | c_n x_n = c![l_1 x_1 \ldots l_n x_n]) \ldots )))
\end{align*}
\]
Named values and application

Named value declaration:
\[(\text{val } p = v \ e) \Rightarrow (\text{new } c (\llbracket v \rightarrow c \rrbracket \mid c?p = e))\]

Application:
\[
(\forall v_1 \ldots v_n) \quad \llbracket (v \mid T_1 \ldots T_n \mid l_1 v_1 \ldots l_n v_n) \rightarrow c \rrbracket \\
= (\text{new } c' (\llbracket v \rightarrow c' \rrbracket \mid c' ? x = \ldots) \\
(\text{new } c_1 (\llbracket v_1 \rightarrow c_1 \rrbracket \mid c_1 ? x_1 = \ldots) \\
(\text{new } c_n (\llbracket v_n \rightarrow c_n \rrbracket \mid c_n ? x_n = \\
x! T_1 \ldots T_n[l_1 x_1 \ldots l_n x_n c]) \ldots))
\]
Outline

1. Introduction

2. The Language
   - Pict and $\pi$
   - Core language
   - The High level
   - An example

3. The type system
   - Types
   - Type safety
Hello world.

run ( print!"hello" — print!"world" )
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]
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

1. Introduction

2. The Language
   - Pict and $\pi$
   - Core language
   - The High level
   - An example

3. The type system
   - Types
   - Type safety
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.
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 on channel types

- Refinements of the channel type \( \hat{T} \).
  - \( !T \) for output only
  - \( ?T \) for reading only.
- Natural subtype relation since \( \hat{T} \) 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.
The core language is explicitly typed, but some type information can be derived from the context.
The \( x \) in \( c?x=e \) has type \( \text{int} \) if \( c \) has type \( \wedge \text{int} \).
The inference algorithm is local in that it only uses the immediate surrounding context to determine the type.
Outline

1 Introduction

2 The Language
   - Pict and π
   - Core language
   - The High level
   - An example

3 The type system
   - Types
   - Type safety
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!
• Pict a programming language based on $\pi$.
• A typesystem for Pict.
• Pict can be implemented efficiently.