CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Section 8.1

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2021
Pict: Programming with Processes

Programming with the $\pi$ calculus, can be done in Pict (Pierce and Turner, 2000), which incorporates types, records, and pattern matching into an asynchronous choice-free variant of the $\pi$ calculus.

<table>
<thead>
<tr>
<th>$\pi$ Calculus</th>
<th>Pict</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>()</td>
<td>Empty process</td>
</tr>
<tr>
<td>$\bar{c}.x.0$</td>
<td>$c!x$</td>
<td>Write $x$ on channel $c$ asynchronously</td>
</tr>
<tr>
<td>$c(x).P$</td>
<td>$c?x = p$</td>
<td>Read $x$ on channel $c$</td>
</tr>
<tr>
<td>$P</td>
<td>Q$</td>
<td>$(p</td>
</tr>
<tr>
<td>$(\nu c)P$</td>
<td>(new $c$ $p$)</td>
<td>New channel $c$, scope restricted to $P$</td>
</tr>
<tr>
<td>$!c(x).P$</td>
<td>$c?*x = p$</td>
<td>Replicated input $x$ on channel $c$</td>
</tr>
</tbody>
</table>
Reference Cell in the $\pi$ calculus and Pict

\[ \text{Ref}(g, s, i) \triangleq (\nu l)(\bar{li} \mid \text{GetServer}(l, g) \mid \text{SetServer}(l, s)) \]

\[ \text{GetServer}(l, g) \triangleq !g(c).l(v).(\bar{cv} \mid \bar{lv}) \]

\[ \text{SetServer}(l, s) \triangleq !s(c, v').l(v).(\bar{c} \mid \bar{lv'}) \]

```python
def refInt [res: Int Sig]] =
  (new contents: ^Int
   run contents!0
   def get [res:!Int]
     = contents?v = ( contents!v | res!v )
   def set [v:Int c:Sig]
     = contents?_ = ( contents!v | c![] )
   res![get set]
  )
```

CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Section 8.1 – p. 3/20
\[\pi\] calculus and Pict

The most significant semantic differences between Pict and the \(\pi\) calculus are:

- **Asynchrony**: Pict only allows asynchronous output: this is equivalent to restricting “continuations” of output actions to the empty process \(0\).
- **No choice**: Pict does not support the nondeterministic choice operator \(P + Q\).
- **Restricted replication**: Pict restricts replication of processes to replicated input expressions \(!x(y).P\).
- **No matching**: Pict does not support the match and mismatch operators in the \(\pi\) calculus: \(\text{if } x = y \text{ then } P, \text{ if } x \neq y \text{ then } P\). It supports booleans and more general conditional expressions.
\[ \pi \text{ calculus and Pict} \]

The most significant semantic differences between Pict and the \( \pi \) calculus are:

- **Typed channels:** Pict statically typechecks programs to ensure that channels carry the correct data type at run-time.

- **Tuples and records:** Pict supports the polyadic \( \pi \) calculus generalizing tuples of channels to tuples of tuples, etc. Pict also supports records (tuples with labeled elements) and pattern matching to bind input variables.
A Pict Example

Let us consider the $\pi$ calculus expression:

$$\bar{c}a. P \mid c(x). Q$$

Since Pict only supports asynchronous output, we can only express:

$$\bar{c}a. 0 \mid c(x). Q$$

A similar expression can be written in Pict as follows:

```
run (c!a | c?x=print!"done!")
```
A Pict Example

The program, to be executable using Pict, also needs to define types for the variables \( a \) and \( c \). If we are only going to send a *signal* on channel \( c \), we can use the empty record value and type. The whole program would then look as follows:

```pict
run (new a:[]
    (new c:^[]
        (c!a | c?x=print!"done!")))
```

A shorter version could simply use the constant value `[]` instead of variable \( a \), as follows:

```pict
run (new c:^[]
    (c[] | c?x=print!"done!"))
```
Booleans in the $\pi$ Calculus

We encode a boolean as a channel that receives two channels, a true and a false channel, and if it represents true it replies with a signal over the first channel, otherwise, it sends a signal over the second channel.

\[
\begin{align*}
\text{True}(b) & \triangleq b(t, f).\bar{t} \\
\text{False}(b) & \triangleq b(t, f).\bar{f} \\
\text{If}(b, t, e) & \triangleq (\nu s)(\nu f) \ b(s, f).(s.\bar{t} \mid f.\bar{e})
\end{align*}
\]
Booleans in Pict

In Pict, booleans could be encoded as follows:

\[
\begin{align*}
\text{b?}[t \ f] &= t![\ ] \quad \{- \text{represents true -}\} \\
\text{b?}[t \ f] &= f![\ ] \quad \{- \text{represents false -}\}
\end{align*}
\]

A process that wants to test the value of the boolean can be written as follows:

\[
(b![t \ f] \\
| (t?[] = \text{print!"True"} \\
| (f?[] = \text{print!"False"}))
\]
Booleans in Pict

Once again, we need to assign types to the different channels in the program, namely \( b \), \( t \), and \( f \). So, the full executable program in Pict would read as follows:

```plaintext
run (new b:^[] []
     (b?[t f] = f![] {− represents false −}
     |(new t:^[]
        (new f:^[]
           (b![t f]
            |(t?[] = print!"True"
            |f?[] = print!"False")))))))
```
Pict Programming Language Syntax

\[
\begin{align*}
\text{Label} & \;::=\; [\text{Id} \;=] & \text{Optional label} \\
\text{Path} & \;::=\; \text{Id}\{\text{Id}\}^* & \text{Var/record field path} \\
\text{OType} & \;::=\; [: \text{Type}] & \text{Optional type} \\
\text{Val} & \;::=\; & \text{Values} \\
& \quad \text{Path} & \text{Var/record field path} \\
& \quad [: \text{Label} \text{Val} \ldots \text{Label} \text{Val}] & \text{Record} \\
& \quad \text{Bool} \mid \text{Char} \mid \text{Int} \mid \text{String} & \text{Primitive type constant} \\
\text{Type} & \;::=\; & \text{Types} \\
& \quad \text{^Type} & \text{Input/output channel} \\
& \quad [: \text{Label} \text{Type}\ldots\text{Label} \text{Type}] & \text{Record type} \\
& \quad \text{Bool} \mid \text{Char} \mid \text{Int} \mid \text{String} & \text{Primitive types}
\end{align*}
\]
Pict Programming Language Syntax

\[ Pat ::= \]
\[ \quad \quad \quad Id \ OType \]
\[ \quad | \quad [ \text{Label Pat} \ldots \text{Label Pat} ] \]
\[ \quad | \quad . \ OType \]
\[ \quad | \quad Id \ OType \ @ \ Pat \]

\[ Dec ::= \]
\[ \quad \text{new Id} : \text{Type} \]
\[ \quad | \quad \text{def } Id_1 \ Abs_1 \ \text{and} \ldots \text{and} \ Id_n \ Abs_n \]
\[ \quad | \quad \text{type } Id = \text{Type} \]
# Pict Programming Language Syntax

\[
\text{Proc} \quad ::= \quad \text{Processes} \\
\quad \quad \quad \quad \text{Val} \quad ! \quad \text{Val} \quad \text{Output atom} \\
\quad \quad \quad \quad \text{Val} \quad ? \quad \text{Abs} \quad \text{Input prefix} \\
\quad \quad \quad \quad \text{Val} \quad ?^{*} \quad \text{Abs} \quad \text{Replicated input prefix} \\
\quad \quad \quad \quad () \quad \text{Empty process} \\
\quad \quad \quad \quad (\text{Proc} \quad | \quad \text{Proc} \quad ) \quad \text{Parallel composition} \\
\quad \quad \quad \quad (\text{Dec} \quad \text{Proc} \quad ) \quad \text{Local declaration} \\
\quad \quad \quad \quad \text{if Val then Proc else Proc} \quad \text{Conditional} \\
\text{Abs} \quad ::= \quad \text{Process abstractions} \\
\quad \quad \quad \quad \text{Pat} \quad = \quad \text{Proc} \\
\text{Program} \quad ::= \quad \text{Programs} \\
\quad \quad \quad \quad \text{run} \quad \text{Proc}
User-Declared Types

• We can declare a new Boolean type using the `type` keyword.
• We can chain multiple declarations and multiple processes:
  \((Dec_1 \ Dec_2 \ Proc)\) is translated to \((Dec_1 \ (Dec_2 \ Proc))\) and
  \((Proc_1 \ | \ Proc_2 \ | \ Proc_3)\) is translated to \((Proc_1 \ | \ (Proc_2 \ | \ Proc_3))\).

So the boolean example could have been written as follows:

```plaintext
run (type Boolean = ^[^[] ^[]]
    new b:Boolean
    (b?[t f] = f![]  {- represents false -}
    |(new t:^[]
        new f:^[]
        (b![t f]
        |t[] = print!"True"
        |f[] = print!"False")))))
```

CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Section 8.1 – p. 14/20
Process Definitions

Process definitions in Pict can be used for True and False:

\[
\text{def } tt[b:\text{Boolean}] = b?[t \ f] = t![] \\
\text{and } ff[b:\text{Boolean}] = b?[t \ f] = f![]
\]

- The parameters of a definition must be explicitly annotated with a type.
- Declarations are implemented as a creation of a new channel with replicated input on it. That way, process invocation is the same as writing over the process declaration channel.

For example, the above definition is translated into:

\[
\text{new } tt:\hat{[}\text{Boolean}] \\
\text{new } ff:\hat{[}\text{Boolean}] \\
(tt?*[b:\text{Boolean}] = b?[t \ f] = t![]) \\
|ff?*[b:\text{Boolean}] = b?[t \ f] = f![])
\]
**Wildcard Patterns and Program Declarations**

```plaintext
new tt:^[Boolean]
new ff:^[Boolean]
(tt?*[b:Boolean] = b?[t f] = t![[])
|ff?*[b:Boolean] = b?[t f] = f![[])
```

`tt![b]` can then be used as an abbreviation for `b?[t f] = t![[]`. One more refinement in the `True` and `False` definitions is that `f` is not used in `tt` and `t` is not used in `ff`, so we can use a wildcard pattern:

```plaintext
def tt[b:Boolean] = b?[t _] = t![[]
and ff[b:Boolean] = b?[_ f] = f![[]
```

Finally, declarations can be written without enclosing parentheses, and in such cases, any declared variables are bound in all the remaining clauses of the program. That is, `Dec_1 \ldots Dec_n` run `Proc` is translated into `run (Dec_1 \ldots Dec_n Proc)`.
Boolean Example Program

Thus, the full Pict program after defining a testing procedure follows:

```pict
type Boolean = ^[^[] ^[]]

def tt[b:Boolean] = b?[t _] = t![[]]
and ff[b:Boolean] = b?[_ f] = f![[]]

def test[b:Boolean] =
    (new t:^[] new f:^[]
     (b![t f]
      |t?[] = print!"True"
      |f?[] = print!"False"))

new b:Boolean
run (ff![b] | test![b])
```
Records

Records are an important kind of value, e.g.:

\[
\begin{align*}
\text{new x:}^\mathbb{\mathbb{\{a=\text{Bool } b=\text{Bool } c=[]\}}} \\
\text{run x!}^\mathbb{\mathbb{[a=\text{false } b=\text{true } c=[]]\}} \\
\text{run x?}^\mathbb{\mathbb{[a=p } b=q } c=r\]} &= \\
&\quad \text{if } q \text{ then print!"True"} \\
&\quad \text{else print!"False"}
\end{align*}
\]

defines a channel x that communicates records with three fields with labels a, b, and c of types boolean, boolean, and signal (empty record) respectively.
Record Patterns and Tuples

Patterns can also have wildcards, e.g.:

```
run x?[a=_ b=q c=_] = ...
```

Two other alternatives are:

```
run x?r = if r.b then ...
```

and

```
run x?s@[a=p b=q c=r] = if s.b then...p...r...
```

**Tuples** are unlabeled records, e.g.:

```
new y:^[Bool Bool []]
run y![false true []]
```

```
run y?[p q r] =
    if q then print!"True"
    else print!"False"
```
Pict Structural Congruence and Operational Semantics

Same as (asynchronous) $\pi$ calculus, e.g., scope extrusion:

$$bv(d) \cap fv(e_2) = \emptyset$$

$$((d \ e_1) \ | \ e_2) \equiv (d \ (e_1 \ | \ e_2))$$

(STR-EXTRUDE)

And communication reduction rule:

$$\{v/p\} \quad defined$$

$$(x!v \ | \ x?p = e) \rightarrow e\{v/p\}$$

(RED-COMM)

For example:

$$((\text{new } y:[] \ x!y) \ | \ x?z=z![[]])$$

$$\equiv$$

$$((\text{new } y:[] (x!y \ | \ x?z=z![[]])))$$

$$\rightarrow$$

$$((\text{new } y:[] y![[]])$$