Using the $\pi$-Calculus

Overview
- Evolution
- Values as names
- Boolean values as processes
- Executor, a simple object model, lists
- The polyadic $\pi$-calculus
- Mobile telephones
- Processes as parameters
- A concurrent programming language

References
- Robin Milner, “Communicating and Mobile Systems”
- Robin Milner, Joachim Parrow, David Walker, “A Calculus of Mobile Processes, Part I+II”
Evolution

\[ \overline{x}(y) \mid x(u)\overline{u}(v) \mid \overline{x}(z) \text{ can evolve to } \overline{y}(v) \mid \overline{x}(z) \text{ or } \overline{x}(y) \mid \overline{z}(v) \]

\[ v \ x(\overline{x}(y) \mid x(u)\overline{u}(v)) \mid \overline{x}(z) \text{ can evolve to } \overline{y}(v) \mid \overline{x}(z) \]

\[ \overline{x}(y) \mid \neg x(u)\overline{u}(v) \mid \overline{x}(z) \text{ can evolve to } \]
\[ \overline{x}(y) \mid \neg x(u)\overline{u}(v) \mid \overline{z}(v) \text{ or } \overline{y}(v) \mid \neg x(u)\overline{u}(v) \mid \overline{x}(z) \]

and \[ \overline{y}(v) \mid \neg x(u)\overline{u}(v) \mid \overline{z}(v) \]
Values As Names

If the values with which we wish to compute are drawn from a finite set, say \( V = \{v_1, \ldots, v_n\} \), then we can simply designate \( n \) names to denote these values as constants (e.g. \( v_1 \) to stand for \( v_1 \), \( \ldots \), \( v_n \) to stand for \( v_n \)).

For example, consider the case \( V = \{t, f\} \), the truth values. We set \( t = T \) and \( f = F \).

The match operator can be used to control computation. For example, the following process can be thought of as a C# \texttt{switch}-statement:

\[
x(y).([y = v_1]P_1 + [y = v_2]P_2 + [y = v_3]P_3)
\]
Abbreviations

Sometimes a communication does not need to carry a parameter. To model this we presuppose a special name, say ε, which is never bound. Then we write

\[ \bar{x}.P \text{ in place of } \bar{x}\varepsilon.P \]
\[ x.P \text{ in place of } x(y).P \text{ if } y \notin \text{fn}(P) \]

We often omit ‘.0’ in a process, and write for example

\[ \bar{x}y \text{ in place of } \bar{x}y.0 \]
Boolean Values As Processes

We can represent Boolean values by the processes $\text{True}_a$ and $\text{False}_a$ emitting Boolean values along channel ‘$a$’, and a process $\text{Case}_a(P, Q)$, receiving a Boolean along channel ‘$a$’ and enacting $P$ or $Q$ depending on the value of the Boolean.

$$\begin{align*}
\text{True}_a &= a(x).a(y).\bar{x} \\
\text{False}_a &= a(x).a(y).\bar{y} \\
\text{Case}_a(P, Q) &= (\nu x \nu y)\bar{a}x.\bar{a}y.(x.P | y.Q)
\end{align*}$$

Now $\text{Case}_a(P, Q) | \text{True}_a$ evolves to $(\nu x \nu y)(x.P | y.Q | \bar{x})$, which in turn evolves to $P$ (up-to structural congruence).
Consider the following process definition

\[ \text{Exec}(x) = x(y) . \overline{y} \]

\text{Exec}(x) may be called an executor. It receives via channel ‘x’ a link, called ‘y’, and then activates that link. We can think of ‘y’ as a trigger of a process.

Now for any process P, we obtain the same behavior in each of the following cases:

- We run P directly,
- We prefix a trigger ‘z’ to P, and pass ‘z’ along the channel ‘x’ to the executor \text{Exec}(x) assuming that x, z \notin \text{fn}(P):

\[ \forall z \ (\overline{x}z | z.P) \]
Exec Example

\[ v \times (v \times (z \times \bar{z}, z.P) \mid \text{Exec}(x)) \]

\[ v \times (v \times (\bar{z}, z.P) \mid x(y). \bar{y}) \]

\[ v \times z (\bar{z}, z.P \mid x(y). \bar{y}) \]

\[ \tau.v \times z (0, z.P \mid \bar{z}) \]

\[ \tau.\tau.v \times z (0, P \mid 0) \]

\[ \tau.\tau.P \]
A concurrent system can be thought of as a process community that appears to be an unstructured collection of autonomous agent. In practice, however, we can identify a structure. In particular, we can identify process group boundaries and communications across group boundaries.

ReferenceCell is a process group that represents an object with one private instance variable and two public methods to set and to access the object’s state.

\[
\text{ReferenceCell} \equiv (v, v) ( \overline{v0} \\
| \text{s}(n).s(r).v(_).(\overline{vn} l \overline{r})\\n| \text{g}(r).v(i).(\overline{vi} l \overline{ri}) )
\]
A List

A list is either \textit{Nil} or \textit{Cons} of value and a list.

The constant \textit{Nil}, the construction \textit{Cons}( V, L), and a list of \(n\) values are defined as follows:

\[
\begin{align*}
\text{Nil}_h & = \!h(n).h(c).\overline{n} \\
\text{Cons}(V, L)_h & = (v \, v \, l)(\!h(n).h(c).\overline{cv}.\overline{cl} \mid V\{v\} \mid L\{l\}) \\
[V_1, \ldots, V_n] & = \text{Cons}(V_1, \text{Cons}(\ldots, \text{Cons}(V_n, \text{Nil})\ldots))
\end{align*}
\]
\[ [1, 2] = \text{Cons}(1, \text{Cons}(2, \text{Nil})) \]

\[ \text{Cons}(2, \text{Nil}) = (v \vee l)(!h(n).h(c).\bar{c}v.d | 2\langle v \rangle | \text{Nil}(l)) \]

\[ = (v \vee l)(!h(n).h(c).\bar{c}v.d | 2\langle v \rangle | (!l(n).l(c).\bar{n})) \]

\[ \text{Cons}(1, [2]) = (v' \vee l')(!h(n).h(c).\bar{c}v'.d' | 1\langle v' \rangle | [2]l') \]

\[ = (v' \vee l')(!h(n).h(c).\bar{c}v'.d' | 1\langle v' \rangle | ((v \vee l)!l'(n).l'(c).\bar{c}v.d | 2\langle v \rangle | (!l(n).l(c).\bar{n}))) \]

With \( v' = !v(r).\bar{r}v \)
Head, Tail, and IsEmpty

\[ \text{Nil}_h = !h(n).h(c).\tilde{n} \]
\[ \text{Cons}(V, L)_h = (v v l)(!h(n).h(c).\tilde{v}.\tilde{c}.l| V(v) | L(l)) \]
\[ [V_1, \ldots, V_n] = \text{Cons}(V_1, \text{Cons}(\ldots, \text{Cons}(V_n, \text{Nil})\ldots)) \]

\[ \text{Head}(r) = (v n, c)(\tilde{h}.\tilde{n}.c(v).c(l).\tilde{v} r) \]
\[ \text{Tail}(r) = (v n, c)(\tilde{h}.\tilde{n}.c(v).c(l).\tilde{v} r) \]
\[ \text{IsEmpty} = (v n, c)(\tilde{h}.\tilde{n}.n."Yes" + c."No") \]
List Experiments

IsEmpty(r) | Nil_h = (v n, c)(hnhc.n."Yes" + c."No") | h(n).h(c).n

evolves to (v n, c)(n."Yes" + c."No" | n)

which evolves to "Yes"

Head(r) | [1, 2] =

(v v l)(hnhc.c(v)c(l).vr) | (v v' l')(h(n).h(c).cv'd' | 1(v') | [2'](l'))

Evolves to (v v l v')(vr) | [1, 2](h) | !v'(r).r1

which evolves to r1
The Polyadic $\pi$-Calculus

In the monadic $\pi$-calculus, an interaction involves the transmission of a single name from one process to another. A natural and convenient extension is to admit processes that pass *tuples* of names.

The processes of the polyadic $\pi$-calculus are defined in the same way as the processes of the monadic $\pi$-calculus, except that the prefixes are given by

$$\pi ::= \bar{x}(\bar{y}) \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi$$

where no name occurs more than once in the tuple $\tilde{z}$ in an input prefix.

We write $|\tilde{x}|$ for the length of the tuple $\tilde{x}$. If the length of the corresponding tuple is zero then we write $x$ for $x(\tilde{z})$ and $\bar{x}$ for $\bar{x}(\bar{y})$. 
Polyadic Interaction

The intended interpretations of the new polyadic prefixes are that \( \bar{x}(\bar{y}).P \) can send the tuple \( \langle \bar{y} \rangle \) via ‘x’ and continue as P, and that \( x(\bar{z}).Q \) can receive a tuple \( \langle \bar{y} \rangle \) via ‘x’ and continue as \( \{\bar{y}/\bar{z}\}Q \).

The reaction relation \( \rightarrow \) over \( P^{\pi_0} \) contains exactly those transition that can be inferred from the rules in the following table:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU</td>
<td>( \tau.P + M \rightarrow P )</td>
</tr>
<tr>
<td>P-INTER</td>
<td>( (x(\bar{z}).P + M) \rightarrow {\bar{y}/\bar{z}}P )</td>
</tr>
<tr>
<td>PAR</td>
<td>( P \rightarrow P' \rightarrow P'</td>
</tr>
<tr>
<td>RES</td>
<td>( P \rightarrow P' \rightarrow v a P \rightarrow v a P' )</td>
</tr>
<tr>
<td>STRUCT</td>
<td>( P \equiv Q \rightarrow Q \rightarrow Q' )</td>
</tr>
</tbody>
</table>
Church’s Encoding of Booleans

We can represent a Boolean value as a channel along which we send/receive two other channels for the next true and false interaction.

We define two processes True and False that serve as a process representation of their corresponding Boolean values. Both processes create a new channel ‘b’ that serves as the location of the Boolean value and return ‘b’ along the result channel ‘r’.

\[
\begin{align*}
\text{True}(b) & \equiv b(t, f).t \\
\text{False}(b) & \equiv b(t, f).f \\
\text{Not}(b, c) & \equiv b(t, f).\overline{c}(f, t)
\end{align*}
\]
Mobile Telephones

CAR(talk₁, switch₁)

BASE₁

switch₁

talk₁

alert₁

give₁

alert₂

give₂

PROVIDER₁

IDLEBASE₂
\[ \text{SYSTEM}_1 = (\nu \text{ talk}_i, \text{ switch}_i, \text{ give}_i, \text{ alert}_i : i = 1,2) \\
(CAR(\text{talk}_1, \text{ switch}_1) | \text{BASE}_1 | \text{IDLEBASE}_2 | \text{PROVIDER}_1) \]
A car is parametric upon a *talk* channel and a *switch* channel. On the *talk* channel it can repeatedly talk, but at any time it may receive along the *switch* channel two new channels, which the car must then start to use (new base station):

\[
\text{CAR}(\text{talk, switch}) = \text{talk.CAR}(\text{talk, switch}) + \text{switch}(\text{talk', switch'}).\text{CAR}(\text{talk', switch'})
\]
A BASE can repeatedly talk with the CAR; but at any time it can receive along its give channel two new channels, which it should communicate to the CAR, and then become idle itself:

\[
\text{BASE}(t, s, g, a) = t \cdot \text{BASE}(t, s, g, a) + g(t', s') \cdot s(t', s'). \text{IDLEBASE}(t, s, g, a)
\]

An IDLEBASE may be told along its alert channel to become active:

\[
\text{IDLEBASE}(t, s, g, a) = a \cdot \text{BASE}(t, s, g, a)
\]
The PROVIDER knows initially that the CAR is in contact with BASE\textsubscript{1}. It can decide (according to a provider-specific protocol) to transmit the channel talk\textsubscript{2} and switch\textsubscript{2} to the CAR via BASE\textsubscript{1}, and alert IDLEBASE\textsubscript{2} of this fact:

\[
\text{PROVIDER}_1 = \text{give}_1\langle \text{talk}_2, \text{switch}_2 \rangle \text{alert}_2 \cdot \text{PROVIDER}_2
\]

\[
\text{PROVIDER}_2 = \text{give}_2\langle \text{talk}_1, \text{switch}_1 \rangle \text{alert}_1 \cdot \text{PROVIDER}_1
\]
Evolving System

\[ \text{SYSTEM}_1 \]
\[ \equiv \text{CAR}(\text{talk}_1, \text{switch}_1) | \text{BASE}_1 | \text{IDLEBASE}_2 | \text{PROVIDER}_1 \]
\[ \rightarrow \text{CAR}(\text{talk}_1, \text{switch}_1) | \text{switch}_1(\text{talk}_2, \text{switch}_2).\text{IDLEBASE}_1 | \text{IDLEBASE}_2 | \text{alert}_2.\text{PROVIDER}_2 \]
\[ \rightarrow \text{CAR}(\text{talk}_2, \text{switch}_2) | \text{IDLEBASE}_1 | \text{IDLEBASE}_2 | \text{alert}_2.\text{PROVIDER}_2 \]
\[ \rightarrow \text{CAR}(\text{talk}_2, \text{switch}_2) | \text{IDLEBASE}_1 | \text{BASE}_2 | \text{PROVIDER}_2 \]
\[ \equiv \text{SYSTEM}_2 \]
New System

CAR(talk_2, switch_2)

IDLEBASE_1

alert_1

give_1

PROVIDER_2

BASE_2

talk_2

switch_2

alert_2

give_2
Passing Processes As Parameters

- Passing processes as parameters is not represented directly in the $\pi$-calculus.

- In a direct representation we would write something like

\[ \nu x (\bar{x}P | x(p).p) \]

where $p$ is a variable over processes, and $P$ is a process expression.
Suppose a direct representation of process parameters is given.

We can extend our example as follows:
- The sender, after sending $P$, wishes to run $Q$;
- The receiver (or executor), after receiving $P$, wishes to run it in parallel with $R$.

We would write

$$\nu x (x P.Q | x(p).(p | R))$$

where we assume that $x \notin \text{fn}(P, Q, R)$. A suitable generalized expansion law would equate this to

$$\tau.(Q | (P | R))$$
**Shared Private Name**

We can further extend our example by assuming that prior to the transmission of P there exists a restricted channel ‘w’ between P and Q.

\[ \nu \, x(\nu \, w(\bar{x}P.Q) \mid x(p).(p \mid R)) \]

We have two alternatives for how the transmission of P should treat the private channel ‘w’:

- **Dynamic binding:** \( \tau.(\nu \, w \, Q \mid (P \mid R)) \)
  
  That is, the private link between P and Q is broken and the meaning of ‘w’ is defined with respect to its current scope.

- **Static binding:** \( \tau.\nu \, w'(\{w'/w\}Q \mid (\{w'/w\}P \mid R)) \)
  
  We use a form a scope extrusion where \( w' \) is a fresh name, that is \( w' \not\in \text{fn}(P, Q, R) \).
Modeling Process Parameters

We start with the following process expression

\[ \nu \times (\nu w(\overline{x}P.Q) | x(p).(p | R)) \]

and replace it with

\[ \nu \times (\nu z w(\overline{x}z.(z.P | Q)) | x(y).\overline{y}.R) \]

where \( y, z \notin \text{fn}(R) \), which by expansion, is equal to

\[ \tau.\nu z (\nu w(z.P | Q)) | \overline{z}R \]

Now we change to bound name \( w \) to \( w' \notin \text{fn}(R) \) and obtain

\[ \tau.\nu z w'({w'/w}z.P | {w'/w}Q | \overline{z}R) \]

which, by expansion and then the discard of the restriction becomes

\[ \tau.\tau.(\nu w'({w'/w}P | {w'/w}Q | R) \]
A Concurrent Language

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V ::= X \mid Y \mid \ldots$</td>
<td>Variable</td>
</tr>
<tr>
<td>$F ::= + \mid - \mid \ldots \mid 0 \mid 1 \mid \ldots$</td>
<td>Function symbols</td>
</tr>
<tr>
<td>$C ::= V = E$</td>
<td>Assignment</td>
</tr>
<tr>
<td>$C ; C$</td>
<td>Sequential Composition</td>
</tr>
<tr>
<td>if $E$ then $C$ else $C$</td>
<td>Conditional Statement</td>
</tr>
<tr>
<td>while $E$ do $C$</td>
<td>While Statement</td>
</tr>
<tr>
<td>let $D$ in $C$ end</td>
<td>Declaration</td>
</tr>
<tr>
<td>$C \parallel C$</td>
<td>Parallel Composition</td>
</tr>
<tr>
<td>input $V$</td>
<td>Input</td>
</tr>
<tr>
<td>output $E$</td>
<td>Output</td>
</tr>
<tr>
<td>skip</td>
<td></td>
</tr>
<tr>
<td>$D ::= \text{var} ; V$</td>
<td>Variable Declaration</td>
</tr>
<tr>
<td>$E ::= V$</td>
<td>Variable Expression</td>
</tr>
<tr>
<td>$R ; (E_1, \ldots, E_n)$</td>
<td>Function Call</td>
</tr>
</tbody>
</table>
Ambiguous Meaning

\[ X = 0; \]
\[ X = X + 1 \text{ par } X = X + 2 \]

What is the value of \( X \) at the end of the second statement?
Basic Elements

We assume that each element of the source language is assigned a process expression.

Variables: \( X(\text{init}) \equiv (\nu \nu, \text{setX}, \text{getX}) \)

\[
\begin{align*}
( \nu(\text{init}) \\
| \!\text{setX}(n, r).\nu.(\nu(n) | \check{r}) \\
| \!\text{getX}(r).\nu(i).(\nu(i) | \check{r}(i)) )
\end{align*}
\]

Skip: \[ \text{done} \]

\( C_1 ; C_2: \nu c(\{c/\text{done}\}C_1 | c.C_2) \)

\( C_1 \text{ par } C_2: (\nu l, r, t)(\check{t}(\text{true}) | \{l/\text{done}\}C_1 | \{r/\text{done}\}C_2 | \\
(\nu l, r, t)(\text{if } b \text{ then } r.\text{skip else } \check{l} | \check{t}(\text{false})) | \\
(\nu l, r, t)(\text{if } b \text{ then } l.\text{skip else } \check{r} | \check{t}(\text{false}))) \)
Expressions

\[ [X] = (\nu \text{ ack})(\text{getX}(\text{ack}) | \text{ack}(v).\text{res}(v)) \]

\[ F(E_1, \ldots, E_n) = \text{arg}_1(x_1) \ldots \text{arg}_n(x_n).\bar{F}(x_1, \ldots, x_n, \text{res}) \]

\[ [F(E_1, \ldots, E_n)] = (\nu \text{ arg}_1, \ldots, \text{arg}_n)(\{\text{arg}_1/\text{res}\}[E_1]\{\text{arg}_1/\text{res}\} | \ldots \{\text{arg}_n/\text{res}\}[E_n] | [F]) \]
Operation Sequence

\[ X = 0; \]
\[ X = X + 1 \text{ par } X = X + 2 \]

What is the value of \( X \) at the end of the second statement?

According to the former definitions the value of \( X \) is either 1, 2, or 3. The three values are possible since every atomic action can occur in an arbitrary and meshed order.

To guarantee a specific result (e.g., 1 or 2), we need to employ semaphores.