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

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 $v_1$, $v_2$ to stand for $v_2$, $v_3$ to stand for $v_3$).

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# switch statement:

$$x(y).I[y = v_1]p_1 + I[y = v_2]p_2 + I[y = v_3]p_3$$
Abbreviations

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

  $x.P$  in place of $x_\epsilon.P$

  $x.P$  in place of $x(y).P$ if $y \not\in \text{fn}(P)$

- We often omit '.0' in a process and write for example

  $xy$  in place of $x_\epsilon.y.0$

Boolean Values As Processes

- We can represent Boolean values by the processes True$\alpha$ and False$\alpha$, emitting Boolean values along channel 'a', and a process

  Case$\alpha$(P, Q), receiving a Boolean along channel 'a' and enacting P or Q depending on the value of the Boolean.

  $\nu = True | Q) (P, Case) a (\alpha) x | y.Q | x y)(x. P(a x a y. True) a(y. False)$

  $\nu = (a | x y)(x.P | y.Q)$

  Now Case$\alpha$(P, Q) True$\alpha$ evolves to $(v x y)(x.P | y.Q)$, which in turn evolves to P (up-to structural congruence).

Executor

- Consider the following process definition

  $\text{Exec}(x) = x(y).T$

  $\text{Exec}(x)$ 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 $x, z \not\in \text{fn}(P)$:

    $\nu z (x) z.P$
An Simple Object Model

A concurrent system can be thought of as a process community that appears to be an unstructured collection of autonomous agents. 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.

$$ReferenceCell = (v \; v) \; (\ast n) \; | s(n).s(r).v_{(\cdot)}(\ast n) \cdot \; | s_1(v).v_{(\cdot)}(\ast n)$$

A List

A list is either Null or Cons of value and a list.

$$\text{Nil}, \text{Cons}(V, L), \text{and a list of } n \text{ values are defined as follows:}$$

$$\text{Nil} = \ast n.h(c).\ast n$$
$$\text{Cons}(V, L) = \{v \; v\} | s(n).h(c).v_{(\cdot)}(\ast n) | L | b)$$
$$[V_1, ..., V_n] = \text{Cons}(V_1, \text{Cons}(\text{..., Cons}(V_n, \text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}))$$
\[ [1, 2] = \text{Cons}(1, \text{Cons}(2, \text{Nil})) \]

\[ \text{Cons}(2, \text{Nil}) = (v \in l)(h(n), h(c), c(v)) = 2(v) | \text{Nil}(l) \]

\[ = (v \in l)(h(n), h(c), c(v)) \]

\[ \text{Cons}(1, [2]) = (v \in l')(h(n), h(c), c(v)) = 1(v') | [2](l') \]

\[ = (v \in l')(h(n), h(c), c(v)) \]

\[ \text{with } v(v) = v'(v') \]

---

Head, Tail, and IsEmpty

\[ \text{Nil} = (n, h(n), h(c)) \]

\[ \text{Cons}(V, L) = (v \in l)(h(n), h(c), c(v)) \]

\[ = \text{Cons}(V, \text{Cons}(\ldots, \text{Cons}([1, 2]), \text{Nil}))) \]

\[ \text{Head}(r) = (v \in c)(\text{fin}, c)(v), c(l), \bar{r}) \]

\[ \text{Tail}(r) = (v \in c)(\text{fin}, c)(v), c(l), \bar{r}) \]

\[ \text{IsEmpty} = (v \in c)(\text{fin}, c) \]

---

List Experiments

\[ \text{IsEmpty}(r) \mid \text{Nil} = (v \in c)(\text{fin}, c) \]

\[ = (v \in c)(\text{fin}, c) \]

\[ \mid \text{Cons}(V, \ldots) \mid \text{Nil} \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]

\[ \text{evolves to } (v \in c) \]
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

$$x := \pi(y) \mid x(z) \mid \pi x = \pi y$$

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

We write $|z|$ for the length of the tuple $z$. If the length of the corresponding tuple is zero then we write $x$ for $\pi z$ and $\pi y$ for $\pi x$.

Polyadic Interaction

The intended interpretations of the new polyadic prefixes are that $\pi y \parallel x \parallel P$ can send the tuple $\{y\}$ via $\pi x$ and continue as $P$, and that $x \parallel Q$ can receive a tuple $\{y\}$ via $\pi x$ and continue as $Q \parallel \{y\}$.

The reaction relation $\rightarrow$ over $P^{\pi\pi}$ 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>$\tau P = P$</td>
<td>$\tau$ is a process transition.</td>
</tr>
<tr>
<td>$P \parallel Q \rightarrow P \parallel Q$</td>
<td>$\parallel$ distributes over $\rightarrow$.</td>
</tr>
<tr>
<td>$\pi P = P$</td>
<td>$\pi$ is a process transition.</td>
</tr>
<tr>
<td>$P \parallel P \rightarrow P \parallel P$</td>
<td>$\parallel$ distributes over $\rightarrow$.</td>
</tr>
<tr>
<td>$\pi P \parallel Q \rightarrow \pi P \parallel Q$</td>
<td>$\pi$ distributes over $\parallel$.</td>
</tr>
<tr>
<td>$P \parallel Q \rightarrow P \parallel Q$</td>
<td>$\parallel$ distributes over $\rightarrow$.</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 $y'$.

$$\begin{align*}
\text{True}(b) &= \text{bit}(t, f) \equiv x \parallel y, b(x) \\
\text{False}(b) &= \text{bit}(t, f) \equiv x \parallel y, b(x) \\
\text{Not}(b, c) &= \text{bit}(t, f) \equiv x \parallel y, b(x) \\
\end{align*}$$
Car

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):

\[
\text{CAR(talk, switch)} = \text{talk.CAR(talk, switch)} + \text{switch(talk', switch').CAR(talk', switch')}
\]
Base

- 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:

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

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

\[
\text{IDLEBASE}(t, s, g, a) = \text{a.BASE}(t, s, g, a)
\]

Provider

- The PROVIDER knows initially that the CAR is in contact with BASE1. It can decide (according to a provider-specific protocol) to transmit the channel talk2 and switch2 to the CAR via BASE1, and alert IDLEBASE2 of this fact:

\[
\text{PROVIDER.alert.switch}, \text{talk}, \text{give} \text{PROVIDER}
\]

Evolving System

\[
\begin{align*}
\text{SYSTEM}_1 & = \text{CAR(talk, switch)} | \text{BASE} | \text{IDLEBASE} | \text{PROVIDER} \\
& \quad | \text{CAR(talk, switch)} | \text{switch(talk, switch)}, \text{IDLEBASE} | \\
& \quad | \text{CAR(talk, switch)} | \text{IDLEBASE}, \text{alert}, \text{PROVIDER} \\
& \quad | \text{CAR(talk, switch)} | \text{IDLEBASE}, \text{BASE}, \text{PROVIDER}_2 \\
& = \text{SYSTEM}_0
\end{align*}
\]
Passing Processes As Parameters

- Passing processes as parameters is not represented directly in the π-calculus.
- In a direct representation we would write something like
  \[ \nu x(p).p) | x(p).p) \]
  where \( p \) is a variable over processes, and \( P \) is a process expression.

Scope

- 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
  \[ \tau (Q | P | R) \]
  where we assume that \( x \not\in \text{fn}(P, Q, R) \). 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.

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

- We have two alternatives for how the transmission of P should treat the private channel ‘w’:
  - Dynamic binding: \[ \nu x (w | P . Q | x (p . (P | R) \}) \]
    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: \[ \nu x (w | P . Q | x (p . (P | R) \}) \]
    We use a form of scope extrusion where \( w' \) is a fresh name, that is \( w' \notin \text{fn}(P, Q, R) \).

Modeling Process Parameters

- We start with the following process expression

\[ \nu x (v | w | P . Q | x (p . (P | R) \}) \]

and replace it with

\[ \nu x (v z | w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v z w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v z w' | w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v w' | w | z | P . Q | x (p . (P | R) \}) \]

A Concurrent Language

- We have the following process expression

\[ \nu x (v w | P . Q | x (p . (P | R) \}) \]

and replace it with

\[ \nu x (v z w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v z w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v z w' | w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v w' | w | z | P . Q | x (p . (P | R) \}) \]

A Concurrent Language

- We have the following process expression

\[ \nu x (v w | P . Q | x (p . (P | R) \}) \]

and replace it with

\[ \nu x (v z w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v z w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v z w' | w | z | P . Q | x (p . (P | R) \}) \]

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

\[ \nu x (v w' | w | z | P . Q | x (p . (P | R) \}) \]
Ambiguous Meaning

X = 0; 
X = X + 1 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:

init = (v, setX, getX)

(\(x:\text{init}\):
  \{setX(n, r).v.(n || r)
  \{getX(v).w.((v || r)) || r}}

Skip:

done

\(C_1 \cdot C_2\):
\(v < ((/c/\text{done})C_1 \cdot C_2)

\(C_1 \parallel C_2\):
\(v \parallel (l, r).t\{\text{true}\} || \text{false}\}

Expressions

\([X] = \)
\((v \\text{ack})|getX(\text{ack}) | \text{ack}(v) \cdot \text{x}(v))

\(F(E_1, \ldots, E_n) = \)
\(\text{arg}_1(x_1) \ldots \text{arg}_n(x_n) \cdot F(x_1, \ldots, x_n, \text{res})\)

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

\((\{\text{arg}_1/\text{res}\}|E_1) | F)\)
Operation Sequence

\[
\begin{align*}
X &= 0; \\
X &= X + 1 \text{ par } X = X + 2
\end{align*}
\]

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.