Pi-calculus

syntax and reduction semantics

Francesco Zappa Nardelli
INRIA Rocquencourt, MOSCOVA research team

francesco.zappa_nardelli@inria.fr

MPRI Concurrency course with:

Pierre-Louis Curien (PPS), Roberto Amadio (PPS), Catuscia Palamidessi (INRIA Futurs)

MPRI - Concurrency	October 20, 2006
High-level programming languages

For non-distributed, non-concurrent programming, they are pretty good. We have ML (SML/OCaml), Haskell, Java, C#, with:

- type safety
- rich concrete types — data types and functions
- abstraction mechanisms for program structuring — ML modules and abstract types, type classes and monads, classes and objects, ...

But this is only within single executions of single, sequential programs.

What about distributed computation?
Challenges (idiosyncratic survey)

- **Local concurrency**: $\pi$-calculus, Join, Pict, ...
- **Mobile computations**: JoCaml, Nomadict Pict, ...
- **Marshalling**: choice of distributed abstractions, and trust assumptions: Acute, HashCaml, ...
- **Dynamic (re)binding and evaluation strategies**: exchanging values between programs
- **Type equality between programs**: run-time type names, type-safe and abstraction-safe interaction (and type equality within programs)
- **Typed interaction handles**: establishing shared expression-level names between programs
- **Version change**: type safety in the presence of dynamic linking. Controlling dynamic linking. Dynamic update
- **Semantics for real-world network abstractions**: TCP, UDP, Sockets
- **Security**: security policies, executing untrusted code, protocols, language based
- **Module structure again**: first-class/recursive/parametric modules. Exposing interfaces to other programs via communication
Local concurrency

**Local**: within a single failure domain, within a single trust domain, low-latency interaction.

- Pure (implicit parallelism or skeletons — parallel map, etc.)
- Shared memory
  - mutexes, cvars (*incomprehensible, uncomposable, common*)
  - transactional (Venari, STM Haskell/Java, AtomCaml, ...)
- Message passing
  semantic choices: asynchronous/synchronous, different synchronisation styles (CSP/CCS, Join, ...), input-guarded/general nondeterministic choice, ...

cf Erlang [AVWW96], Telescript, Facile [TLK96,Kna95], Obliq [Car95], CML [Rep99], Pict [PT00], JoCaml [JoC03], Alice [BRS+05], Esterel [Ber98], ...
In these lectures...

- Simplify by considering just interaction.
- What are the equations of interactions?
- Find a logic for interaction.
- Find new/correct paradigms for programming.
- What's about distribution?
- Mobility?
- Security?

Understand some key concepts behind concurrency theory (from a programming language perspective).
In CCS, a system *evolves* when two threads *synchronise* over the same name:

\[ \overline{b}.P \parallel b.Q \rightarrow P \parallel Q \]

We will focus on *reductions* for the time being (that is, forget about LTSs until next lecture). Summary...
CCS, reduction semantics

We define reduction, denoted $\rightarrow$, by

$$a.P \parallel \overline{a}.Q \rightarrow P \parallel Q$$

where, the structural congruence relation, denoted $\equiv$, is defined as:

$$P \parallel Q \equiv Q \parallel P$$

$$P \rightarrow P'$$

$$P \parallel Q \rightarrow P' \parallel Q$$

$$P \equiv P' \rightarrow Q' \equiv Q$$

$$(\nu x)P \rightarrow (\nu x)P'$$

$$P \equiv P' \rightarrow \nu a.P \equiv \nu a.(P \parallel Q)$$ if $a \notin \text{fn}(Q)$

**Theorem** $P \rightarrow Q$ iff $P \xrightarrow{\tau} \equiv Q$. 


Value passing

Names can be interpreted as *channel names*: allow channels to carry values, so instead of pure outputs $\bar{a}.P$ and inputs $a.P$ allow e.g.: $\bar{a}(15,3).P$ and $a(x,y).Q$.

Value 6 being sent along channel $x$:

$$\bar{x}(6) \parallel x(u).\bar{y}(u) \rightarrow (\bar{y}(u))\{^6_u\} = \bar{y}(6)$$

Restricted names are different from all others:

$$\bar{x}(5) \parallel (\nu x)(\bar{x}(6) \parallel x(u).\bar{y}(u)) \rightarrow \bar{x}(5) \parallel (\nu x)(\bar{y}(6)) \equiv$$

$$\bar{x}(5) \parallel (\nu x')(\bar{x}'(6) \parallel x'(u).\bar{y}(u)) \rightarrow \bar{x}(5) \parallel (\nu x'')(\bar{y}(6))$$

(note that we are working with alpha equivalence classes).
Exercise

Program a server that increments the value it receives.

\[ !x(u).\overline{x}^u + 1 \]

Argh!!! This server exhibits exactly the problems we want to avoid when programming concurrent systems:

\[ \overline{x}^3.x(u).P \parallel \overline{x}^7.x(v).Q \parallel !x(u).\overline{x}^u + 1 \rightarrow \ldots \]

\[ \ldots \rightarrow P^{8/\!u} \parallel Q^{4/\!u} \parallel !x(u).\overline{x}^u + 1 \]
Ideas...

Allow those values to include channel names.

A new implementation for the server:

\[ !x(u, r).\bar{r}(u + 1) \]

This server prevents confusion provided that the return channels are distinct.

How can we guarantee that the return channels are distinct?

The restriction operator we have is overly restrictive...
1. A name received on a channel can then be used itself as a channel name for output or input — here $y$ is received on $x$ and the used to output $7$:

$$
\bar{x}(y) \parallel x(u).\overline{u}(7) \rightarrow \overline{y}(7)
$$

2. A restricted name can be sent outside its original scope. Here $y$ is sent on channel $x$ outside the scope of the $(\nu y)$ binder, which must therefore be moved (with care, to avoid capture of free instances of $y$). This is scope extrusion:

$$(\nu y)(\bar{x}(y) \parallel y(v).P) \parallel x(u).\overline{u}(7) \rightarrow (\nu y)(y(v).P \parallel \overline{y}(7))$$

$$\rightarrow (\nu y)(P\{7/v\})$$
The (simplest) $\pi$-calculus

Syntax:

\[
\begin{align*}
P, Q & ::= 0 \quad \text{nil} \\
   & | \quad \parallel Q \quad \text{parallel composition of } P \text{ and } Q \\
   & | \quad \vec{c}\langle v \rangle.P \quad \text{output } v \text{ on channel } c \text{ and resume as } P \\
   & | \quad c(x).P \quad \text{input from channel } c \\
   & | \quad (\nu x)P \quad \text{new channel name creation} \\
   & | \quad !P \quad \text{replication}
\end{align*}
\]

Free names (alpha-conversion follows accordingly):

\[
\begin{align*}
\text{fn}(0) & = \emptyset \\
\text{fn}(\vec{c}\langle v \rangle.P) & = \{c, v\} \cup \text{fn}(P) \\
\text{fn}((\nu x)P) & = \text{fn}(P) \setminus \{x\} \\
\text{fn}(P \parallel Q) & = \text{fn}(P) \cup \text{fn}(Q) \\
\text{fn}(c(x).P) & = (\text{fn}(P) \setminus \{x\}) \cup \{c\} \\
\text{fn}(!P) & = \text{fn}(P)
\end{align*}
\]
\[ \pi\text{-calculus, reduction semantics} \]

Structural congruence:

\[
\begin{align*}
    P \parallel 0 & \equiv P \\
    P \parallel Q & \equiv Q \parallel P \\
    (P \parallel Q) \parallel R & \equiv P \parallel (Q \parallel R) \\
    !P & \equiv P \parallel !P \\
    (\nu x)(\nu y)P & \equiv (\nu y)(\nu x)P \\
    P \parallel (\nu x)Q & \equiv (\nu x)(P \parallel Q) \text{ if } x \notin \text{fn}(P)
\end{align*}
\]

Reduction rules:

\[
\begin{align*}
    \bar{c}\langle v \rangle . P \parallel c(x). Q & \rightarrow P \parallel Q\{v/\overline{x}\} \\
    P & \rightarrow P' \\
    P \parallel Q & \rightarrow P' \parallel Q \\
    (\nu x)P & \rightarrow (\nu x)P' \\
    P \equiv P' \rightarrow Q' & \equiv Q \\
    P & \rightarrow Q
\end{align*}
\]
Expressiveness

A small calculus (and the semantics only involves name-for-name substitution, not term-for-variable substitution), but very expressive:

- encoding data structures
- encoding functions as processes (Milner, Sangiorgi)
- encoding higher-order $\pi$ (Sangiorgi)
- encoding synchronous communication with asynchronous (Honda/Tokoro, Boudol)
- encoding polyadic communication with monadic (Quaglia, Walker)
- encoding choice (or not) (Nestmann, Palamidessi)
- ...

Example: polyadic with monadic

Let us extend our notion of monadic channels, which carry exactly one name, to polyadic channels, which carry a vector of names, i.e.

\[
P ::= \overline{x}\langle y_1, \ldots, y_n \rangle.P \quad \text{output}
\]

\[
x(y_1, \ldots, y_n).P \quad \text{input}
\]

with the main reduction rule being:

\[
\overline{x}\langle y_1, \ldots, y_n \rangle P \parallel x(z_1, \ldots, z_n).Q \rightarrow P \parallel Q\{y_1, \ldots, y_n/z_1, \ldots, z_n\}
\]

Is there an encoding from polyadic to monadic channels?
Polyadic with monadic, ctd.

We might try:

\[
\begin{align*}
[[x \langle y_1, \ldots, y_n \rangle . P]] &= \bar{x} \langle y_1 \rangle \ldots \bar{x} \langle y_n \rangle . [[P]] \\
[[x(y_1, \ldots, y_n) . P]] &= x(y_1) \ldots x(y_n) . [[P]]
\end{align*}
\]

but this is broken! Why?

The right approach is use new binding:

\[
\begin{align*}
[[x \langle y_1, \ldots, y_n \rangle . P]] &= (\nu z)(\bar{x} \langle z \rangle . \bar{z} \langle y_1 \rangle \ldots \bar{z} \langle y_n \rangle . [[P]]) \\
[[x(y_1, \ldots, y_n) . P]] &= x(z).z(y_1) \ldots z(y_n) . [[P]]
\end{align*}
\]

where \( z \notin \text{fn}(P) \) (why?). (We also need some well-sorted assumptions.)
Recursion

Alternative to replication: recursive definition of processes.

Recursive definition:

\[ K = (\tilde{x}).P \]

Constant application:

\[ K[\tilde{a}] \]

Reduction rule:

\[
\begin{align*}
K &= (\tilde{x}).P \\
K[\tilde{a}] &\rightarrow P\{\tilde{a}/\tilde{x}\}
\end{align*}
\]
Recursion vs. Replication

**Theorem**  Any process involving recursive definitions is representable using replication, and conversely replication is redundant in presence of recursion.

The proof requires some techniques we have not seen, but...

Intuition: given

\[ F = (\tilde{x}).P \]

where \( P \) may contain recursive calls to \( F \) of the form \( F[\tilde{z}] \), we may replace the RHS with the following process abstraction containing no mention of \( F \):

\[
(\tilde{x}).(\nu f)(\overline{f}\langle\tilde{x}\rangle \parallel !f(\tilde{x}).P')
\]

where \( P' \) is obtained by replacing every occurrence of \( F[\tilde{z}] \) by \( \overline{f}\langle\tilde{z}\rangle \) in \( P \), and \( f \) is fresh for \( P \).
Data as processes: booleans

Consider the truth-values \{\text{True}, \text{False}\}. Consider the abstractions:

\[
T = (x).x(t, f).t \langle \rangle \quad \text{and} \quad F = (x).x(t, f).f \langle \rangle
\]

These represent a located copy of a truth-value at \(x\). The process

\[
R = (\nu t)(\nu f)\overline{b}(t, f). (t().P \parallel f().Q)
\]

where \(t, f \notin \text{fn}(P, Q)\) can test for a truth-value at \(x\) and behave accordingly as \(P\) or \(Q\):

\[
R \parallel T[b] \rightarrow P \parallel (\nu t, f)f().Q
\]

The term obtained behaves as \(P\) because the thread \((\nu t, f)f().Q\) is deadlocked.
Data as processes: integers

Using a unary representation.

\[
[[k]] = (x).x(z, o).(\bar{o}\langle\rangle)^k.\bar{z}\langle\rangle
\]

where \((\bar{o}\langle\rangle)^k\) abbreviates \(\bar{o}\langle\rangle.\bar{o}\langle\rangle.\ldots.\bar{o}\langle\rangle\) \((k\) occurrences).

Operations on integers can be expressed as processes. For instance,

\[
succ = (x, y).!x(z, o).\bar{o}\langle\rangle.\bar{y}\langle z, o\rangle
\]

Which is the role of the final output on \(z\)? (Hint: omit it, and try to define the test for zero).
Another representation for integers

```
type Nat = zero | succ Nat
Define:
[[zero]] = (x).!x(z, s).z⟩
[[succ]] = (x, y).!x(z, s).s⟨y⟩
and for each e of type Nat:
[[succ e]] = (x).(νy)([[succ]]|x, y⟩ || [[e]]|y⟩)
```

This approach generalises to arbitrary datatypes.
A step backward: defining a language

Recipe:

1. define the *syntax* of the language (that is, specify what a program is);

2. define its *reduction semantics* (that is, specify how programs are executed);

3. define when *two terms are equivalent* (that is, hum...?!).

Share and enjoy the new language...
Suppose that $P$ and $Q$ are equivalent (in symbols: $P \simeq Q$).

Which properties do we expect?

**Preservation under contexts** For all contexts $C[-]$, we have $C[P] \simeq C[Q]$;

**Same observations** If $P \downarrow x$ then $Q \downarrow x$, where $P \downarrow x$ means that we can observe $x$ at $P$ (or $P$ can do $x$);

**Preservation of reductions** $P$ and $Q$ must mimic their reduction steps (that is, they realise the same nondeterministic choices).
Reduction-closed barbed congruence

Let reduction barbed congruence, denoted $\simeq$, be the largest symmetric relation over processes that is preserved by contexts, barb preserving, and reduction closed.

Remark: reduction barbed congruence is a weak equivalence: the number of internal reduction steps is not important in the bisimulation game imposed by “reduction closed”.
Some equivalences (?)

Compare the processes

1. $P = \overline{x}\langle y \rangle$ and $Q = 0$

2. $P = \overline{a}\langle x \rangle$ and $Q = \overline{a}\langle z \rangle$

3. $P = (\nu x)\overline{x}\langle \rangle.R$ and $Q = 0$

4. $P = (\nu x)(\overline{x}\langle y \rangle.R_1 \parallel x(z).R_2)$ and $Q = (\nu x)(R_1 \parallel R_2\{y/z\})$

Argh... we need other proof techniques to show that processes are equivalent!

Remark: we can reformulate barb preservation as “if $P \mathcal{R} Q$ and $P \downarrow x$ imply $Q \downarrow x$”. This is sometimes useful...
Example: local names are different from global names

Show that in general

\[(\nu x)!P \not\equiv !(\nu x)P\]

Intuition: the copies of \(P\) in \((\nu x)!P\) can interact over \(x\), while the copies of \((\nu x)P\) cannot.

We need a process that interacts with another copy of itself over \(x\), but that cannot interact with itself over \(x\). Take

\[P = \overline{x}\langle \rangle \oplus x().\overline{b}\langle \rangle\]

where \(Q_1 \oplus Q_2 = (\nu w)(\overline{w}\langle \rangle \parallel w().Q_1 \parallel w().Q_2\).

We have that \((\nu x)!P \downarrow b\), while \(!((\nu x)P \uparrow b\).
Exercises

1. Compare the transitions of $F[u, v]$, where $F = (x, y).x(y).F[y, x]$ to those of its encoding in the recursion free calculus (use replication).

2. Consider the pair of mutually recursive definitions

   $$
   G = (u, v). (u().H[u, v] || k().H[u, v])
   $$

   $$
   H = (u, v).v().G[u, v]
   $$

   Write the process $G[x, y]$ in terms of replication (you have to invent the technique to translate mutually recursive definitions yourself).

3. Implement a process that negates at location $a$ the truth-value found at location $b$. Implement a process that sums of two integers (using both the representations we have seen).

References

Books

- Robin Milner, Communicating and mobile systems: the $\pi$-calculus. (CUP, 1999).

Tutorials available online:

- Joachim Parrow, An introduction to the pi-calculus. http://user.it.uu.se/~joachim/intro.ps