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: π-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 \textit{evolves} when two threads \textit{synchronise} over the same name:

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

We will focus on \textit{reductions} for the time being (that is, forget about LTSs until next lecture). Summary...
We define reduction, denoted $\rightarrow$, by

$$a.P \parallel a.Q \rightarrow P \parallel Q$$

$$P \rightarrow P'$$
$$P \parallel Q \rightarrow P' \parallel Q$$
$$P \rightarrow P' \quad (\nu x)P \rightarrow (\nu x)P'$$
$$P \equiv P' \rightarrow Q' \equiv Q$$

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

$$P \parallel Q \equiv Q \parallel P$$
$$\quad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$$
$$P \parallel 0 \equiv P$$
$$!P \equiv P \parallel !P$$
$$\quad (\nu a)P \parallel Q \equiv (\nu a)(P \parallel Q) \text{ if } a \not\in \text{fn}(Q)$$

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 $a.P$ and inputs $a.P$ allow e.g.: $a\langle15,3\rangle.P$ and $a(x,y).Q$.

Value 6 being sent along channel $x$:

$$x\langle6\rangle \parallel x(u).y\langle u\rangle \rightarrow (y\langle u\rangle)\{^6\langle u\rangle\} = y\langle 6\rangle$$

Restricted names are different from all others:

$$x\langle 5\rangle \parallel (\nu x)(x\langle 6\rangle \parallel x(u).y\langle u\rangle) \rightarrow x\langle 5\rangle \parallel (\nu x)(y\langle 6\rangle)$$

$$\equiv$$

$$x\langle 5\rangle \parallel (\nu x')(x'\langle 6\rangle \parallel x'(u).y\langle u\rangle) \rightarrow x\langle 5\rangle \parallel (\nu x'')(y\langle 6\rangle)$$

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

Program a server that increments the value it receives.

\[ !x(u).x(u + 1) \]

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

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

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

Allow those values to include channel names.

A new implementation for the server:

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

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...
The π-calculus

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 $7$:

$$x(y) \parallel x(u).u\langle 7 \rangle \rightarrow y\langle 7 \rangle$$

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:

$$(\nu y)(x(y) \parallel y(v).P) \parallel x(u).u\langle 7 \rangle \rightarrow (\nu y)(y(v).P \parallel y\langle 7 \rangle)$$

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

Syntax:

\[ P, Q ::= 0 \quad \| \quad \text{nil} \]

\[ P \parallel Q \quad \text{parallel composition of } P \text{ and } Q \]

\[ \langle v \rangle . P \quad \text{output } v \text{ on channel } c \text{ and resume as } P \]

\[ c(x).P \quad \text{input from channel } c \]

\[ (\nu x) P \quad \text{new channel name creation} \]

\[ !P \quad \text{replication} \]

Free names (alpha-conversion follows accordingly):

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

Structural congruence:

\[
\begin{align*}
P \parallel 0 & \equiv P \\
(P \parallel Q) \parallel R & \equiv P \parallel (Q \parallel R) \\
!P & \equiv P \parallel !P \\
(\nu x)(\nu y)P & \equiv (\nu y)(\nu x)P \\
(P \parallel (\nu x)Q) & \equiv (\nu x)(P \parallel Q) \text{ if } x \not\in \text{fn}(P)
\end{align*}
\]

Reduction rules:

\[
\begin{align*}
\text{c}(v).P \parallel c(x).Q & \rightarrow P \parallel Q\{v/x\}
\end{align*}
\]
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.

\[ P ::= x\langle y_1, ..., y_n \rangle.P \quad \text{output} \]
\[ \quad \mid x(y_1, ..., y_n).P \quad \text{input} \]

with the main reduction rule being:

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

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

We might try:

\[
\[[x\langle y_1, \ldots, y_n \rangle.P]] = x\langle y_1 \rangle \ldots x\langle y_n \rangle.[[P]]
\]
\[
[[x(y_1, \ldots, y_n).P]] = x(y_1) \ldots x(y_n).[[P]]
\]

but this is broken! Why?

The right approach is use new binding:

\[
\[[x\langle y_1, \ldots, y_n \rangle.P]] = (\nu z)(x\langle z \rangle.z\langle y_1 \rangle \ldots z\langle y_n \rangle.[[P]])
\]
\[
[[x(y_1, \ldots, y_n).P]] = x(z).z(y_1) \ldots z(y_n).[[P]]
\]

where \( z \not\in \text{fn}(P) \) (why?). (We also need some well-sorted assumptions.)
Recursion

Alternative to replication: recursive definition of processes.

Recursive definition:

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

Constant application:

\[ K[\bar{a}] \]

Reduction rule:

\[ \frac{K = (\bar{x}).P}{K[\bar{a}] \rightarrow P \{ \bar{a}/\bar{x} \}} \]
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

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

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

\[ (\bar{x}).(\nu f)(f(\bar{x}) \parallel !f(\bar{x}).P') \]

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

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

\[ T = (x).x(t, f).\top \] and \[ F = (x).x(t, f).\bot \]

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

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

where \( t, f \not\in fn(P, Q) \) 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.

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

where \((\langle\rangle)^k\) abbreviates \(\langle\rangle.\langle\rangle.\ldots.\langle\rangle\) (k occurrences).

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

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

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

\[
\text{type Nat = zero \mid succ Nat}
\]

Define:

\[
[[\text{zero}]] = (x).!x(z,s).z\langle \rangle
\]
\[
[[\text{succ}]] = (x,y).!x(z,s).s\langle y\rangle
\]

and for each \( e \) of type \( \text{Nat} \):

\[
[[\text{succ } e]] = (x).(\nu y)([[\text{succ}}] x,y] \parallel [[e]y])
\]

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).
Formally

A relation $\mathcal{R}$ between processes is

**preserved by contexts:** if $P \mathcal{R} Q$ implies $C[P] \mathcal{R} C[Q]$ for all contexts $C[-]$.

**barb preserving:** if $P \mathcal{R} Q$ and $P \downarrow x$ imply $Q \downarrow x$, where $P \downarrow x$ holds if there exists $P'$ such that $P \rightarrow^* P'$ and $P' \downarrow x$, while

$$P \equiv (\nu \tilde{n})(x(y).P' \parallel P'')$$

or

$$P \equiv (\nu \tilde{n})(x(u).P' \parallel P'')$$

for $x \notin \tilde{n}$;

**reduction closed:** if $P \mathcal{R} Q$ and $P \rightarrow P'$, imply that there is a $Q'$ such that $Q \rightarrow^* Q'$ and $P' \mathcal{R} Q'$ ($\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$).
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. $P = x\langle y \rangle$ and $Q = 0$

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

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

4. $P = (\nu x)(x\langle y \rangle.R_1 \parallel x(z).R_2)$ 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 $P \triangleright\triangleright Q$ and $P \Downarrow x$ imply $Q \Downarrow x$”. This is sometimes useful...
Example: local names are different from global names

Show that in general

\[(\nu x)!P \nless\neq !(\nu x)P\]

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

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

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

We have that \((\nu x)!P \downarrow b\), while \(!(\nu x)P \not\downarrow b\).
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

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

   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).

4. Design a representation for lists using $\pi$-calculus processes. Implement list append.
References

Books


Tutorials available online:

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