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)

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).

CCS, synchronisation

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 $\rightarrowtriangle$, by

$$a.P \parallel \overline{a}.Q \;\rightarrowtriangle\; P \parallel Q$$

$$\frac{P \;\rightarrowtriangle\; P'}{P \parallel Q \;\rightarrowtriangle\; P' \parallel Q} \qquad \frac{P \;\rightarrowtriangle\; P'}{(\boldsymbol{\nu} x)P \;\rightarrowtriangle\; (\boldsymbol{\nu} x)P'} \qquad \frac{P \equiv P' \rightarrowtriangle Q' \equiv Q}{P \rightarrowtriangle Q}$$

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

$$P \parallel Q \equiv Q \parallel P \qquad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R)$$

$$P \parallel \mathbf{0} \equiv P \qquad !P \equiv P \parallel !P \qquad (\boldsymbol{\nu} a)P \parallel Q \equiv (\boldsymbol{\nu} a)(P \parallel Q) \text{ if } a \notin \mathrm{fn}(Q)$$

Theorem $P \;\rightarrowtriangle\; Q$ 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 $\overline{a}.P$ and inputs $a.P$ allow e.g.: $\overline{a}\langle 15, 3\rangle.P$ and $a(x, y).Q$.

Value 6 being sent along channel x:

$$\overline{x}\langle 6\rangle \;\Big|\Big|\; x(u).\overline{y}\langle u\rangle \;\rightarrowtriangle\; (\overline{y}\langle u\rangle)\{{}^{6}/_{u}\} \;=\; \overline{y}\langle 6\rangle$$

Restricted names are different from all others:

$$\begin{array}{ccc}
\overline{x}\langle 5\rangle \;||\; (\boldsymbol{\nu} x)(\overline{x}\langle 6\rangle \;||\; x(u).\overline{y}\langle u\rangle) & \rightarrowtriangle & \overline{x}\langle 5\rangle \;||\; (\boldsymbol{\nu} x)(\overline{y}\langle 6\rangle) \\
\equiv & & \equiv \\
\overline{x}\langle 5\rangle \;||\; (\boldsymbol{\nu} x')(\overline{x'}\langle 6\rangle \;||\; x'(u).\overline{y}\langle u\rangle) & \rightarrowtriangle & \overline{x}\langle 5\rangle \;||\; (\boldsymbol{\nu} x'')(\overline{y}\langle 6\rangle)
\end{array}$$

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

Exercise

Program a server that increments the value it receives.

$$!x(u).\overline{x}\langle u+1\rangle$$

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

$$\overline{x}\langle 3\rangle.x(u).P \;\|\; \overline{x}\langle 7\rangle.x(v).Q \;\|\; !x(u).\overline{x}\langle u+1\rangle \;\twoheadrightarrow \ldots$$

$$\ldots \twoheadrightarrow P\{^8/_u\} \;\|\; Q\{^4/_u\} \;\|\; !x(u).\overline{x}\langle u+1\rangle$$

Ideas...

Allow those values to include channel names.

A new implementation for the server:

$$!x(u, r).\overline{r}\langle u + 1\rangle$$

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:

$$\overline{x}\langle y\rangle \;\Big|\Big|\; x(u).\overline{u}\langle 7\rangle \;\rightharpoonup\; \overline{y}\langle 7\rangle$$

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

$$\begin{aligned}(\boldsymbol{\nu} y)(\overline{x}\langle y\rangle \;||\; y(v).P) \;||\; x(u).\overline{u}\langle 7\rangle \quad &\rightharpoonup \quad (\boldsymbol{\nu} y)(y(v).P \;||\; \overline{y}\langle 7\rangle)\\ &\rightharpoonup \quad (\boldsymbol{\nu} y)(P\{{}^{7}\!/\!_{v}\})\end{aligned}$$

The (simplest) π-calculus

Syntax:

$$
\begin{array}{rcll}
P, Q & ::= & \mathbf{0} & \text{nil} \\
 & | & P \mid\mid Q & \text{parallel composition of } P \text{ and } Q \\
 & | & \overline{c}\langle v\rangle.P & \text{output } v \text{ on channel } c \text{ and resume as } P \\
 & | & c(x).P & \text{input from channel } c \\
 & | & (\boldsymbol{\nu} x)P & \text{new channel name creation} \\
 & | & !P & \text{replication}
\end{array}
$$

Free names (alpha-conversion follows accordingly):

$$
\begin{array}{rclrcl}
\mathrm{fn}(\mathbf{0}) & = & \emptyset & \mathrm{fn}(P \mid\mid Q) & = & \mathrm{fn}(P) \cup \mathrm{fn}(Q) \\
\mathrm{fn}(\overline{c}\langle v\rangle.P) & = & \{c, v\} \cup \mathrm{fn}(P) & \mathrm{fn}(c(x).P) & = & (\mathrm{fn}(P) \setminus \{x\}) \cup \{c\} \\
\mathrm{fn}((\boldsymbol{\nu} x)P) & = & \mathrm{fn}(P) \setminus \{x\} & \mathrm{fn}(!P) & = & \mathrm{fn}(P)
\end{array}
$$

π-calculus, reduction semantics

Structural congruence:

$$
\begin{array}{rcl}
P \mid\mid 0 & \equiv & P \\
(P \mid\mid Q) \mid\mid R & \equiv & P \mid\mid (Q \mid\mid R)
\end{array}
\qquad
\begin{array}{rcl}
P \mid\mid Q & \equiv & Q \mid\mid P \\
!P & \equiv & P \mid\mid !P
\end{array}
$$

$$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$$

$$P \mid\mid (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \mid\mid Q) \text{ if } x \notin \mathrm{fn}(P)$$

Reduction rules:

$$\overline{c}\langle v\rangle.P \mid\mid c(x).Q \rightarrow P \mid\mid Q\{^v/_x\}$$

$$\frac{P \rightarrow P'}{P \mid\mid Q \rightarrow P' \mid\mid Q} \qquad \frac{P \rightarrow P'}{(\boldsymbol{\nu} x)P \rightarrow (\boldsymbol{\nu} x)P'} \qquad \frac{P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q}$$

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 π (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.

$$
\begin{array}{lcll}
P & ::= & \overline{x}\langle y_1, ..., y_n \rangle.P & \text{output} \\
 & \mid & x(y_1, ..., y_n).P & \text{input}
\end{array}
$$

with the main reduction rule being:

$$\overline{x}\langle y_1, ..., y_n \rangle P \;\Big|\Big|\; x(z_1, ..., z_n).Q \;\rightarrow\; P \;\Big|\Big|\; Q\{^{y_1, \cdots y_n}/_{z_1, \ldots, z_n}\}$$

Is there an encoding from polyadic to monadic channels?

Polyadic with monadic, ctd.

We might try:

$$\begin{array}{rcl} [[\overline{x}\langle y_1, ..., y_n\rangle.P]] & = & \overline{x}\langle y_1\rangle.\ldots.\overline{x}\langle y_n\rangle.[[P]] \\ [[x(y_1, ..., y_n).P]] & = & x(y_1).\ldots.x(y_n).[[P]] \end{array}$$

but this is broken! Why?

The right approach is use new binding:

$$\begin{array}{rcl} [[\overline{x}\langle y_1, ..., y_n\rangle.P]] & = & (\boldsymbol{\nu} z)(\overline{x}\langle z\rangle.\overline{z}\langle y_1\rangle.\ldots.\overline{z}\langle y_n\rangle.[[P]]) \\ [[x(y_1, ..., y_n).P]] & = & x(z).z(y_1).\ldots.z(y_n).[[P]] \end{array}$$

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

Recursion

Alternative to replication: recursive definition of processes.

Recursive definition:

$$K \;=\; (\tilde{x}).P$$

Constant application:

$$K\lfloor\tilde{a}\rfloor$$

Reduction rule:

$$\frac{K = (\tilde{x}).P}{K\lfloor\tilde{a}\rfloor \;\rightarrow\; P\{^{\tilde{a}}/_{\tilde{x}}\}}$$

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

$$F = (\tilde{x}).P$$

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

$$(\tilde{x}).(\boldsymbol{\nu} f)(\overline{f}\langle\tilde{x}\rangle \; \big|\big| \; !f(\tilde{x}).P')$$

where P' is obtained by replacing every occurrence of $F\lfloor\tilde{z}\rfloor$ by $\overline{f}\langle\tilde{z}\rangle$ in P, and f is fresh for P.

Data as processes: booleans

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

$$T \;=\; (x).x(t,f).\bar{t}\langle\rangle \quad \text{and} \quad F \;=\; (x).x(t,f).\overline{f}\langle\rangle$$

These represent a *located copy* of a truth-value at x. The process

$$R \;=\; (\boldsymbol{\nu} t)(\boldsymbol{\nu} f)\bar{b}\langle t,f\rangle.(t().P \,\Big|\Big|\, f().Q)$$

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

$$R \,\Big|\Big|\, T\lfloor b\rfloor \rightarrow\!\!\!\!\triangleright\!\!\rightarrow\!\!\!\!\triangleright P \,\Big|\Big|\, (\boldsymbol{\nu} t,f)f().Q$$

The term obtained behaves as P because the thread $(\boldsymbol{\nu} t,f)f().Q$ is deadlocked.

Data as processes: integers

Using a unary representation.

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

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

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

$$\mathrm{succ} \;=\; (x,y).!x(z,o).\overline{o}\langle\rangle.\overline{y}\langle z,o\rangle$$

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:

$$
\begin{aligned}
[\![\texttt{zero}]\!] &= (x).!x(z,s).\overline{z}\langle\rangle \\
[\![\texttt{succ}]\!] &= (x,y).!x(z,s).\overline{s}\langle y\rangle
\end{aligned}
$$

and for each e of type `Nat`:

$$[\![\texttt{succ}\ e]\!] = (x).(\boldsymbol{\nu} y)([\![\texttt{succ}]\!]\lfloor x,y\rfloor \;\Big|\Big|\; [\![e]\!]\lfloor y\rfloor)$$

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...

Equivalent?

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 $P \mathrel{\mathcal{R}} Q$ implies $C[P] \mathrel{\mathcal{R}} C[Q]$ for all contexts $C[-]$.

barb preserving: if $P \mathrel{\mathcal{R}} Q$ 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

$$P \equiv (\boldsymbol{\nu}\tilde{n})(\overline{x}\langle y\rangle.P' \mid\mid P'') \text{ or } P \equiv (\boldsymbol{\nu}\tilde{n})(x(u).P' \mid\mid P'') \text{ for } x \notin \tilde{n} \ ;$$

reduction closed: if $P \mathrel{\mathcal{R}} Q$ and $P \rightarrow P'$, imply that there is a Q' such that $Q \rightarrow^* Q'$ and $P' \mathrel{\mathcal{R}} Q'$ ($\rightarrow^*$ 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. $P = \overline{x}\langle y\rangle$ and $Q = \mathbf{0}$

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

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

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

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

$$(\boldsymbol{\nu} x)!P \;\not\approx\; !(\boldsymbol{\nu} x)P$$

Intuition: the copies of P in $(\boldsymbol{\nu} x)!P$ can interact over x, while the copies of $(\boldsymbol{\nu} x)P$ 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 $Q_1 \oplus Q_2 = (\boldsymbol{\nu} w)(\overline{w}\langle\rangle \;||\; w().Q_1 \;||\; w().Q_2$.

We have that $(\boldsymbol{\nu} x)!P \Downarrow b$, while $!(\boldsymbol{\nu} x)P \not\Downarrow b$.

Exercises

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

2. Consider the pair of mutually recursive definitions

$$\begin{array}{rcl} G & = & (u, v).(u().H\lfloor u, v \rfloor \parallel k().H\lfloor u, v \rfloor) \\ H & = & (u, v).v().G\lfloor u, v \rfloor \end{array}$$

   Write the process $G\lfloor x, y \rfloor$ in terms of replication (you have to invent the tecnique 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 π-calculus processes. Implement list append.

References

Books

- Robin Milner, Communicating and mobile systems: the π-calculus. (CUP,1999).
- Robin Milner, Communication and concurrency. (Prentice Hall,1989).
- Davide Sangiorgi, David Walker, The π-calculus: a theory of mobile processes. (CUP, 2001).

Tutorials available online:

- Robin Milner, The polyadic pi-calculus: a tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh.
- Joachim Parrow, An introduction to the pi-calculus. `http://user.it.uu.se/~joachim/intro.ps`
- Peter Sewell. Applied pi — a brief tutorial. Technical Report 498, University of Cambridge. `http://www.cl.cam.ac.uk/users/pes20/apppi.ps`