Pi-calculus

types, bestiary

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 

November 10, 2006
Plan (first part of the lecture)

Objective:

reason about concurrent systems using types.

Plan:

1. *Types to prevent run-time errors:*
   simply-typed pi-calculus, soundness, subtyping;

2. *Types to reason about processes:*
   typed equivalences, a labelled characterisation.
Types and sequential languages

In sequential languages, types are “widely” used:

- to detect simple programming errors at compilation time;
- to perform optimisations in compilers;
- to aid the structure and design of systems;
- to compile modules separately;
- to reason about programs;
- ahem, etc...

⇒ types are useful
Data types and pi-calculus

In pi-calculus, the only values are names. We now extend pi-calculus with base values of type int and bool, and with tuples.

Unfortunately (!?) this allows writing terms which make no sense, as

\[
\overline{x}(\text{true}).P \parallel x(y).\overline{z}(y + 1) \Rightarrow \text{bool} + \text{int}
\]

or (even worse)

\[
\overline{x}(\text{true}).P \parallel x(y).\overline{y}(4) \Rightarrow \text{send on bool}?
\]

These terms raise runtime errors, a concept you should be familiar with.
Preventing runtime errors

We know that 3 : int and true : bool.

Names are values (they denote channels). Question: in the term

\[ P \equiv \overline{x}(3).P' \]

which type can we assign to \( x \)?

*Idea*: state that \( x \) is a channel that can transport values of type int. Formally

\[ x : \text{ch(int)}. \]

A complete type system can be developed along these lines...
Simply-typed pi-calculus: syntax and reduction semantics

Types:

\[ T ::= \text{ch}(T) \mid T \times T \mid \text{unit} \mid \text{int} \mid \text{bool} \]

Terms (messages and processes):

\[ M ::= x \mid (M, M) \mid () \mid 1,2,... \mid \text{true} \mid \text{false} \]

\[ P ::= 0 \mid x(y : T).P \mid \overline{x}(M).P \mid P \parallel P \mid (\nu x : T)P \mid \text{match } z \text{ with } (x : T_1, y : T_2) \text{ in } P \mid !P \]

Notation: we write \( w(x, y).P \) for \( w(z : T_1 \times T_2).\text{match } z \text{ with } (x : T_1, y : T_2) \text{ in } P \).
Simply-typed pi-calculus: the type system

Type environment: $\Gamma ::= \emptyset \mid \Gamma, x:T$.

Type judgements:

- $\Gamma \vdash M : T$ value $M$ has type $T$ under the type assignment for names $\Gamma$;
- $\Gamma \vdash P$ process $P$ respects the type assignment for names $\Gamma$.

$P$ is well-typed in $\Gamma$
Simply-typed pi-calculus: the type rules (excerpt)

**Messages:**

\[ \begin{align*}
3 : \text{int} & \\
\Gamma(x) &= T \\
\Gamma \vdash x : T & \\
\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2 & \\
\Gamma \vdash (M_1, M_2) : T_1 \times T_2 & 
\end{align*} \]

**Processes:**

\[ \begin{align*}
\Gamma \vdash 0 & \\
\Gamma \vdash P_1 \quad \Gamma \vdash P_2 & \\
\Gamma \vdash P_1 \parallel P_2 & \\
\Gamma, x : T \vdash P & \\
\Gamma \vdash (\nu x : T)P & \\
\Gamma \vdash x : \text{ch}(T) \quad \Gamma, y : T \vdash P & \\
\Gamma \vdash x(y : T).P & \\
\Gamma \vdash x : \text{ch}(T) \quad \Gamma \vdash M : T \quad \Gamma \vdash P & \\
\Gamma \vdash \overline{x}(M).P & 
\end{align*} \]
Soundness

The soundness of the type system can be proved along the lines of Wright and Felleisen’s *syntactic approach to type soundness*. 

- One idea is to say being stuck = something went wrong (Ok, explicitly),
- extend the syntax with the `wrong` process, and add reduction rules to capture runtime errors:

\[
\begin{align*}
\bar{x}(M).P & \xrightarrow{\tau} \text{wrong} & \text{where } x \text{ is not a name} \\
x(y:T).P & \xrightarrow{\tau} \text{wrong} & \text{where } x \text{ is not a name}
\end{align*}
\]

- prove that if \( \Gamma \vdash P \), with \( \Gamma \) closed, and \( P \rightarrow^* P' \), then \( P' \) does not have `wrong` as a subterm.

\[\rightarrow \text{ progress - well-typed things can make progress, or it's a value} \]
\[\rightarrow \text{ preservation - well-typed things stay well-typed!} \]
Soundness, ctd.

Lemma Suppose that $\Gamma \vdash P$, $\Gamma(x) = T$, $\Gamma \vdash v : T$. Then $\Gamma \vdash P\{^v/x\}$.

Proof. Induction on the derivation of $\Gamma \vdash P$.

Theorem Suppose $\Gamma \vdash P$, and $P \xrightarrow{\alpha} P'$.

1. If $\alpha = \tau$ then $\Gamma \vdash P'$.
2. If $\alpha = a(v)$ then there is $T$ such that $\Gamma \vdash a : \text{ch}(T)$ and if $\Gamma \vdash v : T$ then $\Gamma \vdash P'$.
3. If $\alpha = (\nu \tilde{x} : \tilde{S})a\langle v \rangle$ then there is $T$ such that $\Gamma \vdash a : \text{ch}(T)$, $\Gamma, \tilde{x} : \tilde{S} \vdash v : T$, $\Gamma, \tilde{x} : \tilde{S} \vdash P'$, and each component of $\tilde{S}$ is a link type.

Proof. At the blackboard.
Subtyping

Idea: refine the type of channels $\text{ch}(T)$ into

\begin{align*}
i(T) & \quad \text{input (read) capability} \\
o(T) & \quad \text{output (write) capability}
\end{align*}

This form a basis for subtyping.

Example: the term

\[
x : o(o(T)) \vdash (\nu y : \text{ch}(T)) \ x(y).!y(z : T)
\]

is well-typed because $\text{ch}(T) <: o(T)$. Effect: well-typed contexts cannot interfere with the existing input, because they can only write at channel $y$. 
The subtyping relation, formally

- is a preorder

<table>
<thead>
<tr>
<th></th>
<th>Reflective</th>
<th>Transitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T &lt;: T$</td>
<td>$T_1 &lt;: T_2$ $T_2 &lt;: T_3$</td>
<td>$T_1 &lt;: T_3$</td>
</tr>
</tbody>
</table>

- capabilities can be forgotten

$\text{ch}(T) <: \text{i}(T)$

$\text{ch}(T) <: \text{o}(T)$

- $\text{i}$ is a covariant type constructor, $\text{o}$ is contravariant, $\text{ch}$ is invariant

$T_1 = \text{int}$

$T_2 = \text{real}$

$\text{i}(T_1) <: \text{i}(T_2)$

$\text{o}(T_1) <: \text{o}(T_2)$

$\text{ch}(T_1) <: \text{ch}(T_2)$

it is safe to receive \text{int} on a real channel

it must not send \text{real} when other side is expecting \text{int}
**Subtyping, ctd.**

*Intuition:* if \( x : o(T) \) then it is safe to send along \( x \) values of of a subtype of \( T \). Dually, if \( x : i(T) \) then it is safe to assume that values received along \( x \) belong to a supertype of \( T \).

Type rules must be updated as follows:

\[
\begin{align*}
\Gamma \vdash x : i(T) \quad \Gamma, y:T \vdash P \\
\Gamma \vdash x(y:T).P
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : o(T) \quad \Gamma \vdash M : T \quad \Gamma \vdash P \\
\Gamma \vdash \overline{x}(M).P
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : T_1 \quad T_1 <: T_2 \\
\Gamma \vdash M : T_2
\end{align*}
\]
Exercises

Show that: these are all well-typed.

1. \( a : \text{ch}(\text{int}), b : \text{ch}(\text{real}) \vdash \overline{a}\langle 5 \rangle \parallel a(x).\overline{b}\langle x \rangle, \) assuming \( \text{int} < : \text{real}; \)

2. \( x : \text{o}(\text{o}(T)) \vdash (\nu y : \text{ch}(T))(\overline{x}\langle y \rangle .!y(z)) \)

3. \( x : \text{o}(\text{o}(T)), z : \text{o}(\text{i}(T)) \vdash (\nu y : \text{ch}(T))(\overline{x}\langle y \rangle \parallel \overline{z}\langle y \rangle) \)

4. \( b : \text{ch}(S), x : \text{ch}(\text{i}(S)), a : \text{ch}(\text{o}(\text{i}(S))) \vdash \overline{a}\langle x \rangle \parallel x(y).y(z) \parallel a(x).\overline{x}\langle b \rangle \)
Remarks on i/o types

– different processes may have different visibility of a name:

\[
(\nu x : \text{ch}(T)) \ y\langle x \rangle. z\langle x \rangle. P \parallel y(a : i(T)). Q \parallel z(b : o(T)). R \rightarrow \rightarrow
(\nu x : \text{ch}(T)) (P \parallel Q\{x/a\} \parallel R\{x/b\})
\]

Q can only read from x, R can only write to x.

– acquiring the o and i capabilities on a name is different from acquiring ch:

the term

\[
(\nu x : \text{ch}(\text{unit})) \ y\langle x \rangle. z\langle x \rangle \parallel y(a : i(\text{unit})). z(b : o(\text{unit})). \overline{a}\langle \rangle
\]

is not well-typed.
Types for reasoning

Types can be seen as contracts between a process and its environment: the environment must respect the constraints imposed by the typing discipline.

In turn, types reduce the number of legal contexts (and give us more process equalities).

Example: an observer whose typing is

$$\Gamma = a : o(T), b : T, c : T'$$

$T$ and $T'$ unrelated

- can offer an output $\overline{a}\langle b \rangle$;
- cannot offer an output $\overline{a}\langle c \rangle$, or an input at $a$. 
A “natural” contextual equivalence, informally

Definition (informal): The processes $P$ and $Q$ are equivalent in $\Gamma$, denoted

$$P \simeq_{\Gamma} Q$$

iff $\Gamma \vdash P, Q$ and they are equivalent in all the testing contexts that respect the types in $\Gamma$.

To formalize this equivalence we need to type contexts, at the blackboard...
Semantic consequences of i/o types

Example: the processes

\[ P = (\nu x)\bar{a}(x).\bar{x} \]  
\[ Q = (\nu x)\bar{a}(x).0 \]

are and different in the untyped or simply-typed pi-calculus.

With i/o types, it holds that

\[ P \cong_\Gamma Q \quad \text{for } \Gamma = a : \text{ch(o(unit))} \]

because the residual \( \bar{x} \) of \( P \) is deadlocked (the context cannot read from \( x \)).
Semantic consequences of i/o types, ctd.

Specification and an implementation of the factorial function:

\[
\text{Spec} = ! f(x, r).\bar{r}\langle \text{fact}(x) \rangle
\]
\[
\text{Imp} = ! f(x, r).\text{if } x = 0 \text{ then } \bar{r}\langle 1 \rangle \text{ else } (\nu r').\bar{f}\langle x - 1, r' \rangle.r'(m).\bar{r}\langle x \times m \rangle
\]

In general, Spec \(\not\sim\) Imp. (Why?) \(\Rightarrow f \text{ is not private!}\)

With i/o types, we can protect the input end of the function, obtaining

\[
\Gamma \text{ protect } f
\]
\[
(\nu f)\bar{a}\langle f \rangle.\text{Spec} \equiv_{\Gamma} (\nu f)\bar{a}\langle f \rangle.\text{Imp}
\]

for \(\Gamma = a : \text{ch}(\text{o(int} \times \text{o(int)})\).
Semantic consequences of i/o types, ctd.

\[ P = (\nu x, y)(\bar{a}(x) \parallel \bar{a}(y) \parallel \textit{act}(x).R) \parallel \textit{act}(y).R) \]

\[ Q = (\nu x)(\bar{a}(x) \parallel \bar{a}(x) \parallel \textit{act}(x).R) \]

In the untyped calculus \( P \nsim Q \): a context that tells them apart is

\[- \parallel a(z_1).a(z_2).(z_1().\bar{c}) \parallel z_2(). \]

if \( z_1 = z_2 \), some interaction \( \text{act}(\text{act}()) \)

With i/o types

\[ P \sim_\Gamma Q \quad \text{for } \Gamma = a : \text{ch}(\text{o(unit)}) \]

Notation: I will often omit redundant type informations.
Exercise

1. Extend the syntax, the reduction semantics, and the type rules of pi-calculus with i/o types with the nondeterministic sum operator, denoted +;

2. Show that the terms

\[
P = \overline{b}(x).a(y).(y() \parallel \overline{x})
\]

\[
Q = \overline{b}(x).a(y).(y().\overline{x} + \overline{x}.y())
\]

are not equivalent in the untyped calculus. Propose a i/o typing such that \(P \simeq_{\Gamma} Q\).
References


Pierce, Sangiorgi: *Typing and subtyping for mobile processes*, LICS '93.

Boreale, Sangiorgi: *Bisimulation in name-passing calculi without matching*, LICS '98.

Sangiorgi, Walker: *The pi-calculus*, CUP.

...there is a large literature on the subject. The articles above have been reported because they are explicitly mentioned in this lecture.
Navigating through the literature

Pi-calculus literature describes \textbf{zillions} of slightly different languages, semantics, equivalencies.

Some slides for not getting lost.
Barbed congruence vs. reduction-closed barbed congruence

Let \emph{barbed equivalence}, denoted \( \cong^\bullet \), be the largest symmetric relation that is barb preserving and reduction closed. Barbed equivalence is not preserved by context, so define \emph{barbed congruence}, denoted \( \cong^c \), as

\[
\{(P, Q) : C[P] \cong^\bullet C[Q] \text{ for every context } C[-]\}.
\]

- Barbed congruence is \emph{more natural} and \emph{less discriminating} than reduction-closed barbed congruence (for pi-calculus processes).

- Completeness of bisimulation for image-finite processes holds with respect to barbed congruence, but its proof requires transifinite induction.
Late bisimulation

Change the definition of the LTS:

\[
x(y).P \xrightarrow{x(y)} P
\]

and extend the definition of bisimulation with the clause: if \( P \approx_l Q \) and \( P \xrightarrow{x(y)} P' \), then there is \( Q' \) such that \( Q \xrightarrow{x(y)} Q' \) and for all \( v \) it holds \( P' \{v/y\} \approx_l Q' \{v/y\} \).

- Late bisimulation differs (slightly) from (early) bisimulation. More importantly, the label \( x(y) \) does not denote an interacting context.
Ground bisimulation

Idea: play a standard bisimulation on the late LTS. Or,

Let ground bisimulation be the largest symmetric relation, $\approx_g$, such that whenever $P \approx_g Q$, there is $z \notin \text{fn}(P, Q)$ such that if $P \xrightarrow{\alpha} P'$ where $\alpha$ is $\bar{x}(y)$ or $x(z)$ or $(\nu z)\bar{x}(z)$ or $\tau$, then $Q \xrightarrow{\hat{\alpha}} \approx_g P'$.

Contrast it with bisimilarity: to establish $x(z).P \approx x(z).Q$ it is necessary to show that $P\{v/z\} \approx Q\{v/z\}$ for all $v$. Ground bisimulation requires to test only a single, fresh, name.

However, ground bisimilarity is less discriminating than bisimilarity, and it is not preserved by composition (still, it is a reasonable equivalence for sublanguages of pi-calculus).
Open bisimulation

Full bisimilarity is the closure of bisimilarity under substitutions, and is a congruence with respect to all contexts. Unfortunately, full bisimilarity is not defined co-inductively.

Question: can we give a co-inductive definition of a useful congruence?

Yes, with open bisimulation.

Idea: (on the restriction free calculus) let $\bowtie$ be the largest symmetric relation such that whenever $P \bowtie Q$ and $\sigma$ is a substitution, $P\sigma \xrightarrow{\alpha} P'$ implies $Q\sigma \xrightarrow{\hat{\alpha}} \bowtie P'$. It is possible to avoid the $\sigma$ quantification by means of an appropriate LTS.
Subcalculi

Idea: In pi-calculus contexts have a great discriminating power. It may be useful to consider other languages in which contexts "observe less", so that we have more equations.

Asynchronous pi-calculus: no continuation after an output prefix.

Localized pi-calculus: given \(x(y).P\), the name \(y\) is not used as subject of an input prefix in \(P\).

Private pi-calculus: only output of new names.
Conclusion: back to programming languages

Design choice:

bake into the definition of the language specific communication primitives?

• yes: Pict (Pierce et al.), NomadicPict (Sewell et al.), JoCaml (Moscova), ...

• no: Acute (Sewell et al., Moscova), ...

Some demos

...crossing fingers...