

# Concurrency theory

up-to techniques, functions as processes, types

Francesco Zappa Nardelli

INRIA Paris-Rocquencourt, MOSCOVA research team

`francesco.zappa_nardelli@inria.fr`

together with

Frank Valencia (INRIA Futurs)



Roberto Amadio (PPS)



Emmanuel Haucourt (CEA)



---

## How to prove...

To show that two processes are bisimilar, it is enough to find a bisimulation relating them. Easy?

Example: we want to show that (in the pi-calculus) bisimilarity is preserved by parallel composition. We naturally consider

$$\mathcal{R} = \{(P \parallel R, Q \parallel R) : P \approx Q\}$$

as a candidate bisimulation. But...

---

## The candidate bisimulation

1. may be larger than at first envisaged;
2. may be infinite;

example: to show that  $x(z).\bar{y}\langle z\rangle \approx (\nu w)(x(z).\bar{w}\langle z\rangle \parallel w(v).\bar{y}\langle v\rangle)$ , we must consider:

$$\begin{aligned} & \{(x(z).\bar{y}\langle z\rangle, (\nu w)(x(z).\bar{w}\langle z\rangle \parallel w(v).\bar{y}\langle v\rangle))\} \\ \cup & \{(\bar{y}\langle a\rangle, (\nu w)(\bar{w}\langle a\rangle \parallel w(v).\bar{y}\langle v\rangle)) : a \text{ arbitrary}\} \\ \cup & \{(\bar{y}\langle a\rangle, (\nu w)(\mathbf{0} \parallel \bar{y}\langle a\rangle)) : a \text{ arbitrary}\} \\ \cup & \{(\mathbf{0}, (\nu w)(\mathbf{0} \parallel \mathbf{0}))\} \end{aligned}$$

3. hard to guess;

which is the smallest bisimulation relating  $!!P$  and  $!P$ ?

4. awkward to describe and to work with...

---

## Up-to proof techniques

**Idea:** find classes of relations that:

1. are not themselves bisimulations;
2. can be *automatically* completed into bisimulations.

**Idea, explained:** if we had such a class then to prove that two processes are bisimilar it would be enough to exhibit a relation in this class<sup>1</sup> that contains the two processes.

---

<sup>1</sup>Hopefully, it is easier to find such relation than to find the candidate bisimulation directly.

---

## Warm-up: bisimulation up to structural congruence

A symmetric relation  $R$  is a *bisimulation up-to*  $\equiv$  if whenever  $P \mathcal{R} Q$  and  $P \xrightarrow{\ell} P'$  then there exists a process  $Q'$  such that  $Q \xrightarrow{\hat{\ell}} Q'$  and  $P' \equiv \mathcal{R} \equiv Q'$ .

*Exercise:* prove that if  $P \mathcal{R} Q$  and  $\mathcal{R}$  is a bisimulation up-to  $\equiv$ , then  $P \approx Q$ .

*Exercise:* prove that  $(\nu x)(\bar{x}\langle v \rangle \parallel x(u).P \parallel Q) \approx Q \parallel P$ .

*Exercise:* prove that  $!P \approx !!P$ .

---

## Bisimulation up to non-input context

A symmetric relation  $\mathcal{R}$  is a *bisimulation up-to non-input context* if whenever  $P \mathcal{R} Q$  and  $P \xrightarrow{\ell} P'$  then there exists a process  $Q'$  such that  $Q \xRightarrow{\hat{\ell}} Q'$  and there exist a *non-input context*  $C[-]$  and processes  $P''$  and  $Q''$  such that  $P' \equiv C[P'']$ ,  $Q' \equiv C[Q'']$ , and  $P'' \mathcal{R} Q''$ .

**Exercise:** Prove that if  $\mathcal{R}$  is a bisimulation up to non-input context, then

$$\{(C[P], C[Q]) : P \mathcal{R} Q \text{ and } C[-] \text{ is a non-input context}\}$$

is a bisimulation up to structural congruence.

**Exercise:** Prove that  $!P \parallel !P \approx !P$  (hint: show that the relation  $\mathcal{R} = \{(!P \parallel !P, !P)\}$  is a bisimulation up to non-input context).

---

## Alternative LTS rules for replication

It is often convenient to replace the rule:

$$\frac{P \parallel !P \xrightarrow{\ell} P'}{!P \xrightarrow{\ell} P'}$$

with the three rules:

$$\frac{P \xrightarrow{\ell} P'}{!P \xrightarrow{\ell} P' \parallel !P}$$

$$\frac{P \xrightarrow{\bar{x}\langle y \rangle} P_1 \quad P \xrightarrow{x(y)} P_2}{!P \xrightarrow{\tau} (P_1 \parallel P_2) \parallel !P}$$

$$\frac{P \xrightarrow{(\nu y)\bar{x}\langle y \rangle} P_1 \quad P \xrightarrow{x(y)} P_2}{!P \xrightarrow{\tau} (\nu y)(P_1 \parallel P_2) \parallel !P}$$

---

## Theorems about replication

The equivalence  $!P \parallel !P \approx !P$  shows that duplication of a replicable resource has no behavioural effect. Consider now

$$(\nu x)(P \parallel !x(y).Q)$$

We may call  $!x(y).Q$  a *private resource* of  $P$ . Suppose  $P \equiv P_1 \parallel P_2$ . It holds that

$$(\nu x)(P_1 \parallel P_2 \parallel !x(y).Q) \approx (\nu x)(P_1 \parallel !x(y).Q) \parallel (\nu x)(P_2 \parallel !x(y).Q)$$

provided that  $P_1$  and  $P_2$  never read over  $x$ .



---

## Function as processes: call-by-value lambda calculus

The call-by-value lambda calculus:

terms  $M, N ::= x \mid \lambda x.M \mid M N$

values  $V ::= x \mid \lambda x.M$

Reduction rules:

$$(\lambda x.M) V \rightarrow M\{V/x\}$$

$$\frac{M \rightarrow M'}{M N \rightarrow M' N}$$

$$\frac{N \rightarrow N'}{V N \rightarrow V N'}$$

---

## Lambda vs. pi

- Lambda-calculus substitutes *terms for variables*; pi-calculus relies only on name substitutions;
- In  $(\lambda x.M) N$  the terms  $\lambda x.M$  and  $N$  are committed to interacting with each other, even if they are part of a larger term; interference for the environment can prevent pi-calculus term to interact.
- Lambda-calculus is *sequential*: *parallel or cannot be defined in PCF*

$$\text{Por } M N = \begin{cases} \text{true if } M \text{ or } N \text{ reduces to true} \\ \text{diverges otherwise} \end{cases}$$

---

## Call-by-value CPS transform (Plotkin)

*Idea:* function calls terminate by passing their result to a continuation.

*Example:*

$$\lambda y. \text{let } g = \lambda n. n + 2 \text{ in } (g y) + (g y)$$

can be rewritten as

$$\lambda k y. \text{let } g = \lambda k n. k (n + 2) \text{ in } \\ g (\lambda v. g(\lambda w. k (v + w))) y) y$$

*Transformation:*

$$[[V]] = \lambda k. k[V]$$

$$[x] = x$$

$$[[M N]] = \lambda k. [[M]](\lambda v. [[N]](\lambda w. v w k))$$

$$[\lambda x. M] = \lambda x. [[M]]$$

---

## Call-by-value CPS transform, ctd.

*Exercise:* simulate the  $\beta$ -reduction

$$(\lambda x.M)V \rightarrow M\{V/x\}$$

after the CPS transform.

*Further readings:* if you want to know more about continuations, then some readings:

- Reynolds, *The discoveries of continuations*;
- Appel, *Compiling with continuations*;
- the works of Plotkin, Danvy, Gordon, Felleisen, etc...

(Remark: this list is wildly uncomplete).

---

## Encoding of call-by-value lambda into pi

*Notations:* Suppose  $[[M]]$  is  $(q).Q$ . Then  $y(x)[[M]]$  is  $y(x, q).Q$ , and  $[[M]]r$  is  $Q\{r/q\}$ .

*Encoding:*

$$[[\lambda x.M]] = (p).\bar{p}\langle y \rangle. !y(x)[[M]]$$

$$[[x]] = (p).\bar{p}\langle x \rangle$$

$$[[M N]] = (p).(\nu q)([[M]]q \parallel q(v).(\nu r)([[N]]r \parallel r(w).\bar{v}\langle w, p \rangle))$$

If  $M =_{\beta} V$ , then  $[[M]]p$  reduces to a process that, very roughly, **returns the value  $V$  at  $p$** . When  $V$  is a function, it cannot be passed directly at  $p$ : instead we pass a pointer to the function.

---

## And call-by-name?

The call-by-name lambda calculus:

terms  $M, N ::= x \mid \lambda x. M \mid M N$

Reduction rules:

$$(\lambda x. M) N \rightarrow M\{N/x\}$$
$$\frac{M \rightarrow M'}{M N \rightarrow M' N}$$

---

## Call-by-name CPS transform (Plotkin)

*Transform:*

$$[[x]] = \lambda k. x k$$

$$[[\lambda x.M]] = \lambda k. k (\lambda x. [[M]])$$

$$[[M N]] = \lambda k. [[M]] (\lambda v. v [[N]] k)$$

*Remark:* the encoding of a variable  $x$  is used as a trigger for activating a term and providing it with a location.

*Exercise:* simulate  $\beta$ -reduction in the transform.

---

## Encoding of call-by-name lambda into pi

*Notations:* Suppose  $[[M]]$  is  $(q).Q$ . Then  $y(x)[[M]]$  is  $x(y, q).Q$ , and  $[[M]]r$  is  $Q\{r/q\}$ .

*Encoding:*

$$[[\lambda x.M]] = (p).\bar{p}\langle v \rangle.v(x)[[M]]$$

$$[[x]] = (p).\bar{x}\langle p \rangle$$

$$[[M N]] = (p).(\nu q) ([[M]]q \parallel q(v).(\nu x)(\bar{v}\langle x, p \rangle.!x[[N]]))$$

A term that is evaluated may only be the operand – not the argument – of an application: as such it cannot be copied. Also, the argument is passed without being evaluated, and can be evaluated every time its value is needed.



---

## Other reduction strategies

It is possible to encode other reduction strategies. For instance, **parallel call-by-value**.

$$(\lambda x.M) V \rightarrow M\{V/x\} \qquad \frac{M \rightarrow M'}{M N \rightarrow M' N} \qquad \frac{N \rightarrow N'}{M N \rightarrow M N'}$$

We rely on parallel composition to perform parallel reduction of  $M$  and  $N$ :

$$[[M N]] = (p).(\nu q, r)([[M]]q \parallel [[N]]r \parallel q(v).r(w).\bar{v}\langle w, p\rangle)$$

---

## Observational congruence for call-by-name lambda

Since call-by-name is **deterministic**, the definition of observation equivalence (barbed congruence) can be simplified, by removing the bisimulation clause on interactions.

**Definition:** We say that  $M$  and  $N$  are *observationally equivalent*, if, in all closed contexts  $C[-]$ , it holds that  $C[M] \Downarrow$  iff  $C[N] \Downarrow$ .

A tractable characterisation of observation equivalence:

**Definition:** *Applicative bisimilarity* is the largest symmetric relation  $\approx_\lambda$  on lambda terms such that whenever  $M \approx_\lambda N$ ,  $M \rightarrow^* \lambda x.M'$  implies  $N \rightarrow^* \lambda x.N'$  with  $M'\{L/x\} \approx_\lambda N'\{L/x\}$  for all  $L$ .

Observation equivalence and applicative bisimilarity *coincide* (Stoughton).

---

## Soundness and non-completeness

Let  $M$  and  $N$  be two lambda terms. We write  $M =_{\pi} N$  if  $[[M]] \cong [[N]]$ .

**Theorem (soundness):** If  $M =_{\pi} N$  then  $M \approx_{\lambda} N$ .

We would not expect completeness to hold: pi-calculus contexts are potentially more discriminating than lambda-calculus contexts.

More in detail:

- the CPS transform itself is not complete: there are terms that are applicative bisimilar, but whose CPS images are distinguishable (and thus separated by appropriate pi-calculus contexts);
- the canonical model (Abramsky) is sound but not complete, because the model contains denotations of terms not definable (eg. the *parallel convergence test*), and whose addition increases the discriminating power of the contexts.

See Sangiorgi and Walker book, part 4, for more details.

---

## 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...

---

## 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

$$\bar{x}\langle\text{true}\rangle.P \quad || \quad x(y).\bar{z}\langle y + 1 \rangle$$

or (even worse)

$$\bar{x}\langle\text{true}\rangle.P \quad || \quad x(y).\bar{y}\langle 4 \rangle .$$

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

---

## Preventing runtime errors

We know that  $3 : \text{int}$  and  $\text{true} : \text{bool}$ .

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

$$P \equiv \bar{x}\langle 3 \rangle.P'$$

which type can we assign to  $x$ ?

*Idea:* state that  $x$  is a channel that can transport values of type  $\text{int}$ . Formally

$$x : \text{ch}(\text{int}) .$$

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):*

$$M ::= x \mid (M, M) \mid () \mid 1, 2, \dots \mid \text{true} \mid \text{false}$$
$$P ::= \mathbf{0} \mid x(y : T).P \mid \bar{x}\langle M \rangle.P \mid P \parallel P \mid (\nu x : T)P \\ \mid \text{match } z \text{ with } (x : T_1, y : T_2) \text{ in } P \mid !P$$

*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  $\Gamma$ .



---

## Simply-typed pi-calculus: the type rules (excerpt)

*Messages:*

$$\begin{array}{c} 3 : \text{int} \\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2}{\Gamma \vdash (M_1, M_2) : T_1 \times T_2} \end{array}$$

*Processes:*

$$\begin{array}{c} \Gamma \vdash \mathbf{0} \\ \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \parallel P_2} \quad \frac{\Gamma, x:T \vdash P}{\Gamma \vdash (\nu x : T)P} \\ \frac{\Gamma \vdash x : \text{ch}(T) \quad \Gamma, y:T \vdash P}{\Gamma \vdash x(y : T).P} \quad \frac{\Gamma \vdash x : \text{ch}(T) \quad \Gamma \vdash M : T \quad \Gamma \vdash P}{\Gamma \vdash \bar{x}\langle M \rangle.P} \end{array}$$

---

## Soundness

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

- extend the syntax with the `wrong` process, and add reduction rules to capture runtime errors:

$$\frac{\text{where } x \text{ is not a name}}{\bar{x}\langle M \rangle.P \xrightarrow{\tau} \text{wrong}}$$

$$\frac{\text{where } x \text{ is not a name}}{x(y:T).P \xrightarrow{\tau} \text{wrong}}$$

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

---

## 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  $\alpha = (\nu \tilde{x} : \tilde{S})\bar{a}\langle v \rangle$  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.

---

## pi-calculus is hardly ever used untyped

Types can be enriched in many ways.

- *Familiar type constructs:* union, record, variants, basic values, functions, linearity, polymorphsim, . . .
- *Type constructs specific to processes:* process-passing, receptiveness, deadlock-freedom, sessions, . . .

Remark: types are assigned to names.

(More on types next Thursday).