Reasoning about concurrent systems using types

Davide Sangiorgi

INRIA Sophia Antipolis
2004 route des Lucioles, BP 93
06902 Sophia Antipolis CEDEX

Email: `davide.sangiorgi@inria.fr`
`http://www-sop.inria.fr/meije/personnel/Davide.Sangiorgi.html`

March 99

CONTENTS

- to detect simple kinds of programming errors at compilation time;
- to perform optimisations in compilers;
- to aid the structure and design of systems;
- to compile modules separatly;
- to reason about programs;
- Etc...

- In **sequential languages**: type systems are widely used and generally well-understood.

- In **concurrent languages**: the tradition of type systems is much less established. The research area is nowadays very active.

In this talk: types for reasoning about processes. I will focus on the most basic behavioural property: behavioural equivalence.

As a paradigmatical process calculus I will use the π-**calculus**.

CONTENTS

The untyped π-calculus (core)

$a, b, \ldots, x, y, z, \ldots$	*Names*
	Processes
$P \quad ::= \quad \mathbf{0}$	nil process
$\quad \mid \quad P \mid P$	parallel
$\quad \mid \quad (\boldsymbol{\nu} a)\, P$	restriction
$\quad \mid \quad a(x).\, P$	input
$\quad \mid \quad \overline{a}\, b\,.\, P$	output
$\quad \mid \quad !P$	replication ($= P \mid P \ldots$)

- Transitions: $P \xrightarrow{\tau} P'$ $\quad P \xrightarrow{ab} P'$ $\quad P \xrightarrow{\overline{a}\, b} P'$ $\quad P \xrightarrow{(\boldsymbol{\nu} b)\, \overline{a}\, b} P'$

- Basic communication rules:

$$a(y).\, Q \mid \overline{a}\, b\,.\, P \xrightarrow{\tau} Q\{^{b}\!/\!_{y}\} \mid P$$

$$a(y).\, Q \mid (\boldsymbol{\nu} b)\, \overline{a}\, b\,.\, P \xrightarrow{\tau} (\boldsymbol{\nu} b)\, (Q\{^{b}\!/\!_{y}\} \mid P) \qquad b \text{ fresh.}$$

The polyadic π-calculus (first try...)

$$a, b, \ldots, x, y, z, \ldots \qquad \textit{Names}$$

Processes

$$
\begin{array}{lcll}
P & ::= & \mathbf{0} & \text{nil process} \\
 & \mid & P \mid P & \text{parallel} \\
 & \mid & (\boldsymbol{\nu} a)\, P & \text{restriction} \\
 & \mid & a(\widetilde{x}).\, P & \text{input} \\
 & \mid & \overline{a}\langle \widetilde{b} \rangle.\, P & \text{output} \\
 & \mid & !P & \text{replication}
\end{array}
$$

Abbreviations:

$$
\begin{array}{rcl}
\overline{x}.\, P & \equiv & \overline{x}\langle\rangle.\, P \\
x.\, P & \equiv & x().\, P \\
\overline{x}\langle \widetilde{y} \rangle & \equiv & \overline{x}\langle \widetilde{y} \rangle.\, \mathbf{0} \\
\longrightarrow & \equiv & \xrightarrow{\tau} \\
\Longrightarrow & \equiv & \longrightarrow^{*} \\
\xRightarrow{\mu} & \equiv & \Longrightarrow \xrightarrow{\mu} \Longrightarrow
\end{array}
$$

Example: encoding boolean values

$$\begin{array}{rcl} \mathtt{true}_a & \stackrel{\text{def}}{=} & a(x,y).\,\overline{x} \\ \mathtt{false}_a & \stackrel{\text{def}}{=} & a(x,y).\,\overline{y} \end{array}$$

An if-then-else process, meeting $\mathtt{true}_a$ (assume p, q fresh):

$$\begin{array}{rl} & \mathtt{true}_a \mid (\boldsymbol{\nu} p,q)\, \overline{a}\langle p,q\rangle.\,(p.\,P \mid q.\,Q) \\ = & a(x,y).\,\overline{x} \mid (\boldsymbol{\nu} p,q)\, \overline{a}\langle p,q\rangle.\,(p.\,P \mid q.\,Q) \\ \longrightarrow & (\boldsymbol{\nu} p,q)\,(\overline{p} \mid p.\,P \mid q.\,Q) \\ \longrightarrow & (\boldsymbol{\nu} p,q)\,(\mathbf{0} \mid P \mid q.\,Q) \\ \sim & P \end{array}$$

($\sim$ indicates the application of some simple garbage-collection laws)

Run-time errors

$$\underline{\overline{x}\langle y, z\rangle}.\, P \mid \underline{x(h)}.\, Q \longrightarrow \texttt{wrong}$$

Assigning an arity to each channel is not enough:

$$\begin{array}{rcl} \overline{x}\langle y\rangle.\,\overline{y}\langle z_1, z_2\rangle.\, P \mid x(h).\, h(k).\, Q & \longrightarrow & \\ \underline{\overline{y}\langle z_1, z_2\rangle}.\, P \mid \underline{y(k)}.\, (Q\{{}^{y}\!/\!{}_{k}\}) & \longrightarrow & \texttt{wrong} \end{array}$$

Against errors, we need to **type** names.

CONTENTS

Polyadic π-calculus, typed

			Values
v	$::=$	x	name
	$\mid$	$\langle v_1 \ldots v_n \rangle$	tuple of values
			Processes
P	$::=$	$v(x).\,P$	input
	$\mid$	$\overline{v}\, v\,.\,P$	output
	$\mid$	with $(x_1 \ldots x_n) = v$ do P	tuple destructor
	$\mid$	$\ldots$	$\ldots$
			Types
T	$::=$	$\sharp\, T$	channel type
	$\mid$	$\langle T_1 \ldots T_n \rangle$	tuple types
	$\mid$	$\ldots$	(possibly) base types, recursive types
			Type environments
Γ	$::=$	$\Gamma, x : T$	
	$\mid$	$\emptyset$	

Additional operational rules:

$$\text{with } (x_1 \mathinner{\ldotp\ldotp} x_n) = \langle v_1 \mathinner{\ldotp\ldotp} v_n \rangle \text{ do } P \longrightarrow P\{{}^{v_1 \mathinner{\ldotp\ldotp} v_n}\!/\!_{x_1 \mathinner{\ldotp\ldotp} x_n}\}$$

We can then define as an **abbreviation**:

$$y(x_1 \mathinner{\ldotp\ldotp} x_n).\, P \stackrel{\text{def}}{=} y(z).\ \text{with } (x_1 \mathinner{\ldotp\ldotp} x_n) = z \text{ do } P$$

Type judgements

$$\Gamma \vdash v : T \qquad\qquad \Gamma \vdash P$$

Meaning:

"value v has type T under the type assignment for names Γ",

"process P respects the type assignment for names Γ".

Notation: In this talk, we are not concerned about type inference,
$\Rightarrow$ sometimes add type annotations to bound names.

A sample of typing rules

values:

$$\frac{\Gamma(v) = T}{\Gamma \vdash v \,:\, T} \qquad \frac{\Gamma \vdash v_i \,:\, T_i}{\Gamma \vdash \langle v_1 .. v_n \rangle \,:\, \langle T_1 .. T_n \rangle}$$

processes:

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \frac{\Gamma \vdash v \,:\, \langle T_1 .. T_n \rangle \qquad \Gamma, x_1 : T_1 .. x_n : T_n \vdash P}{\Gamma \vdash \text{ with } (x_1 .. x_n) = v \text{ do } P}$$

$$\frac{\Gamma \vdash w \,:\, \sharp T \qquad \Gamma \vdash v \,:\, T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{w}\, v \,.\, P} \qquad \frac{\Gamma \vdash w \,:\, \sharp T \qquad \Gamma, y : T \vdash P}{\Gamma \vdash w(y).\, P}$$

π-calculus is hardly ever used untyped!

Types can be enriched in many ways.

– **Familiar type constructs** (in place of, or in addition to, products), ex: union, records, variants, basic values, functions, linearity, polymorphism, etc.

– Type constructs **specific to processes**, ex: process-passing, receptiveness, deadlock-freedom, sessions, etc.

– **Types are assigned to names**, though the type of a name may also be seen as the type of the abstractions located at that name [cf: Blue calculus]

CONTENTS

Channel types ($\sharp\, T$) are refined by introducing:

$\mathsf{i}\, T$	input (read) capability
$\mathsf{o}\, T$	output (write) capability

$\Rightarrow$ basis for **subtyping**

Example: $x : \mathsf{o}\, \mathsf{o}\, T \vdash (\boldsymbol{\nu} y : \sharp\, T)\, (\overline{x}\, y\, .\, !y(z).\, \mathbf{0})$ (true, since $\sharp\, T < \mathsf{o}\, T$)

N.B.: The recipient of y cannot **interfere** with the existing input.

i —	$\Rightarrow$	covariance
o —	$\Rightarrow$	contravariance
$\sharp$ —	$\Rightarrow$	invariance

Analogy with mutable variables

$\texttt{ref}\ T$, like $\sharp T$, is invariant.

A variable is used in two ways: $\begin{cases} \text{R-value (read)} \\ \text{L-value (write)} \end{cases}$

If we separate them $\Rightarrow$ the same subtyping as for channels. [Reynolds's Forsythe]

Some properties of typing and subtyping

Narrowing:

If $\Gamma, p : S \vdash P$ and $T < S$, then also $\Gamma, p : T \vdash P$

Subject Reduction:

Suppose $\Gamma \vdash P$ and Γ is closed.

– If $P \longrightarrow P'$, then $\Gamma \vdash P'$.

– If $P \stackrel{(\boldsymbol{\nu}\widetilde{x}:\widetilde{S})\,\overline{a}\,v}{\longrightarrow} P'$, then there is T such that

- $\Gamma \vdash a : \mathsf{o}\, T$;
- $\Gamma, \widetilde{x} : \widetilde{S} \vdash v : T$;
- $\Gamma, \widetilde{x} : \widetilde{S} \vdash P'$.

– and the clause for input...

Types for reasoning

Types: *contracts* between a process and its environment $\Rightarrow$ reduces the number of "legal" contexts. $\Rightarrow$ more process equalities.

For now, take behavioural equivalence to mean:

Two processes are *behaviourally equivalent* if no contexts of the language can tell them apart.

Notation for behavioural equivalence:

$$P \approx_\Gamma Q \qquad \begin{cases} \Gamma \vdash P, Q\ ; \\ \text{if the language is untyped, } \Gamma \text{ is trivial} \end{cases}$$

The testing contexts must respect the types in Γ

Semantic consequences of i/o types

Example 1

$$
\begin{array}{lcl}
P & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} x)\, \overline{a}\, x \,.\, \overline{x} \\
Q & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} x)\, \overline{a}\, x \,.\, \mathbf{0}
\end{array}
$$

In untyped or simply-typed π-calculus: $P \not\approx_\Gamma Q, \qquad \forall \quad \Gamma$

With i/o types: $P \approx_\Gamma Q, \quad$ for $\Gamma \stackrel{\text{def}}{=} a : \sharp\, \mathtt{o}\, \mathtt{unit}$

Example 2

$$
\begin{array}{lcl}
P & \stackrel{\mathsf{def}}{=} & (\boldsymbol{\nu} x\,,\,y)\,\overline{a}\,x\,.\,\overline{a}\,y\,.\,(!x.\,R \mid !y.\,R) \\
Q & \stackrel{\mathsf{def}}{=} & (\boldsymbol{\nu} x)\,(\overline{a}\,x \mid \overline{a}\,x \mid !x.\,R)
\end{array}
$$

You may read: $\begin{array}{lcl} P & \stackrel{\mathsf{def}}{=} & \mathsf{let}\ x = R\ \mathsf{in}\ (\mathsf{let}\ y = R\ \mathsf{in}\ f[x,y]) \\ Q & \stackrel{\mathsf{def}}{=} & \mathsf{let}\ x = R\ \mathsf{in}\ f[x,x] \end{array}$

In untyped or simply-typed π-calculus: $P \not\approx_\Gamma Q, \quad \forall\, \Gamma$

A context that distinguishes them:

$$
[\cdot] \mid a(z_1).\,a(z_2).\,(z_1.\,\overline{c} \mid \overline{z_2})
$$

With i/o types: $P \approx_\Gamma Q$.

if $\Gamma \vdash a\ :\ \mathtt{o\,o\,unit}$

Example 3: the factorial function

$$\mathrm{SPEC} \stackrel{\mathsf{def}}{=} \,!f(x,r).\,\overline{r}\langle \mathit{fact}\,(x)\rangle$$

$$\begin{array}{lll}\mathrm{IMP} \stackrel{\mathsf{def}}{=} \,!f(x,r). & \texttt{if } x = 0 & \texttt{then } \overline{r}\,1 \\ & & \texttt{else } (\boldsymbol{\nu} r')\,(\overline{f}\langle x-1, r'\rangle.\,r'(m).\,\overline{r}\langle x * m\rangle)\end{array}$$

In a simply-typed π-calculus: $\mathrm{SPEC} \not\approx_{f:T} \mathrm{IMP}$ (for all T)

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

$$(\boldsymbol{\nu} f)\,\overline{a}\,f\,.\,\mathrm{SPEC} \approx_{\Gamma} (\boldsymbol{\nu} f)\,\overline{a}\,f\,.\,\mathrm{IMP}$$

if $\Gamma \stackrel{\mathsf{def}}{=} a : \ \sharp\, \mathsf{o}\,(\mathtt{Int} \times \mathsf{o}\,\mathtt{Int})$

i/o **subtyping validates the subtyping rule for functions:**

Following Milner,

$$[\![\lambda x.\, M]\!]_p \stackrel{\mathsf{def}}{=} (\boldsymbol{\nu} x)\, \overline{p}\, x\, .\, !x(z, y).\, [\![M]\!]_y$$

If $\lambda x.\, M : S \to T$, then $p : \mathsf{o}\,\mathsf{o}\,\langle [\![S]\!], \mathsf{o}\,[\![T]\!] \rangle$

which gives this translation on arrow types:

$$[\![S \to T]\!] \stackrel{\mathsf{def}}{=} \mathsf{o}\,\langle [\![S]\!], \mathsf{o}\,[\![T]\!] \rangle$$

where $[\![S]\!]$ is in contravariant position, $[\![T]\!]$ is in covariant position.

Polymorphism (intermezzo)

Type systems for parametric polymorphism constrain the power of observers by preventing them from directly manipulating data values whose types are abstract.

Polymorphic types can be used to hide certain implementation details of processes [cf: Mitchell and Plotkin's representation of data types in the λ-calculus]

The combination of polymorphism and aliasing has surprising effects (information leakage).

No time to cover this topic, ... see references.

CONTENTS

A local environment:

$$\boldsymbol{\nu} x\ (!x(y).\, R \mid Q)$$

(Q and R have only the output capability on x)

Name x is "stateless", and its input end is always available.
$\Rightarrow$ x is *uniformly receptive*.

Uniform receptiveness occurs when modelling functions, objects, communications of processes, RPC protocols, etc...

In π-calculus-based programming languages (Pict, Join, Blue): **def** $x(y) = R$ **in** Q

A simple variant: *linear receptiveness*: x is used at most once.

Semantic consequences of receptive types: some laws

if x is uniformly receptive:

$$\overline{y}\, x\,.\, P \quad \approx_\Gamma \quad (\boldsymbol{\nu} z)\, \overline{y}\, z\,.\,(z \multimap x \mid P)$$

where z is fresh

$$z \multimap x \stackrel{\text{def}}{=} !z(a).\,\overline{x}\, a \qquad \textit{(a link)}$$

(an output of a global name becomes an output of a fresh name)

Not true in the ordinary π-calculus because of contexts like

$$y(u).\,\overline{u}\, c\,.\,\overline{p} \mid [\cdot]$$

If x is uniformly receptive in Γ:

$$\boxed{\overline{x}\, v\,.\, P \approx_{\Gamma} \overline{x}\, v \mid P}$$

(makes a synchronous communication into an asynchronous one)

If $P \longrightarrow P'$ by means of a communication at x, then

$$P \approx_{\Gamma} P'$$

(insensitiveness to internal reductions)

Note: all names uniformly receptive $\Rightarrow$ **confuence**

A distributivity law for local environments:

$$\boxed{\begin{array}{rlrl} \mathsf{def}\ x(y) = R\ \mathsf{in}\ (P \mid Q) & \approx_\Gamma & & \mathsf{def}\ x(y) = R\ \mathsf{in}\ P \\ & & \mid & \mathsf{def}\ x(y) = R\ \mathsf{in}\ \ Q \end{array}}$$

i/o or receptive types are necessary for its validity.

More factorial functions:

$r : \ell\mathtt{rec}(\mathtt{Int}) \stackrel{\mathsf{def}}{=} r$ is a linear receptive channel that carries integers

$$
\begin{array}{rcl}
S1 & \stackrel{\mathsf{def}}{=} & !f(x : \mathtt{Int}, y : \mathtt{Int}, r : \ell\mathtt{rec}(\mathtt{Int})). \\
 & & \quad \mathtt{if}\ \ x = 0 \quad \mathtt{then}\ \overline{r}\, y \\
 & & \qquad\qquad \mathtt{else}\ \ (\boldsymbol{\nu} r')\ (\overline{f}\langle x-1, x*y, r'\rangle.\, r'(m).\, \overline{r}\, m\ )
\end{array}
$$

$$
\begin{array}{rcl}
S2 & \stackrel{\mathsf{def}}{=} & !f(x : \mathtt{Int}, y : \mathtt{Int}, r : \ell\mathtt{rec}(\mathtt{Int})). \\
 & & \quad \mathtt{if}\ \ x = 0 \quad \mathtt{then}\ \overline{r}\, y \\
 & & \qquad\qquad \mathtt{else}\ \ \overline{f}\langle x-1, x*y, r\rangle
\end{array}
$$

Taking account of the receptiveness of f, r and r':
$\Rightarrow$ $S1$ and $S2$ become behaviourally equivalent. [**Tail-call optimisation**]

CONTENTS

How can we prove those equalities?

An extensional definition of behavioural equivalence:

$P \Downarrow_a \stackrel{\text{def}}{=} P$ is capable of performing an input or output at a.

barbed congruence, $\approx_\Gamma$**:**

Let $\stackrel{\bullet}{\approx}$ (**barbed bisimulation**) be the largest symmetric relation on processes such that $P \stackrel{\bullet}{\approx} Q$ implies:

1. if $P \Longrightarrow P'$ then there is Q' such that $Q \Longrightarrow Q'$ and $P' \stackrel{\bullet}{\approx} Q'$;

2. for each channel a, if $P \Downarrow_a$ then also $Q \Downarrow_a$.

Barbed congruence $\stackrel{\text{def}}{=}$ the congruence induced by barbed bisimulation.

Problem (for proofs): The universal quantification on contexts.

Possible remedy techniques:

1. Context lemmas

2. *Intensional* characterisations

3. Algebraic laws

In this talk: 2 (the most powerful technique), for the case of i/o types.

Example: an intensional definition in CCS

observation equivalence (or weak bisimulation), $\simeq$

$P \simeq Q$ implies

– if $P \stackrel{\mu}{\Longrightarrow} P'$ then there exists Q' s.t. $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P' \simeq Q'$ (for $\mu \neq \tau$);

– $\ldots$

$\Rightarrow$ equality between the synchronisation trees of P and Q (precisely: equality as non-well-founded sets).

Theorem In CCS (with guarded sums), barbed congruence and observation equivalence coincide.

Observation equivalence

– coinductive definition

– no quantification on contexts

– "up to techniques" can enhance the coinduction proof method

– used in tools for mechanically checking equalities

– various characterisations: **non-well founded sets** [Aczel], **modal logic** [Hennessy, Milner], **final coalgebras** [Rutten, Turi, Honsell, Lenisa, Montanari, Pistore...] **domain theory** [Abramsky], **open maps** [Joyal, Nielsen, Winskel, Fiore, Cattani, Power,...], Similar characterisations for the untyped and simply-typed π-calculi (**early bisimulation**).

Problem:
beyond simply-typed π-calculus the standard intensional definitions are far too strong!

In Examples 1 and 2 of i/o types, regardless of the types of names:

$$
\begin{array}{rcc}
(\boldsymbol{\nu} x)\, \overline{a}\, x \,.\, \overline{x} & \xrightarrow{\boldsymbol{\nu} x\, \overline{a}\, x} & \xrightarrow{\overline{x}} \\
(\boldsymbol{\nu} x)\, \overline{a}\, x \,.\, \mathbf{0} & \xrightarrow{\boldsymbol{\nu} x\, \overline{a}\, x} & \mathbf{0}
\end{array}
$$

$$
\begin{array}{rcc}
(\boldsymbol{\nu} x, y)\, \overline{a}\, x \,.\, \overline{a}\, y \,.\, (!x.\, R \mid !y.\, R) & \xrightarrow{\boldsymbol{\nu} x\, \overline{a}\, x} & \xrightarrow{\boldsymbol{\nu} y\, \overline{a}\, y} \\
(\boldsymbol{\nu} x)\, \overline{a}\, x \,.\, \overline{a}\, x \,.\, (!x.\, R) & \xrightarrow{\boldsymbol{\nu} x\, \overline{a}\, x} & \xrightarrow{\overline{a}\, x}
\end{array}
$$

N.B: the processes that we want to prove equal are not even trace equivalent!

What is observation equivalence

(ie, labeled bisimilarity) in types calculi?

We shall replace the usual (very informal) interpretation of bisimulation:

Bisimilar processes *perform the same actions* and their derivatives are again bisimilar.

with

Bisimilar processes *react to the same actions offered by the observer* and their derivatives are again bisimilar.

Idea (first try): represent the observer as a type environment.

Example: An observer

$$\Gamma \stackrel{\mathsf{def}}{=} a : \mathtt{o}\,\mathtt{Int}, b : \mathtt{Int}, c : \mathtt{Bool}$$

– can offer an output $\overline{a}\, b$

– cannot offer an output $\overline{a}\, c$, or an input at a.

We look for something like $\Gamma \triangleright P \approx Q$, meaning

> "P and Q cannot be distinguished by a well-typed observer that respects the type constraints in Γ".

Note: the type environment for the observer need not coincide with that of the processes being tested

Up to now, everything looks straightforward but *agh!* .... **aliasing**!

Going back to Example 2 (page 17):

$$\begin{array}{lcl} a(p).\,b(q).\,R \mid P & \Longrightarrow & R\{x/p\}\{y/q\} \mid P' \\ \underbrace{a(p).\,b(q).\,R}_{\text{the tester}} \mid Q & \Longrightarrow & R\{x/p\}\{x/q\} \mid Q' \end{array}$$

At the beginning: the tester is the same

At the end: the tester processses are different

But in the bisimulation game the observer/tester should be unique!

> Solution: separate the observer's view on the identity of names from their real identity.

- *Typed closures*:

$$[\Delta\,;\,\sigma] \qquad \begin{cases} \Delta \text{ (a typing environment) is the observer's perspective;} \\ \sigma \text{ (a function from names to names) is the ``God's-eye'' perspective.} \end{cases}$$

- A transition relation on closures (*allow relation*):

$$[\Delta\,;\,\sigma] \xrightarrow[\delta]{\mu} [\Delta'\,;\,\sigma'] \qquad \begin{cases} \delta & : & \text{the observer's point of view on the action,} \\ & & \text{he/she offers (n.b.: } \delta \text{ respects the types in } \Delta\text{).} \\ \mu = \overline{\delta\sigma} & : & \text{the concrete action of the tested process} \end{cases}$$

Example:

$$\begin{array}{rcl} \Delta & \stackrel{\text{def}}{=} & x : \mathsf{o}\,T, y : T, p : \mathsf{i}\,T \\ \sigma & \stackrel{\text{def}}{=} & \{a/x, b/y, p/p, \} \end{array}$$

$$[\Delta\,;\,\sigma] \quad \xrightarrow[\overline{x}\,y]{ab} \quad [\Delta\,;\,\sigma] \qquad\qquad [\Delta\,;\,\sigma] \quad \xrightarrow[p\,z]{\overline{p}\,b} \quad [\Delta, z : T\,;\,\sigma\{b/z\}] \quad (z \text{ fresh})$$

A closure $[\Delta\,;\,\sigma]$ and a process P form a *configuration* $[\Delta\,;\,\sigma]\ \#\ P$, under the condition that P's typing 'agrees' with $[\Delta\,;\,\sigma]$.

$$[\Delta\,;\,\sigma]\#P \overset{\delta}{\Longrightarrow} [\Delta'\,;\,\sigma']\#P'$$

$$\stackrel{\text{def}}{=}$$

$$[\Delta\,;\,\sigma] \xrightarrow[\delta]{\mu} [\Delta'\,;\,\sigma'] \text{ and } P \overset{\mu}{\Longrightarrow} P'.$$

Two configurations $[\Delta_1\,;\,\sigma_1]\#P_1$ and $[\Delta_2\,;\,\sigma_2]\#P_2$ are *compatible* if $\Delta_1 = \Delta_2$.

Typed bisimilarity:

The largest symmetric relation $\simeq$ on compatible configurations such that $H \simeq K$ implies

1. if $H \stackrel{\delta}{\Longrightarrow} H'$ then there is K' s.t. $K \stackrel{\delta}{\Longrightarrow} K'$ and $H' \simeq K'$;

2. if $H \stackrel{\langle \mu, \lambda \rangle}{\Longrightarrow} H'$ then there is K' s.t.

   – either $K \stackrel{\langle \mu, \lambda \rangle}{\Longrightarrow} K'$,
   – or $K \stackrel{\mu}{\Longrightarrow} \diamond \stackrel{\lambda}{\Longrightarrow} K'$,

   and $H' \simeq K'$.

Aside the second clause (needed in certain rather special cases; let's ignore it for now...), it looks similar to the standard definition of bisimulation; the meaning of "matching transitions" is however quite different.

Theorem Typed bisimilarity coincides with typed barbed congruence (on image-finite processes.)

Is image-finiteness necessary?

N.B: on open processes, as usual, we also need to close under substitutions

CONTENTS

Some references

Polyadic pi-calculus: Milner 91.

i/o **types**: Pierce and Sangiorgi 93, Boreale and Sangiorgi 98, Merro 98. **Extensions**: De Nicola, Ferrari and Pugliese 98, Sewell 98, Hennessy and Riely 98-9, Gay and Hole 99, Röckl 99.

Linearity: Honda 96, Kobayashi, Pierce and Turner 96, Niehren 96, Amadio 97.

Types for confluence: Steffen and Nestmann 95.

Receptiveness: Pierce and Turner 96, Sangiorgi 97, Kobayashi 97.

Types to prevent deadlocks: Yoshida 96, Boudol 97, Kobayashi and Sumii 97-8, Ravara and Vasconcelos 97.

Types for guaranteeing secrecy properties (spi-calculus): Abadi 97.

A general framework for types: Honda 96.

Etc.

More references

Experimental typed programming languages inspired by the π-calculus: Pict (Pierce and Turner), Join (Lévy's group), Blue (Boudol and Dal-Zilio), and Tyco (Vasconcelos's group).

Applications to **object-oriented languages**: Kleist and Sangiorgi 98-9, Dal-Zilio 98, Hüttel, Kleist,Merro, Nestmann 99.

Other work

Here the focus is more specifically on **safety or security properties:**

Type systems to control information flow: Smith and Volpano 96-8, Heintze and Riecke 98, Leroy and Rouaix 98,

Proof-carrying code: Necula and Lee 98.

Race conditions and deadlock: Flanagan and Abadi 99.

Interaction Categories: Abramsky, Gay and Nagarajan 93-95.

Ambients: Cardelli, Ghelli and Gordon 98-9.

Etc.

Future?

- Decidable typed behavioural equivalences
- Logics for typed behavioural equivalences
- Axiomatisations of typed behavioural equivalences
- A typed bisimilarity parametric on the type system
- Computer-aided assistance
- Larger proofs