Concurrency theory

Francesco Zappa Nardelli

INRIA Rocquencourt, MOSCOVA research team

`francesco.zappa_nardelli@inria.fr`

together with

Frank Valencia (INRIA Futurs)   Catuscia Palamidessi (INRIA Futurs)   Roberto Amadio (PPS)

Proving theorems is easy, writing programs is hard!

Anonymous

```
static void copy (char[] s, int i, char[] d, int j, int len)
{
    for (int k = 0; k < len; ++k)
        d[j + k] = s[i + k];
}
```

Claim: the function `copy` copies `len` characters of `s` starting from offset `i` into `d` starting from offset `j`.

...not quite...

Writing program is hard, ctd.

```
P_M_DERIVE(T_ALG.E_BH) :=
    UC_16S_EN_16NS (TDB.T_ENTIER_16S
        ((1.0/C_M_LSB_BH) *
        G_M_INFO_DERIVE(T_ALG.E_BH)))
```

Claim: this instruction multiplies the 64 bits float `G_M_INFO_DERIVE(T_ALG.E_BH)` by a constant and converts the result to a 16 bit unsigned integer.

(in ADA function calls and array access share the same notation).

...unfortunately...

Around 1949...

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realised that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

Sir Maurice Wilkes (1913 -)

..in 2005?

"How to ensure that a system behaves correctly with respect to some specification (implicit ot explicit)?"

Answer: *formal methods*.

. . . types, constraints,
denotational semantics, abstract interpretation,
mathematical logic, concurrency,
new programming languages, . . .

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 (selected)

- Local concurrency: shared memory, semaphores, communication, ...
- Marshalling: choice of distributed abstractions and trust assumptions, ...
- 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, ...

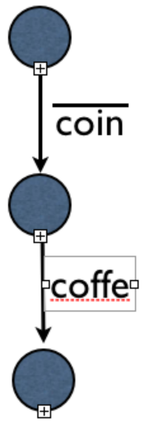
  cf Erlang [AVWW96], Telescript, Facile [TLK96,Kna95], Obliq [Car95], CML [Rep99], Pict [PT00], JoCaml [JoC03], Alice [BRS+05], Esterel [Ber98], ...

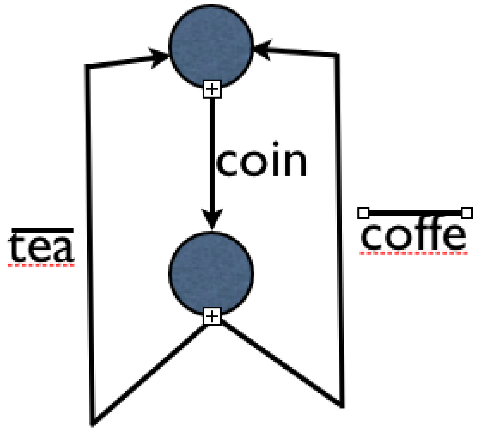
Concurrency theory

Set out to understand some key concepts, reveal then essential unities, classify the important variations, behind the wide variation of concurrent systems.

Nondeterminism and Interaction

Idea: represent the sequential agents of a concurrent system using automatas.





$U = \overline{\text{coin}}.\text{coffe}.\mathbf{0}$ $\qquad$ $M = \text{coin}.(\overline{\text{coffe}}.M + \overline{\text{tea}}.M)$

From automata to CCS

Given an automata,

1. remove final (we are not primarily interested in termination) state,
2. remove initial states (assimilate processes with states, hence any state is "initial" relative to the process).

Such an automaton deprived from initial and final states is called a

labelled transition system

or LTS for short.

From automata to CCS, ctd.

Definition: an LTS is given by:

- a finite set of states, $P, Q, \ldots$;
- a finite alphabet `Act` whose members are called actions, ranged over by μ;
- transitions between them, written $P \xrightarrow{\mu} Q$.

From automata to CCS, ctd.

A LTS together with one of its states, that is, a process, can be described by the following syntax:

$$P \quad ::= \quad \Sigma_{i \in I} \mu_i . P_i \quad \Big| \quad \text{let } \vec{K} = \vec{P} \text{ in } K_j \quad \Big| \quad K$$

Conventions: empty sum denoted by $\mathbf{0}$; Σ sometimes replaced by infix $+$; in practice, one writes a context of (sets of mutually recursive) definitions:

$$K_1 = P_1 \dots K_n = P_n$$

instead of

$$\text{let } \vec{K} = \vec{P} \text{ in } K_j \ .$$

From automata to CCS, ctd.

In CCS

$$\texttt{Act} = L \ \cup \ \overline{L} \ \cup \ \{\tau\}$$

(disjoint union), where L is the set of labels, also called names or channels, and τ is a *silent action* that records a synchronisation.

Conventions:

$$\mu \in \texttt{Act} \qquad \alpha \in L \ \cup \ \overline{L} \qquad \overline{\overline{\alpha}} = \alpha$$

$$\Sigma_{i \in I} a_i.P_i \ = \ (\Sigma_{i \in I \setminus I_0} a_i.P_i) + a_{i_0}.P_{i_0}$$

This notation implicitely views sums as associative and commutative – to be made explicit later.

Synchronisation

Consider the system composed by a customer and by the (slightly revised) vending machine running in parallel:

$$\overline{\mathsf{coin}}.\mathsf{coffe}.\mathbf{0} \;\;\Big|\Big|\;\; \mathsf{coin}.(\overline{\mathsf{coffe}}.\mathbf{0} + \overline{\mathsf{tea}}.\mathbf{0})$$

The two complementary actions $\overline{\mathsf{coin}}$ and coin can synchronise:

$$\overline{\mathsf{coin}}.\mathsf{coffe}.\mathbf{0} \;\;\Big|\Big|\;\; \mathsf{coin}.(\overline{\mathsf{coffe}}.\mathbf{0} + \overline{\mathsf{tea}}.\mathbf{0}) \;\xrightarrow{\tau}\; \mathsf{coffe}.\mathbf{0} \;\;\Big|\Big|\;\; \overline{\mathsf{coffe}}.\mathbf{0} + \overline{\mathsf{tea}}.\mathbf{0}$$

Exercise: draw the automata obtained from the parallel composition $U \;||\; M$.

The parallel operator transforms two automata into a quite different one.

Syntax of CCS

$$
\begin{array}{lcll}
P, Q & ::= & \Sigma_{i\in I}\mu_i.P_i & \text{nondeterministic sum} \\
 & | & \text{let } \vec{K} = \vec{P} \text{ in } K_j & \text{definitions of equations} \\
 & | & K & \text{variable} \\
 & | & P \parallel Q & \text{parallel composition of } P \text{ and } Q \\
 & | & (\boldsymbol{\nu} a)P & \text{scope restriction}
\end{array}
$$

Remark: not yet the ultimate syntax.

Synchronization trees: $P \quad ::= \quad \Sigma_{i\in I}\mu_i.P_i$

Finitary CCS: $P \quad ::= \quad \Sigma_{i\in I}\mu_i.P_i \quad \Big| \quad P \parallel Q \quad \Big| \quad (\boldsymbol{\nu} a)P$

Labelled operational semantics

$$\Sigma_{i\in I}\mu_i.P_i \xrightarrow{\mu_i} P_i \qquad\qquad \frac{P \xrightarrow{\mu} P' \qquad \mu \neq a, \overline{a}}{(\boldsymbol{\nu} a)P \xrightarrow{\mu} (\boldsymbol{\nu} a)P'}$$

$$\frac{P_j\,[\,\ldots, K_i \leftarrow \text{let } \vec{K} = \vec{P} \text{ in } K_i, \ldots\,] \xrightarrow{\mu} P'}{\text{let } \vec{K} = \vec{P} \text{ in } K_j \xrightarrow{\mu} P'}$$

$$\frac{P \xrightarrow{\mu} P'}{P \parallel Q \xrightarrow{\mu} P' \parallel Q} \qquad \frac{Q \xrightarrow{\mu} Q'}{P \parallel Q \xrightarrow{\mu} P \parallel Q'} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

Labelled operational semantics, ctd.

Exercise: draw the LTS defined by the process

$$\text{let } K_1 = a.\bar{c}.K_1 \text{ and } K_2 = b.c.K_2 \text{ in } (\boldsymbol{\nu} c)(K_1 \mathbin{\|} K_2)$$

Labelled operational semantics, ctd.

- τ-transitions correspond to internal evolutions
- α-transitions correspond to interactions with the environment

In λ-calculus, one considers only one (internal) reduction: β.

It is possible to formulate internal reduction in CCS without reference to the environment.

Price to pay: work modulo structural equivalence.

Structural equivalence

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

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

$$(\boldsymbol{\nu} a)P \mid\mid Q \equiv (\boldsymbol{\nu} a)(P \mid\mid Q) \text{ if } a \text{ not free in } Q$$

$$\Sigma_{i\in I}\mu_i.P_i \equiv \Sigma_{i\in I}\mu_{f(i)}.P_{f(i)} \text{ for } f \text{ permutation}$$

$$\text{let } \vec{K} = \vec{P} \text{ in } K_j \equiv P_j\ [\ldots, K_i \leftarrow \text{let } \vec{K} = \vec{P} \text{ in } K_i, \ldots\ ]$$

Reduction operational semantics

The reduction relation, denoted $\rightarrowtriangle$, is defined by the rules below.

$$P_1 + a.P \parallel Q_1 + \overline{a}.Q \;\rightarrowtriangle\; P \parallel Q \qquad\qquad P_1 + \tau.P \;\rightarrowtriangle\; P$$

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

Reduction operational semantics, ctd.

Theorem. The relations $\twoheadrightarrow$ and $\xrightarrow{\tau}\equiv$ coincide.

Exercise: Prove it.

Hint: the claims below are useful.

- If $P \xrightarrow{\mu} P'$ and $P \equiv Q$, then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \equiv Q'$.
- If $P \xrightarrow{\alpha} P'$, then $P \equiv (\boldsymbol{\nu}\vec{a})(\alpha.Q + P_1 \mid\mid P_2)$ and $P' \equiv (\boldsymbol{\nu}\vec{a})(P_1 \mid\mid P_2)$, for some $\vec{a}, P_1, P_2, Q$ with $\alpha \notin \vec{a}$.

Semaphores in CCS

It is easy to *encode* a binary semaphore in CCS:

$$\mathtt{Sem} = P.V.\mathtt{Sem}$$

Example:

$$
\begin{aligned}
&\mathtt{Sem} \;||\; \overline{P}.C_0;\overline{V} \;||\; \overline{P}.C_1;\overline{V} \\
&\quad \rightharpoonup \; V.\mathtt{Sem} \;||\; \overline{P}.C_0;\overline{V} \;||\; C_1;\overline{V} \\
&\quad \rightharpoonup^* \; V.\mathtt{Sem} \;||\; \overline{P}.C_0;\overline{V} \;||\; \overline{V} \\
&\quad \rightharpoonup \; \mathtt{Sem} \;||\; \overline{P}.C_0;\overline{V} \;||\; \mathbf{0}
\end{aligned}
$$

Exercise: Encode $P;Q$ in CCS.

Scope and recursion

Consider (example due to Frank Valencia):

$$P_1 \;=\; \text{let } K = \overline{a} \;\Big|\Big|\; (\boldsymbol{\nu} a)(a.test \;\Big|\Big|\; K)) \text{ in } K \;.$$

Applying the rules, after on unfolding we focus on

$$\overline{a} \;\Big|\Big|\; (\boldsymbol{\nu} a)(a.test \;\Big|\Big|\; \overline{a} \;\Big|\Big|\; (\boldsymbol{\nu} a)(a.test \;\Big|\Big|\; K))$$

and we derive that $P_1 \xrightarrow{\tau} \overline{a} \,||\, (\boldsymbol{\nu} a)(test \,||\, \mathbf{0} \,||\, (\boldsymbol{\nu} a)(a.test \,||\, K))$. Now consider

$$P_2 = \text{let } K = \overline{a} \;\Big|\Big|\; (\boldsymbol{\nu} b)(b.test \;\Big|\Big|\; K)) \text{ in } K$$

The double unfolding yields $\overline{a} \,||\, (\boldsymbol{\nu} b)(b.test \,||\, \overline{a} \,||\, (\boldsymbol{\nu} b)(b.test) \,||\, K)$, which does not reduce as P_1.

Scope and recursion, ctd.

$$\begin{aligned} P_1 &= \text{let } K = \overline{a} \mid\mid (\boldsymbol{\nu} a)(a.test \mid\mid K)) \text{ in } K \\ P_2 &= \text{let } K = \overline{a} \mid\mid (\boldsymbol{\nu} b)(b.test \mid\mid K)) \text{ in } K \end{aligned}$$

There is a tension:

- these two definitions have a different behaviour;
- the identity of bound names should be irrelevant (α-conversion).

Scope and recursion, ctd.

Let us α-convert the bound a in the first definition:

$$P_3 = \mathrm{let}\ K = \overline{a} \mid\mid (\boldsymbol{\nu} b)(b.test \mid\mid K[a \leftarrow b])) \ \mathrm{in}\ K$$

- $K[a \leftarrow b]$ should be a substitution (or explicit relabelling) that is delayed until K is replaced by its actual definition;
- in $(\boldsymbol{\nu} a)P \ = \ (\boldsymbol{\nu} y)P[x \leftarrow y]$ the name y shoud be chosen *not free* for P; but in the example above we cannot chose y until K is unfolded;
- clean solution (definition with parameters): maintain the list of free variables of a constant K: write $K(\vec{x})$ with the restriction that in $\mathrm{let}\ K(\vec{x}) = P\ \mathrm{in}\ Q$ we have $\mathrm{fv}(P) \subseteq \vec{x}$.

Scope and recursion, ctd.

Using this syntax (cf. Milner's book on π-calculus), relabelling can be omitted from syntax, i.e. left implicit, since e.g. $K(a,b)[a \leftarrow c] = K(c,b)$.

Exercise: Express the LTS rule for constants in this new setting.

The dining philosophers: the philosopher

Each philosopher is represented by a process $P_{n,p,a}$, where

- n/h stands for *not hungry*/*hungry*,
- a/p for *absent*/*present* (resp. left and right fork).

$$\begin{array}{lcl}
P_{n,p,a} & = & \tau.P_{h,p,a} + \tau.P_{n,p,a} + \overline{c_l}.P_{n,a,a} \\
P_{n,a,p} & = & \text{symmetric} \\
P_{n,a,a} & = & \tau.P_{n,a,a} + \tau.P_{h,a,a} \\
P_{h,a,a} & = & c_l.P_{h,p,a} + c_r.P_{h,a,p} \\
P_{h,p,a} & = & \overline{c_l}.P_{h,a,a} + c_r.P_{h,p,p} \\
P_{h,a,p} & = & \text{symmetric} \\
P_{h,p,p} & = & eat.P_{n,p,p} \\
P_{n,p,p} & = & \overline{c_l}.P_{n,a,p} + \overline{c_r}.P_{n,p,a}
\end{array}$$

The dining philosophers: linking

$$P_{h,p,a} \;=\; \overline{c_l}.P_{h,a,a} + c_r.P_{h,p,p}$$

The action $\overline{c_l}$

- stands for "give back the left fork",
- should synchronise with the action c_r realised by its *neighbour* process.

We can program this linking using parametrisation $P_{n,p,a}(c_l, c_r)$, and restriction:

$$(\boldsymbol{\nu} f_1)(\boldsymbol{\nu} f_2)(P_{n,p,a}(f_1, f_2) \;||\; P_{n,p,a}(f_2, f_1))$$

Exercise: propose a linking strategy that generalises to $n > 2$ philosophers.

The dining philosophers: properties

Fairness: a hungry philosopher, or a philosopher who has just eaten, is not ignored forever.

Progress: if at least one philosopher is hungry, then eventually one of the hungry philosophers will eat.

We show that, at any stage, Fairness $\Rightarrow$ Progress.

By contradiction: suppose P is the state of the system in which one philospher at least is hungry, and suppose that there is an infinite fair evolution $P \xrightarrow{\tau}^{*}$ that makes no progress. Then:

Step 1: Eventually all philosophers hold at most one fork. No philosopher at any stage can be in state (h, p, p), since by fairness this philosopher will eventually give one of his forks. No philosopher at any stage can be in state (n, p, p) unless it was already in this state in P, since the only way to enter this state is after eating. Hence all the (n, p, p) states will eventually disappear.

Step 2: Eventually, all philosphers hold exactly on fork. This is because if one philospher had no fork, then another one would hold two (n forks for $n - 1$ philosphers).

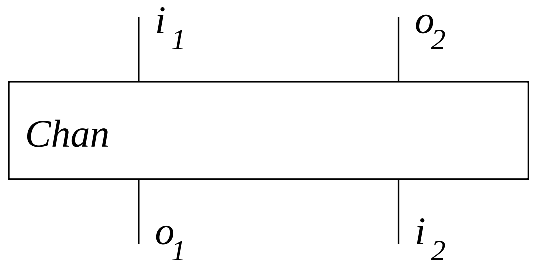
Step 3: When this happens, our philosopher is still hungry (he cannot revert to non-hungry unless he eats), say it is in state (h, p, a), and eventually by Fairness it is his turn. The transition (h, p, p) is forbidden. Hence he gives his fork to the left neighbour. Only a hungry philosopher receives forks, hence the neighbour is in state (h, p, a), but then makes the transition (h, p, p) which is also forbidden.

Buffers

A bidirectional 1-place communication channel:

$$Chan(i_1, i_2, o_1, o_2) = i_1.\overline{o_1}.Chan(i_1, i_2, o_1, o_2) + i_2.\overline{o_2}.Chan(i_1, i_2, o_1, o_2)$$

graphically represented as:



Exercise: by means of only constant application, parallel composition, and restriction, use two copies of $Chan$ to construct a bidirectional 2-place channel.

Exercise: do you really trust your implementation of bidirectional 2-place channel?