A logical view on scheduling in concurrency

Emmanuel Beffara

I2M, CNRS & Université d’Aix-Marseille

Chocola – 3 avril 2014
Introduction
  Proofs as processes
  Processes as untyped proofs
  Why we should search further

Proofs as schedules
  MLL with actions
  Soundness and completeness

Uniform translations
  Asynchronous translation
  Synchronous translation

Discussion
Logic vs computation

- The *formulae as types* approach:
  
  \[
  \begin{align*}
  \text{formula} & \leftrightarrow \text{type} \\
  \text{proof rules} & \leftrightarrow \text{primitive instructions} \\
  \text{proof} & \leftrightarrow \text{program} \\
  \text{normalization} & \leftrightarrow \text{evaluation}
  \end{align*}
  \]

- The *proof search* approach:
  
  \[
  \begin{align*}
  \text{formula} & \leftrightarrow \text{program} \\
  \text{proof rules} & \leftrightarrow \text{operational semantics} \\
  \text{proof construction} & \leftrightarrow \text{execution} \\
  \text{proof} & \leftrightarrow \text{successful run}
  \end{align*}
  \]
The *formulae as types* approach:

- formula ↔ type
- proof rules ↔ primitive instructions
- proof ↔ program
- normalization ↔ evaluation

The *proof search* approach:

- formula ↔ program
- proof rules ↔ operational semantics
- proof construction ↔ execution
- proof ↔ successful run

How can we fit *concurrency* into this framework?

What is a proper *denotational semantics* for concurrency?
Cut elimination in proof nets is an interactive process:
Cut elimination in proof nets is an interactive process:
Cut elimination in proof nets is an interactive process:

It is natural to represent it a language for interactive processes:

\[(νz)(νxy)(\bar{z}\langle xy \rangle | P | Q | z(xy)R)\]
Proofs as processes

- Cut elimination in proof nets is an interactive process:

\[ (\nu z)(\nu x y)(\tilde{z}\langle xy \rangle | P | Q) | z(xy)R \rightarrow (vxy)(P | Q | R) \]

- It is natural to represent it a language for interactive processes:
Proofs as processes

This idea was first implemented in

Gianluigi Bellin and Phil Scott
On the π-calculus and linear logic
Theoretical Computer Science, 1994
Proofs as processes

This idea was first implemented in

Gianluigi Bellin and Phil Scott
On the $\pi$-calculus and linear logic
Theoretical Computer Science, 1994

Good points:

- Adequate representation of proof dynamics
- Study of information flow through proofs
Proofs as processes

This idea was first implemented in

Gianluigi Bellin and Phil Scott
On the π-calculus and linear logic
Theoretical Computer Science, 1994

Good points:
- Adequate representation of proof dynamics
- Study of information flow through proofs

Limitations:
- Requires a lot of coding
- Touches processes of a very restricted form
- Does not provide much insight on the π-calculus
Typing processes in linear logic

Axiom and cut:

\[ u \rightarrow v \vdash u : \downarrow A \perp, v : \uparrow A \]

\[ P \vdash \Gamma, \bar{x} : A \quad Q \vdash \bar{x} : A \perp, \Delta \]

\[ (v\bar{x})(P \mid Q) \vdash \Gamma, \Delta \]

Multiplicatives:

\[ P \vdash \Gamma, \bar{x} : A \quad Q \vdash \bar{y} : B, \Delta \]

\[ P \mid Q \vdash \Gamma, \bar{x}\bar{y} : A \otimes B, \Delta \]

\[ P \vdash \Gamma, \bar{x} : A \quad \bar{y} : B \]

\[ P \vdash \Gamma, \bar{x}\bar{y} : A \bowtie B \]

Actions:

\[ P \vdash \Gamma, \bar{x} : A \]

\[ u(\bar{x}).P \vdash \Gamma, u : \downarrow A \]

\[ \bar{u}(\bar{x}).P \vdash \Gamma, u : \uparrow A \]

Exponentials for replication, additives for external choice.
Typing processes in linear logic

The system on the previous slide was introduced in

EB
A concurrent model for linear logic
MFPS 2006

but was found to be strongly related to

Nobuko Yoshida, Martin Berger, and Kohei Honda
Strong normalisation in the π-calculus
LICS 2001

Bellin and Scott’s encoding decomposes inside.
Independently developed:

Luís Caires and Frank Pfenning
Session types as intuitionistic linear propositions
Concur 2010

appears as a fragment.
Typing processes in linear logic

Good things:

- Typed processes cannot diverge or deadlock.
- Typing is preserved by reduction *up to structural congruence*.
- Extends to differential linear logic *through “algebraic” extensions of process calculi*.
- Induces translations of the λ-calculus into the π-calculus.
Typing processes in linear logic

Good things:

- Typed processes cannot diverge or deadlock.
- Typing is preserved by reduction up to structural congruence.
- Extends to differential linear logic through “algebraic” extensions of process calculi.
- Induces translations of the λ-calculus into the π-calculus.

Shortcomings:

- Typed processes are essentially functional.
- Only top-level cut elimination matches execution.
- Many well-behaved interaction patterns are not typable.

\[ a.\bar{b} \mid b.\bar{c} \mid \bar{a}.c.d \]
Dual approach: implement processes as proofs in a suitable logic.
Processes as untyped proofs

Dual approach: implement processes as proofs in a suitable logic.

- Translating *all* processes requires an untyped proof language.
Processes as untyped proofs

Dual approach: implement processes as proofs in a suitable logic.

- Translating all processes requires an untyped proof language.
- Standard linear logic is not an option because of confluence.
Processes as untyped proofs

Dual approach: implement processes as proofs in a suitable logic.

- Translating *all* processes requires an untyped proof language.
- Standard linear logic is not an option because of confluence.
- Differential linear logic allows for explicit non-determinism:

\[
\frac{P \vdash \Gamma \quad Q \vdash \Gamma}{P + Q \vdash \Gamma}
\]

Its rules allow for an implementation of all processes.

Thomas Ehrhard and Olivier Laurent
Interpreting a finitary π-calculus in differential interaction nets
Concur 2007
Processes as untyped proofs

Good points:

- Does provide insights on concurrent processes
- Relates algebraic proof semantics and process semantics
Processes as untyped proofs

Good points:
- Does provide insights on concurrent processes
- Relates algebraic proof semantics and process semantics

Limitations:
- Not clear how to get logic back into the process language
- Prefixing is only described very indirectly:
  $\pi$-calculus $\rightarrow$ solos calculus $\rightarrow$ differential nets
Proof normalization, aka *cut elimination*:
- the meaning of a proof is in its normal form,
- normalization is an *explicitation* procedure,
- it really wants to be confluent.

Interpretation of concurrent processes:
- the meaning is the *interaction*, the final (irreducible) state is less relevant,
- a given process may behave very differently depending on scheduling decisions.

*Some information is missing.*
Proofs as schedules

The principles of our new interpretation:

- *formula* ↔ *type of interaction*
- *proof rules* ↔ *primitives for building schedules*
- *proof* ↔ *schedule for a program*
- *normalization* ↔ *evaluation according to a schedule*

This is not exactly:

- *Curry-Howard* for processes:
  proofs are not programs, but behaviours of programs

- *Proof search*:
  the dynamics is not in proof construction but in cut-elimination

but a sort of middle ground in between.
The first step: a logical description of all executions.

EB and Virgile Mogbil
Proofs as executions
IFIP TCS 2012 — Chocola 14/3/2013

How we proceed:

- Back to CCS, for now.
- Slightly change the logic to represent actions explicitly.
- Match each execution with cut elimination of some proof.
Multiplicative CCS

We consider a CCS-style process calculus.

\[ P, Q := 1 \quad \text{inaction} \]
\[ a.P \quad \text{perform } a \text{ then do } P \]
\[ P \mid Q \quad \text{interaction of } P \text{ and } Q \]
\[ (\nu a)P \quad \text{scope restriction} \]

There is one source of non-determinism:
the pairing of associated events upon synchronization

\[ a.P \mid a.Q \mid \bar{a}.R \rightarrow \begin{cases} a.P \mid Q \mid R \\ P \mid a.Q \mid R \end{cases} \]
Types of schedules:

\[ A, B := \langle a \rangle A \quad \text{do action } a \text{ and then act as } A \]
\[ A \otimes B \quad \text{two independent parts, one as } A, \text{ the other as } B \]
\[ A \otimes B \quad A \text{ and } B \text{ are both exhibited, but correlated} \]
\[ \alpha \quad \text{an unspecified behaviour (type variable)} \]
\[ \alpha^\perp \quad \text{something that can interact with } \alpha \]
\[ (\forall \alpha A, \exists \alpha A \quad \text{quantification over behaviours}) \]

Transforming schedules:

\[ A_1, \ldots, A_n \vdash B \quad \text{behave as type } B \text{ in association} \]
\[ \text{with processes behaving as each type } A_i \]

Two-sided version.
MLL with actions
The formulas

Types of schedules:

\[ A, B := \langle a \rangle A \]  
\[ A \otimes B \]  
\[ A \bowtie B \]  
\[ \alpha \]  
\[ \alpha^\perp \]  
\[ (\forall \alpha A, \exists \alpha A) \]

Do action \( a \) and then act as \( A \)

Two independent parts, one as \( A \), the other as \( B \)

\( A \) and \( B \) are both exhibited, but correlated

An unspecified behaviour (type variable)

Something that can interact with \( \alpha \)

Quantification over behaviours

Transforming schedules:

\[ \vdash A^\perp_1, \ldots, A^\perp_n, B \]

Behave as type \( B \) in association

With processes behaving as each type \( A_i \)

Duality:

\[ (A \otimes B)^\perp = A^\perp \bowtie B^\perp, \quad (\langle a \rangle A)^\perp = \langle \bar{a} \rangle (A^\perp). \]
Types of schedules:

\[
A, B := \langle a \rangle A \\
A \otimes B \\
A \bowtie B \\
\alpha \\
\alpha^\perp \\
( \forall \alpha A, \exists \alpha A)
\]

- \(A, B := \langle a \rangle A\): do action \(a\) and then act as \(A\)
- \(A \otimes B\): two independent parts, one as \(A\), the other as \(B\)
- \(A \bowtie B\): \(A\) and \(B\) are both exhibited, but correlated
- \(\alpha\): an unspecified behaviour (type variable)
- \(\alpha^\perp\): something that can interact with \(\alpha\)
- \(( \forall \alpha A, \exists \alpha A\): quantification over behaviours

Transforming schedules:

\[
P \vdash A_1^+, \ldots, A_n^+, B
\]

- \(P\) can behave as type \(B\) in association with processes behaving as each type \(A_i\)

Duality:

\[
(A \otimes B)^\perp = A^\perp \bowtie B^\perp, \quad (\langle a \rangle A)^\perp = \langle \overline{a} \rangle (A^\perp).
\]
MLL with actions

Proof rules

Axiom and cut:

\[
\begin{align*}
\frac{}{1 \vdash A^\bot, A} \\
\frac{P \vdash \Gamma, A \quad Q \vdash A^\bot, \Delta}{P \mid Q \vdash \Gamma, \Delta}
\end{align*}
\]

Multiplicatives:

\[
\begin{align*}
\frac{P \vdash \Gamma, A \quad Q \vdash B, \Delta}{P \mid Q \vdash \Gamma, A \otimes B, \Delta} \\
\frac{P \vdash \Gamma, A, B}{P \vdash \Gamma, A \mathbin{\&} B}
\end{align*}
\]

Actions:

\[
\begin{align*}
\frac{P \vdash \Gamma, A}{a.P \vdash \Gamma, \langle a \rangle A} \\
\frac{P \vdash \Gamma, A}{P \vdash \Gamma, \forall \alpha A}
\end{align*}
\]

Quantification:

\[
\begin{align*}
\frac{P \vdash \Gamma, A}{P \vdash \Gamma, \forall \alpha A} \\
\frac{\alpha \notin \text{fv}(\Gamma)}{P \vdash \Gamma, \exists \alpha A}
\end{align*}
\]

Ceci n’est pas un système de types.
MLL with actions

Proof rules

Axiom and cut:

\[
\begin{align*}
1 & \vdash A^\bot, A \\
\frac{P \vdash \Gamma, A \quad Q \vdash A^\bot, \Delta}{P \mid Q \vdash \Gamma, \Delta}
\end{align*}
\]

Multiplicatives:

\[
\begin{align*}
P \vdash \Gamma, A \\
Q \vdash B, \Delta \\
P \mid Q \vdash \Gamma, A \otimes B, \Delta
\end{align*}
\]

\[
\begin{align*}
P \vdash \Gamma, A, B
\end{align*}
\]

Actions:

\[
\begin{align*}
P \vdash \Gamma, A
\end{align*}
\]

\[
\begin{align*}
a. P & \vdash \Gamma, \langle a \rangle A
\end{align*}
\]

Quantification:

\[
\begin{align*}
P \vdash \Gamma, A \\
\quad \alpha \notin \text{fv}(\Gamma) \\
P \vdash \Gamma, A[B/\alpha]
\end{align*}
\]

\[
\begin{align*}
P \vdash \Gamma, \forall \alpha A
\end{align*}
\]

\[
\begin{align*}
P \vdash \Gamma, \exists \alpha A
\end{align*}
\]

Ceci n’est pas un système de types.
MLL with actions

Proof nets

MLLa admits proof nets: those of MLL plus unary links for modalities.

- Modality rules commute with everything, indeed $A \simeq \langle a \rangle A$.
- Correctness criteria: the same as MLL.
- We avoid second-order quantification for simplicity, we stick with parametricity in type variables.
The cyclic example

The following proof is an annotation for $a.\overline{b} \mid b.\overline{c} \mid \overline{a}.c.d$:

If we use boxes, we have a “head cut elimination” matching execution:

$$a.\overline{b} \mid b.\overline{c} \mid \overline{a}.c.d$$
The cyclic example

The following proof is an annotation for $a.\overline{b} \mid b.\overline{c} \mid \overline{a}.c.d$:

If we use boxes, we have a “head cut elimination” matching execution:

$$a.\overline{b} \mid b.\overline{c} \mid \overline{a}.c.d \rightarrow \overline{b} \mid b.\overline{c} \mid c.d$$
The cyclic example

The following proof is an annotation for \( a\overline{b} | b\overline{c} | \overline{a}.c.d \):

\[
\begin{align*}
\alpha & \perp \\
\langle \overline{b} \rangle & \langle \c \rangle & \langle b \rangle \langle b \rangle & \langle c \rangle & \langle d \rangle
\end{align*}
\]

If we use boxes, we have a “head cut elimination” matching execution:

\[
a\overline{b} | b\overline{c} | \overline{a}.c.d \rightarrow \overline{b} | b\overline{c} | c.d
\]
The cyclic example

The following proof is an annotation for $a\bar{b} | b\bar{c} | \bar{a}.c.d$:

If we use boxes, we have a “head cut elimination” matching execution:

$$a\bar{b} | b\bar{c} | \bar{a}.c.d \rightarrow \bar{b} | b\bar{c} | c.d \rightarrow \bar{c} | c.d$$
The cyclic example

The following proof is an annotation for $a.\overline{b} \mid b.\overline{c} \mid \overline{a}.c.d$:

If we use boxes, we have a “head cut elimination” matching execution:

$$a.\overline{b} \mid b.\overline{c} \mid \overline{a}.c.d \rightarrow \overline{b} \mid b.\overline{c} \mid c.d \rightarrow \overline{c} \mid c.d \rightarrow d$$
The results of step 1

**Theorem (Soundness)**

*Typing is preserved by reduction, head cut-elimination steps correspond to execution steps.*

The definition of “head” cut-elimination requires boxes for modality rules, to keep track of prefixing.

**Theorem (Completeness)**

*For every lock-avoiding run \( P_1 \rightarrow \ldots \rightarrow P_n \) there are annotations such that \( \pi_1 : P_1 \vdash \Gamma \rightarrow \ldots \rightarrow \pi_n : P_n \vdash \Gamma \) is a cut elimination sequence.*
Observations

Every execution correspond to some proof:
  - the proof provides a schedule (pairing between actions),
  - cut elimination provides actual execution.

These proofs have very different types:
  - the type is deduced from the execution, it describes control flow according a particular schedule;
  - the type describes a way for a process interacts with its environment,
  - no most general type.

Step 2: make things more uniform.
For annotating a process \( a.P | Q | \bar{a}.R \) in an execution step

\[
a.P | Q | \bar{a}.R \quad \rightarrow \quad P | Q | R
\]

on may need some plumbing:
The trick for actions prefixes

For annotating a process $a.P \mid Q \mid \bar{a}.R$ in an execution step

$$a.P \mid Q \mid \bar{a}.R \rightarrow P \mid Q \mid R$$
on may need some plumbing:
The trick for actions prefixes

For annotating a process $a.P \parallel Q \parallel \bar{a}.R$ in an execution step

$$a.P \parallel Q \parallel \bar{a}.R \rightarrow P \parallel Q \parallel R$$

on may need some plumbing:
The trick for actions prefixes

For annotating a process $a.P \mid Q \mid \bar{a}.R$ in an execution step

$$a.P \mid Q \mid \bar{a}.R \rightarrow P \mid Q \mid R$$

on may need some plumbing:

\[
\begin{array}{c}
\begin{tikzpicture}[scale=0.5,auto]
  \node (P) at (0,0) [circle, fill=red!20] {P};
  \node (Q) at (3,0) [circle, fill=red!20] {Q};
  \node (R) at (6,0) [circle, fill=red!20] {R};
  \node (A) at (0,-1) {$A$};
  \node (B) at (6,-1) {$B$};
  \node (a) at (-1,-2) {$\langle a \rangle$};
  \node (a_bar) at (5,-2) {$\langle \bar{a} \rangle$};
  \node (A_perp) at (0,-3) {$A^\perp$};
  \node (B_perp) at (6,-3) {$B^\perp$};
  \node (A_times_B_perp) at (3,-3) {$A \otimes B^\perp$};
  \path (P) edge (A);
  \path (Q) edge (A_perp);
  \path (R) edge (B_perp);
  \path (Q) edge (B);
  \path (A) edge (a);
  \path (A_perp) edge (a_bar);
  \path (A) edge (A_times_B_perp);
  \path (A_perp) edge (B_perp);
\end{tikzpicture}
\end{array}
\]
The trick for actions prefixes

For annotating a process $a.P | Q | \tilde{a}.R$ in an execution step

$$a.P | Q | \tilde{a}.R \rightarrow P | Q | R$$

on may need some plumbing:

The type of $\tilde{a}.R$ depends on that of $Q$, even if only $Q$ only interacts with $P$. 
The trick for actions prefixes

For annotating a process $a.P | Q | \bar{a}.R$ in an execution step

$$a.P | Q | \bar{a}.R \rightarrow P | Q | R$$

on may need some plumbing:

The construction does not depend on the types: *parametricity in* $\alpha$

one can always proceed the same way.
Type assignment

“Asynchronous” version

Definition

Terms of MCCS are translated into MLLa formulas as follows:

\[
\begin{align*}
[1]_A & := \forall \alpha \, \alpha \perp \otimes \alpha \\
[P | Q]_A & := [P]_A \otimes [Q]_A \\
[a.P]_A & := \forall \alpha \, \langle a \rangle \alpha \perp \otimes ([P]_A \otimes \alpha) = \forall \alpha \, \langle \bar{a} \rangle \alpha \rightarrow ([P]_A \otimes \alpha) \\
[\bar{a}.P]_A & := \forall \beta \, ([P]_A \otimes \beta \perp) \otimes \langle \bar{a} \rangle \beta = \forall \beta \, ([P]_A \rightarrow \beta) \rightarrow \langle a \rangle \beta
\end{align*}
\]

Name hiding is left aside for now.
Proof assignment
“Asynchronous” version

**Fact**

*For every* $P$, the type $\lceil P \rceil_A$ has one cut-free proof $\langle P \rangle_A$.

For actions:

\[
\langle a.P \rangle_A = \langle a \rangle \quad \text{and} \quad \langle \bar{a}.P \rangle_A = \langle \bar{a} \rangle
\]
Soundness and completeness
“Asynchronous” version

**Theorem**

There is an execution $P \rightarrow^* 1$ if and only if $\lceil P \rceil_A \rightarrow [1]_A$ is provable in MLL (without modality rules).
Theorem

There is an execution $P \rightarrow^* 1$ if and only if $[P]_A \dashv [1]_A$ is provable in MLL (without modality rules).

From execution to implication:
- each execution step is provable.

From implication to execution:
- find a first interaction,
  exploiting the correctness criterion for a proof of $[P]_A \dashv [1]_A$. 
Suppose there is some proof of $[a_1.P_1 | \ldots | a_n.P_n]_A \rightarrow [1]_A$ but no two $a_i$ can synchronize:
Suppose there is some proof of $[a_1.P_1 | \ldots | a_n.P_n]_A \xrightarrow{\tau} [1]_A$ but no two $a_i$ can synchronize:
Suppose there is some proof of $[a_1.P_1 | \ldots | a_n.P_n]_A \rightarrow [1]_A$ but no two $a_i$ can synchronize:
Soundness and completeness

“Asynchronous” version: finding the first action

Suppose there is some proof of \([a_1.P_1 | \ldots | a_n.P_n]_A \rightarrow [1]_A\) but no two \(a_i\) can synchronize:

\[
\pi_1 \rightarrow \langle a_1 \rangle \alpha_1
\]
\[
\pi_2 \rightarrow \langle a_2 \rangle \alpha_2
\]
\[
\pi_3 \rightarrow \langle a_3 \rangle \alpha_3
\]
\[
\pi_k \rightarrow \langle a_k \rangle \alpha_k
\]

...
Soundness and completeness

“Asynchronous” version: finding the first action

Suppose there is some proof of $[a_1.P_1 | ... | a_n.P_n]_A \Rightarrow [1]_A$ but no two $a_i$ can synchronize:

Impossible because of acyclicity!
### Type assignment

#### “Synchronous” version

**Definition**

Terms of MCCS are translated into MLLa formulas as follows:

<table>
<thead>
<tr>
<th>Formula</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1]_S ) := ( \forall \alpha , \alpha^\perp \otimes \alpha )</td>
<td>= ( \forall \alpha , \alpha \rightarrow \alpha )</td>
</tr>
<tr>
<td>([P</td>
<td>Q]_S ) := ([P]_S \otimes [Q]_S )</td>
</tr>
<tr>
<td>([a.P]_S ) := ( \forall \alpha , &lt;a&gt;(\alpha^\perp \otimes ([P]_S \otimes \alpha)) )</td>
<td>= ( \forall \alpha , &lt;a&gt; (\alpha \rightarrow ([P]_S \otimes \alpha)) )</td>
</tr>
<tr>
<td>([\bar{a}.P]_S ) := ( \forall \beta , &lt;\bar{a}&gt; ([P]_S \otimes \beta^\perp) \otimes \beta )</td>
<td>= ( \forall \beta , &lt;\bar{a}&gt; ([P]_S \rightarrow \beta) \rightarrow \beta )</td>
</tr>
</tbody>
</table>

*Spot the difference!*
Fact

For every $P$, the type $\lceil P \rceil_S$ has one cut-free proof $(P)_S$.

For actions:

\[
(\langle a.P \rangle)_S = \alpha \perp \alpha
\]

\[
(\langle \bar{a}.P \rangle)_S = \beta \perp \beta
\]
### Theorem

There is an execution $P \rightarrow^* Q$ if and only if $[P]_S \rightarrow [Q]_S$ is provable in MLL (without modality rules).
Theorem

There is an execution $P \rightarrow^{\ast} Q$ if and only if $\lceil P \rceil_S \rightarrow \lceil Q \rceil_S$ is provable in MLL (without modality rules).

From execution to implication:

\[
\begin{align*}
A^\perp & \quad A \\
B^\perp & \quad B
\end{align*}
\]

\[
\begin{align*}
[R]_S^\perp & \quad [R]_S \\
\& & \quad \&
\end{align*}
\]

with

\[
\begin{align*}
A & = \langle a \rangle ([Q]_S^\perp \& ([P]_S \otimes [Q]_S)) \\
B & = [P]_S \otimes [Q]_S
\end{align*}
\]

proves $\lceil (a.P \mid \bar{a}.Q) \mid R \rceil_S \rightarrow \lceil (P \mid Q) \mid R \rceil_S$
Soundness and completeness
“Synchronous” version

**Theorem**

*There is an execution* $P \rightarrow^* Q$ *if and only if* $\llbracket P \rrbracket_S \rightarrow \llbracket Q \rrbracket_S$ *is provable in MLL (without modality rules).*

From execution to implication:
- each execution step is provable.

From implication to execution:
- take a proof of $\llbracket P \rrbracket_S \rightarrow \llbracket Q \rrbracket_S$
- cut it against $\llbracket P \rrbracket_S$, eliminate the cut
- read back process terms from intermediate steps
**Pairings**

**Definition**

A *pairing* is an association between occurrences of dual actions

\[ p_1 : P = a.b.A | \bar{a}.c.B | \bar{b}.\bar{c}.C | a.\bar{c} \]

\[ p_2 : \]

**Definition**

A *determinisation* of \( P \) along a pairing \( p \) is a renaming \( \partial_p(P) \) of actions in \( P \) where names are equal only for related actions.

\[ \partial_{p_1}(P) = a_1.b_1.\partial(A) | \bar{a}_2.c_1.\partial(B) | \bar{b}_2.\bar{c}_2.\partial(C) | a_2.\bar{c}_1 \]

\[ \partial_{p_2}(P) = a_1.b_1.\partial(A) | \bar{a}_1.c_1.\partial(B) | \bar{b}_1.\bar{c}_1.\partial(C) | a_2.\bar{c}_2 \]
Facts about pairings:

- each run induces a pairing
- runs are equivalent up to permutation of independent events iff they induce the same pairing
- if $p$ is a consistent pairing of $P$ then $p$ is the unique maximal consistent pairing of $\partial_p(P)$

Hence pairings are *execution schedules* and determinized terms represent them inside the process language.

**Observation**

Pairings are related to placements of axiom links in proofs of $[P]_A \to [1]_A$. 
Some points deserve more investigation:

**Replication:** everything extends smoothly by setting $[!P]_A = ![P]_A$.

**Choice:** additives are the natural option

**Name hiding:** the situation is not obvious
  - use quantifiers?
    - existential? nabla?
  - partial scheduling?
    - $(va)P$ is $P$ with some proof that decides what happens on $a$

**Name passing:** need to fix hiding first!
Current state of affairs:

- A logical description of scheduling in processes
- Explicitation of *control flow* through processes
- Hints for a new study of prefixing in processes
Further directions

Current state of affairs:

- A logical description of scheduling in processes
- Explicitation of control flow through processes
- Hints for a new study of prefixing in processes

Ongoing questions:

- Which semantics for the logic of schedules?  
  coherence spaces for MLLa, etc
- CPS-like interpretation of processes?  
  the translation of actions is a kind of double negation
- A logical account on π-to-solos encoding?  
  by relating to other systems
Work in progress...