Describing processes and their executions in proof theory

Emmanuel Beffara

I2M, Université d'Aix-Marseille

Journées Géocal-LAC, 28-29 novembre 2016

Mobile processes and logic

- Process algebras
- Proofs as programs
- Typing processes

Logical description of processes

- A logic for process interaction
- Unifying type systems

Logical description of executions

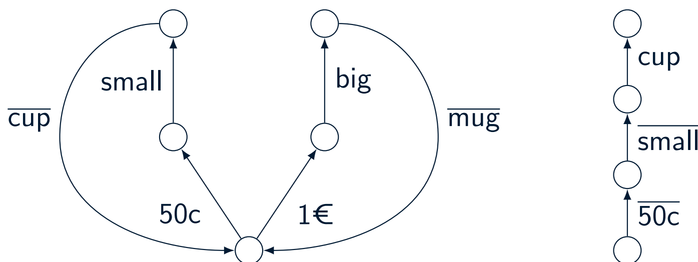
- Proofs as schedules
- Some semantics

1

Mobile processes and logic

Let us start with a coffee

The classical coffee machine example:



the machine: $M = \text{50c} \,.\, \text{small} \,.\, \overline{\text{cup}} \,.\, M + \text{1€} \,.\, \text{big} \,.\, \overline{\text{mug}} \,.\, M$

the client: $C = \overline{\text{50c}} \,.\, \overline{\text{small}} \,.\, \text{cup}$

Process algebra: CCS

We use a formal language to represent programs that interact by message passing:

$$
\begin{array}{ll}
P, Q := a.P,\ \ \bar{a}.P & \text{action prefix (receive, send)} \\
\qquad P \mid Q,\ \ P + Q,\ \ 0 & \text{parallel composition, choice, stop} \\
\qquad *P & \text{replicated process} \\
\qquad (\nu a)P & \text{name restriction}
\end{array}
$$

One can define the dynamics as a transition system:

$$
\frac{}{a.P \xrightarrow{a} P} \qquad \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \frac{P \xrightarrow{\ell} P'}{P + Q \xrightarrow{\ell} P'} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}
$$

or as a *reduction* up to structural congruence:

$$
(a.P + P') \mid (\bar{a}.Q + Q') \to P \mid Q
$$

Name passing: the π-calculus

Take the variant with value passing:

$$a(x).P \mid \bar{a}\langle v\rangle.Q \to P[v/x] \mid Q$$

and say that x and v are channel names:

$$a(x).\bar{x}\langle c\rangle \mid \bar{a}\langle b\rangle \mid b(y).P \to \bar{b}\langle c\rangle \mid b(y).P \to P[c/y]$$

then you get the π-calculus:

$$\begin{array}{lll} P, Q := & a(x).P,\ \ \bar{a}\langle v\rangle.P & \text{action prefix} \\ & P \mid Q,\ \ P + Q,\ \ 0 & \text{composition, choice, stop} \\ & *P & \text{replicated process} \\ & (\nu a)P & \text{name restriction} \end{array}$$

It is sometimes called *mobile processes* because the communication topology evolves during execution.

Proofs as programs

The *formulae as types* approach:

$$\begin{array}{rcl} \text{formula} & \leftrightarrow & \text{type} \\ \text{proof rules} & \leftrightarrow & \text{primitive instructions} \\ \text{proof} & \leftrightarrow & \text{program} \\ \text{normalization} & \leftrightarrow & \text{evaluation} \end{array}$$

Works perfectly for λ-calculus and intuitionistic logic:

- *proof semantics* for LJ, AF2, Type theory...
- *program semantics* for PCF and extensions...
- *type systems* for programming languages...
- *operational insights* through game semantics, linear logic...

Proofs as programs – typed λ-calculi

- Computation: $(\lambda x.M)N \to_\beta M[N/x]$
- Typing: $\dfrac{}{\Gamma, x : A \vdash x : A}$ $\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$ $\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M)N : B}$

Theorem

Reduction is confluent, typed terms are strongly normalising.

Many variations around this:

- extensions that preserve good properties (quantification, dependent types, etc.)
- restrictions (linearity, implicit complexity)
- inconsistent extensions (arbitrary fixed points, $D = D \to D$)

Proofs as programs – interaction and concurrency?

Intuitive adaptation of *formulae as types* for the concurrent world:

formula	$\leftrightarrow$	type	*but what is a type?*
proof rules	$\leftrightarrow$	primitive instructions	
proof	$\leftrightarrow$	process	
normalization	$\leftrightarrow$	execution	

The logic should feature

- linearity : agents duplication is explicit
- symmetry : processes are not functions (rather relations)

It should be able to account for non-determinism.

Typing processes

One assigns *types* to different elements of language in order to put constraints on the set of allowed expressions and thus statically ensure properties of terms:

- compatibility between senders and receivers,
- control on name usage,
- guarantees on global good behaviour,
- respect of communication protocols. . .

Simple types

The simples type system for the π-calculus:

types of names	$T := B$	base type
	$\sharp(T_1 \dots T_n)$	transports tuples
type of processes	$A := *$	well-formed

Judgements have the shape

$$x_1 : T_1, \dots, x_n : T_n \vdash P : *$$

with proof rules like

$$\frac{\Gamma \vdash u : \sharp T \qquad \Gamma, x : T \vdash P : *}{\Gamma \vdash u(x).P : *} \qquad \frac{\Gamma \vdash u : \sharp T \qquad \Gamma \vdash v : T \qquad \Gamma \vdash P : *}{\Gamma \vdash \bar{u}\langle v \rangle.P : *}$$

and everything is fine.

Input and output capabilities – subtyping

One extends the system with polarity information:

$$
\begin{array}{lll}
T := B & & \text{base type} \\
& i(T_1 \dots T_n) & \text{used for input only} \\
& o(T_1 \dots T_n) & \text{used for output only} \\
& \sharp(T_1 \dots T_n) & \text{no polarity constraint}
\end{array}
$$

If $\Gamma, u : iT \vdash u(x).P : *$ then P uses x according to type T.
If $\Gamma, u : oT \vdash \bar{u}\langle v \rangle.P : *$ then P sends capability T on v.

This naturally induces a notion of *subtyping*:

- $\sharp\vec{T} \leq i\vec{T}$ and $\sharp\vec{T} \leq o\vec{T}$
  if P respects $x : i\vec{T}$ then it respects $x : \sharp\vec{T}$
- if $T \leq U$ then $iT \leq iU$ and $oU \leq oT$
  subtyping on communicated values, sending is contravariant

Termination by stratification

How can we make sure that a process cannot run infinitely?

types for names	$T := B$	base type
	$\sharp_k(T_1 \dots T_n)$	channel at level k
types for processes	$A := *_k$	makes calls at level $\leq k$

On imposes that a replicated process only makes calls to replicated processes of lower levels:

$$\frac{\Gamma \vdash u : \sharp_k T \quad \Gamma \vdash v : T \quad \Gamma \vdash P : *_\ell \quad k \leq \ell}{\Gamma \vdash \bar{u}\langle v\rangle.P : *_\ell}$$

$$\frac{\Gamma \vdash u : \sharp_k T \quad \Gamma, x : T \vdash P : *_\ell \quad \ell < k}{\Gamma \vdash !u(x).P : *_\ell}$$

Yuxin Deng, Davide Sangiorgi
Ensuring termination by typability
IFIP TCS 2004

Session types

One defines protocols for what happens on each channel:

$T, U := B,\ \overline{B}$	production or consumption of a value
$\downarrow T,\ \uparrow T$	receiving, sending a value of type T
$T; U$	sequential composition
$T \mathbin{\&} U,\ \ T \oplus U$	offering a choice, choosing

One can define rules to type some π-calculus with these session types on channels.

Kohei Honda
Types for dyadic interaction
Concur 1993

Kohei Honda, Vasco T. Vasconcelos, Makoto Kubo
Language primitives and type discipline for structured communication-based programming
ESOP 1998

Other type systems

There are systems in the literature to achieve various other things:

- ensure that processes are deadlock-free and reactive

  Naoki Kobayashi
  A new type system for deadlock-free processes
  Concur 2006

- generalise session types to more than two agents

  Kohei Honda, Nobuko Yoshida, Marco Carbone
  Multiparty asynchronous session types
  POPL 2008

- characterise functional or imperative behaviours

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

etc.

Proofs as programs – Linear logic

Proofs in linear logic are strongly reminiscent of interacting processes:

- a cut is a connection between two parts of a system,
- cut elimination makes dual actions interact,
- quotienting proofs up to permutations of rules reveals parallelism in interaction and simplifies syntax, like structural congruence

It is tempting to interpret proofs as processes and transpose known tools from the λ-calculus into the concurrent world.

Samson Abramsky
Computational interpretations of linear logic
TCS 1993

Gianluigi Bellin and Philip J. Scott
On the π-calculus and linear logic
TCS, 1994

EB
A concurrent model for linear logic
MFPS 2006

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

Linear logic and then π-calculus – the problems

Shortcomings:

- Typed processes are essentially functional.
- Many well-behaved interaction patterns are not typable.

$$a.\bar{b} \mid b.\bar{c} \mid \bar{a}.c.d$$

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 state is less relevant,
- a process may behave differently depending on scheduling.

Some information is missing.

2

Logical description of processes

The approach

We want to preserve what works in proof-process correspondences while respecting the intrinsic structure of each language:

- one logically sound typing rule for each syntactic construct
- type isomorphism as a structural congruence for types

Then we study how we can do variations on the basic structure

- identify fragments that match existing systems
- identify (possibly inconsistent) logical principles that yield more permissive systems

Linear logic with named quantifiers

Types of schedules:

$A, B := \exists_a \vec{x}.A,\ \forall_a \vec{x}.A$	send/receive some tuple on *a* then act as *A*
$A \oplus B,\ A \mathbin{\&} B$	choice between *A* and *B* (internal, external)
$A \otimes B,\ A \parr B$	both *A* and *B* (independent, correlated)
$!A,\quad ?A$	repeated behaviour
$\mathbf{1},\quad \bot$	empty behaviour (neutrals of $\otimes$ and $\parr$)

Negation $\sim A$ exchanges the left and right columns.

Interpretation of a typing judgement:

$\vdash P : A_1, \ldots, A_n$ — P *can* exhibit the behaviours A_i, they may be correlated

Typing rules by example – I

- Reception exactly corresponds to universal quantification:

$$\frac{\vdash P : \Gamma, A \qquad x \text{ does not occur in } \Gamma}{\vdash u(x).P : \Gamma, \forall_u x.A}$$

- Parallel composition is introduced as a conjunction:

$$\frac{\vdash P : \Gamma, A \qquad \vdash Q : \Delta, B}{\vdash P \mid Q : \Gamma, \Delta, A \otimes B} \qquad \frac{\vdash P : \Gamma, A, B}{\vdash P : \Gamma, A \parr B}$$

Typing rules by example – I

- Reception exactly corresponds to universal quantification:

$$\frac{\vdash P : \Gamma, A \qquad x \text{ does not occur in } \Gamma}{\vdash u(x).P : \Gamma, \forall_u x.A}$$

- Parallel composition is introduced as a conjunction:

$$\frac{\vdash P : \Gamma, A \qquad \vdash Q : \Delta, B}{\vdash P \mid Q : \Gamma, \Delta, A \otimes B} \qquad \frac{\vdash P : \Gamma, A, B}{\vdash P : \Gamma, A \invamp B}$$

- Emission combines existential quantification and conjunction:

$$\frac{\vdash P : \Gamma, B}{\vdash \bar{u}\langle v \rangle.P : \Gamma, \exists_u x.{\sim}A(x), A(v) \otimes B}$$

"On u, I send some name x and expect a behaviour $A(x)$ from the receiver; besides, I expose $A(v)$ in parallel with B."

Typing rules by example – II

- A name can be made private if its behaviour is correct:

$$\frac{\vdash P : \Gamma, U \qquad u \notin \Gamma}{\vdash (\nu u)P : \Gamma} \quad \text{if} \quad U = \begin{cases} \exists_u x.{\sim}A \otimes \forall_u x.A & \text{linear usage} \\ ?\exists_u x.{\sim}A \otimes !\forall_u x.A & \text{client/server} \\ \ldots \text{ other possibilities } \ldots \end{cases}$$

Typing rules by example – II

- A name can be made private if its behaviour is correct:

$$\frac{\vdash P : \Gamma, U \qquad u \notin \Gamma}{\vdash (\nu u) P : \Gamma} \quad \text{if} \quad U = \begin{cases} \exists_u x. \sim A \otimes \forall_u x. A & \text{linear usage} \\ ?\exists_u x. \sim A \otimes !\forall_u x. A & \text{client/server} \\ \ldots \text{ other possibilities } \ldots \end{cases}$$

- Subtyping allows reasoning on behaviours:

$$\frac{\vdash P : \Gamma, A \qquad A \leq B}{\vdash P : \Gamma, B} \ \text{SUB}$$

where behavioural subtyping $A \leq B$ is generated by rules like

$$\begin{array}{ll} A \otimes (B \parr C) \leq (A \otimes B) \parr C & \text{semi-distributivity} \\ A \otimes \mathbf{1} \leq A & \text{neutrality of } \mathbf{1} \\ ?A \parr ?A \leq ?A & \text{contraction} \end{array}$$

… and possibly other logically sound implications …

Good behaviour of typed processes

Theorem

For all typed term $\vdash P : \Gamma$ and reduction $P \to Q$ on a private name, $\vdash Q : \Gamma$ holds.

Theorem

If a typing $\vdash P : \Gamma$ is derivable, then

- *P has no infinite sequence of internal reductions,*
- *if Γ contains an action, then P has either an internal transition or an observable transition described by Γ.*

Logical fragments

If each name must occur at most once in the type, we get

EB
A concurrent model for linear logic
MFPS 2006

If, moreover, we impose intuitionism and restrict to the sub-language

$$A, B := \uparrow\bot,\ \uparrow(A \parr B),\ \uparrow ?A,\ \downarrow\bot,\ \downarrow(A \parr B),\ \downarrow ?A \quad :$$

H. DeYoung, L. Caires, F. Pfenning, and B. Toninho
Cut reduction in linear logic as asynchronous session-typed communication
CSL 2012

Up to a simple encoding for sessions, we get

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

Logical fragments – some remarks

The restrictions to the language in these restrictions impose that name restrictions (νx) are associated to parallel composition, which makes processes much more regular:

- subtyping is no longer necessary,
- structural congruence does not preserve typability,
- preservation of types by execution only holds up to structural congruence.

Further restriction of these fragments yield standard translations of λ-calculi.

Beyond logical consistency

If we extend the subtyping logic with rules to identify dual connectives ($A \otimes B \simeq A \parr B$, $1 \simeq \bot$, $!A \simeq ?A$) we get (essentially):

Naoki Kobayashi, Benjamin C. Pierce and David N. Turner
Linearity and the pi-calculus
ACM Transactions on Programming Languages and Systems, 1999.

The "logic" has some good properties but does not eliminate cuts:

- soundness with respect to execution is preserved
- termination and lock-freeness are lost

(Apply the same approach with light logic and get termination by stratification?)

3

Logical description of executions

Soundness

If subtyping is extended to allow (logically sound) rules like

$$A \otimes \sim A \leq \bot$$

then things change radically.

- Typing is not preserved by arbitrary execution, BUT
- each typing induces an way of executing the term,
- each correct execution of a term can by transformed into a typing.

In other words:

- replace "must"-type properties with "may"-type properties,
- proofs are not processes but *schedules* of processes.

Proofs as schedules

The principles of our interpretation:

formula	$\leftrightarrow$	type of interaction
proof rules	$\leftrightarrow$	primitives for building schedules
proof	$\leftrightarrow$	schedule for a program
normalization	$\leftrightarrow$	evaluation according to a schedule

This is not exactly:

- *Curry-Howard* for processes:
  proofs are not programs, but behaviours of programs
- *Proof search*:
  there is significant dynamics in cut-elimination

but a sort of middle ground in between.

EB and Virgile Mogbil
Proofs as executions
IFIP TCS 2012

Proofs-as-schedules theorems

Normal form: hiding is used only for bound output (e.g. $(\boldsymbol{\nu} x)\bar{u}\langle x\rangle$).

Theorem (soundness)

For each typing derivation of $\vdash P : \Gamma$ *there is an execution* $P \to^* P'$ *with* $\vdash P' : \Gamma$ *and* P' *is in normal form.*

Idea of the proof: perform cut elimination on the proof of $\vdash P : \Gamma$, read each elimination as a congruence or a reduction step of P.

Proofs-as-schedules theorems

Normal form: hiding is used only for bound output (e.g. $(\boldsymbol{\nu} x)\bar{u}\langle x\rangle$).

Theorem (soundness)

For each typing derivation of $\vdash P : \Gamma$ there is an execution $P \rightarrow^* P'$ with $\vdash P' : \Gamma$ and P' is in normal form.

Idea of the proof: perform cut elimination on the proof of $\vdash P : \Gamma$, read each elimination as a congruence or a reduction step of P.

Theorem (completeness)

For all execution $P \rightarrow^* Q$ where each name is used linearly, if Q is in normal form then P is typeable.

Idea of the proof: show the each normal form is typeable and that each reduction step on linear names preserves typeability backwards.

Proofs-as-schedules and semantics

Work in progress

The correspondence extends to models

- process $\leftrightarrow$ type $\leftrightarrow$ space (think event structure)
- schedule $\leftrightarrow$ proof $\leftrightarrow$ morphism (think configuration)
- execution $\leftrightarrow$ proof normalisation $\leftrightarrow$ (linear ordering)

The models we get are naturally non-interleaving.

4

Conclusion

Summing up

A larger notion of typing with new interpretations:

- significantly non-functional behaviour fits in
- many type systems fit as specialisations
  - constraining the language of type
  - adding logical principles
- new insights in semantics of processes
  - relates proof semantics and models of concurrency
  - hints at new approaches to compositionality in operational semantics
- work in progress ...