A linear π-Calculus in Coq (Work In Progress)

Cinzia Di Giusto
Enrico Tassi

The Prose (PROvers for SEssions) Project

- Micro French 1 year funding outcome of an OPCT coffee break
- 5 participants (C. Di Giusto, M. Giunti, K. Peters, A. Ravara, E. Tassi)





- Goal:

Kicking off a network of collaborations on mechanized proofs for behavioural types

The big plan

- Pick a language: the linear π-calculus
- And its properties: a well typed process has no linear violations
- Choose a tool: Coq
- Identify main problems: how to represent binders

The linear π-calculus

$$
\begin{array}{llll}
P ::= \mathsf{nil} & \text{(nil)} & \mid \; P \parallel Q & \text{(composition)} \\
\quad \mid u?x.P & \text{(input)} & \mid \; *u?x.P & \text{(replicated input)} \\
\quad \mid u!v.P & \text{(output)} & \mid \; (\nu a : T)(P) & \text{(restriction)}
\end{array}
$$

The type system

$$m \in \textsc{Mul} ::= \omega \quad \text{(unrestricted)} \quad \mid \iota \quad \text{(linear)}$$

$$p, q \in \textsc{Pol} ::= \updownarrow \quad \text{(input \& output)} \quad \mid \emptyset \quad \text{(empty)}$$

$$\mid \downarrow \quad \text{(input)} \quad \mid \uparrow \quad \text{(output)}$$

$$T, S \in \textsc{Typ} ::= \top \quad \text{(top)} \quad \mid p[T]^m \quad \text{(channel)}$$

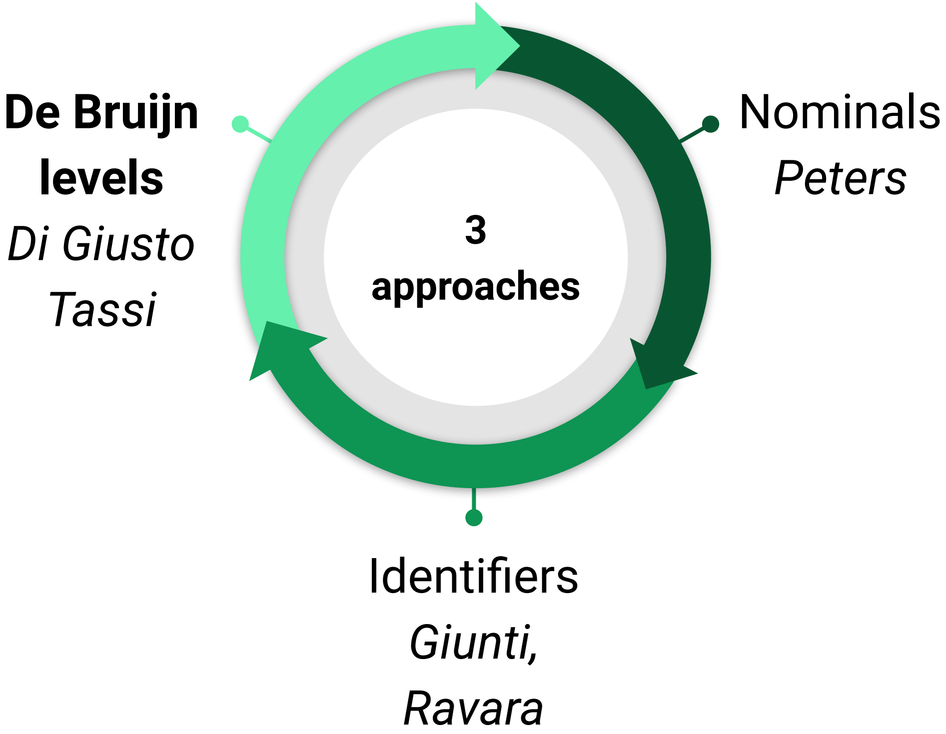
$$\frac{\Gamma, u{:}p[T]^m - \uparrow[T]^m, v{:}T' - T \vdash P \qquad \uparrow \in p}{\Gamma, u{:}p[T]^m, v{:}(\mathrm{v}, T') \vdash u!v.P}$$

Properties

- A closed process P has a linearity violations if P contains two subprocess prefixed with the same action

A well typed process has not linearity violations

Binders



- 3 syntaxes
- 3 semantics
- 3 type systems
- Prove their equivalences

Personal Background / Objectives / Plan

- Experience in devel. Coq & devel. of libraries for Mathematics
- Library/Methodology to ease the adoption of Coq to formalize process algebras and their types
- Gather experience (doing it yourself is the less efficient but most certain way of understanding something) then improve Coq/libs/tools

SSReflect & Mathematical Components

- Used to formalize mathematics, mostly
- Developed and maintained over more than a decade
- No “magic”, a lot of discipline in writing Coq code and iterated improvements
- I want to see if/how all that can be applied in this context

Binders: De Bruijn Levels (not indexes)

- Good implementation choice for binder mobility, used in a project of mine that I want to eventually verify (Elpi)
- Not really “locally nameless”, no number->name change when moving under a binder
- Example

$$\begin{array}{rl} \text{Indexes:} & \lambda x.(\lambda y.\lambda z.f\ x_2\ y_1\ z_0)\ x_0 \to_\beta \lambda x.\lambda z.f\ x_1\ x_1\ z_0 \\ \text{Levels:} & \lambda x.(\lambda y.\lambda z.f\ x_0\ y_1\ z_2)\ x_0 \to_\beta \lambda x.\lambda z.f\ x_0\ x_0\ z_1 \end{array}$$

Inspiration (1/2)

- Autosubst 2, Well Scoped terms (Intrinsically Scoped)
- `term : nat -> Type`
- `| Var (v : fin n) : term n`
- `t : term 0` is a term with no free variables

Inspiration (2/2)

- HOAS & Abella
- Arity = Type index

```
proc = proc 0

name -> proc = proc 1
```

```
sig finite-pic.

% Three syntactic types are used: name (for names), action (for
% actions), and proc (for processes).  The type o denotes the type of
% propositions.

% The constructors for proc are 'null', 'taup', 'match', 'plus',
% 'par', and 'nu' denote, respectively, the mull process, the tau
% prefix, match prefix, the non-deterministic choice operator, the
% parallel composition, and the restriction operator of the
% pi-calculus. The input and output prefixes are encoded as in and
% out.

kind name, proc   type.

type null         proc.
type taup         proc -> proc.
type plus, par    proc -> proc -> proc.
type match, out   name -> name -> proc -> proc.
type in           name -> (name -> proc) -> proc.
type nu           (name -> proc) -> proc.

kind action       type.
type tau          action.
type up, dn       name -> name -> action.

% One step transition for free transitions
type one          proc ->         action  ->          proc  -> o.
% One step transition for binding transitions
type oneb         proc -> (name -> action) -> (name -> proc) -> o.
```

Tools from Mathematical Components

- $`\mathrm{I}_n := \Sigma_{(x:nat)} x < n$ for variables (proof irrelevance)
- $top : `\mathrm{I}_{?n.+1}$ for some ?n to be inferred from the context
- $val\ n : `\mathrm{I}_n$ >-> nat inserted automatically
- $\wedge : nat \to `\mathrm{I}_{?n.+1}$ to "fix" typing
- inordK n x : x < n -> val n (^ x) = x
- $\{ffun\ `\mathrm{I}_n \to \ldots\}$ type environment (object language)

Well scoped processes

```
Inductive process (fv : nat) :=
 | Nu (l : type) (p : process fv.+1)
 | Input (chan : 'I_fv) (p : process fv.+1)
 | RecInput (chan : 'I_fv) (p : process fv.+1)
 | Output (chan : 'I_fv) (value : 'I_fv) (p : process fv)
 | Parallel (p1 p2 : process fv)
 | Zero.

Definition process_ind  (P : forall fv : nat, process fv -> Type) :
  … -> forall (fv : nat) (p : process fv), P fv p
```

Index or non-uniform parameter?

Semantics

```
Inductive closed_step fv : process fv -> label fv -> process fv -> Type :=
  | Recv c v p :

     (* ------------------------------------ *)
     fv |- c `?? p --- Inp c v ---> { top := v }p

 | CloL l (c : 'I_fv) (p q : process fv) (v : 'I_fv.+1) (p1 q1 : process fv.+1) :

     fv ..|- p --- Inp ^c ^v ---> p1                       ->
     fv ..|- q --- Pas ^c ^v l ---> q1                     ->
     (* --------------------------------------------------- *)
     fv |- p `|| q --- Cha c ---> `nu l ({ ^v := top }(^+ p1 `|| q1)
...
with open_step fv : process fv -> label fv.+1 -> process fv.+1 -> Type :=
 | Open l (c : 'I_fv) (p p1 : process fv.+1) :                        (* c < fv, hence != top *)

     fv.+1 |- p --- Out ^c top ---> p1                        ->
     (* ------------------------------------------------------ *)
     fv ..|- `nu l p --- Pas ^c top l ---> p1

 | Recv_open : …

where "fv |- p --- a ---> q" := (@closed_step fv p a q)
and "fv ..|- p --- a ---> q" := (@open_step fv p a q).
```

Typing

```
Inductive typechecks fv : environment fv -> process fv -> Type :=

  | TyOutput (e : environment fv) u v pu tu mu tv tu' tv' p :

     e u = Chan mu pu tu                               # eu
     e v = tv                                          # ev
     {pu = Up} + {pu = UpDown}                         # cap_u
     type_remove (Chan mu pu tu) (Chan mu Up tu) tu'   # trm_u
     type_remove tv tu tv'                             # trm_v
     typechecks (update v tv' (update u tu' e)) p      # typ__IHp
     (*-----------------------------------------------*)
     typechecks e (u `!! v , p)
...
```

Induction steps and names introduction

- The `=> [^ block ]` intro pattern

```
elim: ty_p => [^ p].
```

- Some operations have “hard” syntactic type-requirements

```
Fixpoint subst {fv} ... (p : process fv.+1) : process fv := ...

elim/proc2: p => [^ p] in v c *.
```

Inversion steps

```
Inductive step_spec fv (p0 : process fv ) (l0 : label fv) (q0 : process fv) :
                                  process fv ->      label fv ->     process fv -> Type :=
 | RecvSpec c v p :
     unit                         # RecvSpec
     p0 ::= c `?? p               # def_p
     l0 ::= Inp c v               # def_l
     q0 ::= { top := v } p        # def_q
    (* ------------------------------------------------------------ *)
     step_spec p0 l0 q0 (c `?? p) (Inp c v) ({ top := v } p)
...
Lemma inv_stepP fv p a q :   fv |- p --- a ---> q   ->   step_spec p l q p l q.
Proof. prove_inversion. Qed.

... case/inv_stepP: p0_step_q0 => [^ pq0_ ] // ...
... subst_inv in Hyp1 .. Hypn ...
```

So far, no big proof finished, hence no strong opinions, but...

- DB indexes are not super easy in definitions
  - scoping is easy (thanks to well scoped terms)
  - (re)capturing is not (explicit lifting in a definition...)
- Well Scoped terms are not super easy either
  - setting up induction requires some experience
  - duplication in semantics (can we be more elegant?)
- Inversion lemmas are too boring to write
  - good chance to be automatically generated (almost there)
- Disciplined management of context
  - almost easy, a few refinements of =>[^ block] in the pipes
  - easy-to-repair proofs

Thanks for listening!

Questions?