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 $\pi$-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 $\pi$-calculus

\[ P ::= \text{nil} \quad (\text{nil}) \quad | \quad P \ || \ Q \quad (\text{composition}) \]

\[ | \ u?x.P \quad (\text{input}) \quad | \quad *u?x.P \quad (\text{replicated input}) \]

\[ | \ u!v.P \quad (\text{output}) \quad | \quad (\nu a : T)(P) \quad (\text{restriction}) \]
The type system

\[ m \in \text{Mul} ::= \omega \quad \text{(unrestricted)} \quad | \quad \nu \quad \text{(linear)} \]

\[ p, q \in \text{Pol} ::= \uparrow \quad \text{(input & output)} \quad | \quad \emptyset \quad \text{(empty)} \]

\[ \downarrow \quad \text{(input)} \quad | \quad \uparrow \quad \text{(output)} \]

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

\[ \Gamma, u : p[T]^m - \uparrow [T]^m, v : T' - T \vdash P \quad \uparrow \in p \]

\[ \Gamma, u : p[T]^m, v : (v, T') - u! v. P \]
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

- De Bruijn levels
- Nominals
- Identifiers

- 3 approaches
- 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

  Indexes: \( \lambda x. (\lambda y. \lambda z. f \ x_2 \ y_1 \ z_0) \ x_0 \to_\beta \lambda x. \lambda z. f \ x_1 \ x_1 \ z_0 \)

  Levels: \( \lambda x. (\lambda y. \lambda z. f \ x_0 \ y_1 \ z_2) \ x_0 \to_\beta \lambda x. \lambda z. f \ x_0 \ x_0 \ z_1 \)
Inspiration (1/2)

- Autosubst 2, Well Scoped terms (Intrinsically Scoped)
- \( \text{term} : \text{nat} \rightarrow \text{Type} \)
- \( \mid \text{Var} \ (v : \text{fin} \ n) : \text{term} \ n \)
- \( t : \text{term} \ 0 \) is a term with no free variables
Inspiration (2/2)

- HOAS & Abella
- Arity = Type index

\[
\text{proc} = \text{proc} \ 0
\]

\[
\text{name} \rightarrow \text{proc} = \text{proc} \ 1
\]
Tools from Mathematical Components

- ‘I_n := Σ(x:nat) x < n for variables (proof irrelevance)
- top : ‘I_{?n.+1} for some ?n to be inferred from the context
- val n : ‘I_n --> nat inserted automatically
- ^ : nat --> ‘I_{?n.+1} to “fix” typing
- inordK n x : x < n -> val n (^ x) = x
- {ffun ‘I_n --> …} type environment (object language)
Well scoped processes

\textbf{Inductive} \textit{process} (fv : nat) := \\
\hspace{1cm} | \text{Nu} (l : \text{type}) (p : \text{process} \ fv. +1) \\
\hspace{1cm} | \text{Input} (\text{chan} : 'I_{fv}) (p : \text{process} \ fv. +1) \\
\hspace{1cm} | \text{RecInput} (\text{chan} : 'I_{fv}) (p : \text{process} \ fv. +1) \\
\hspace{1cm} | \text{Output} (\text{chan} : 'I_{fv}) (\text{value} : 'I_{fv}) (p : \text{process} \ fv) \\
\hspace{1cm} | \text{Parallel} (p1 \ p2 : \text{process} \ fv) \\
\hspace{1cm} | \text{Zero}.

\textbf{Definition} \textit{process\_ind} (P : \text{forall} \ fv : \text{nat}, \text{process} \ fv -> \text{Type}) : \\
\hspace{1cm} ... -> \text{forall} (fv : \text{nat}) (p : \text{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.+) (p1 q1 : process fv.+) :

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) :

 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 \( \Rightarrow [\ ^ \ block \ ] \) intro pattern

\[
\text{elim: ty}_p \Rightarrow [^ p].
\]

- Some operations have “hard” syntactic type-requirements

\[
\text{Fixpoint subst \{fv\} \ldots (p : process fv.+1) : process fv := \ldots}
\]

\[
\text{elim/proc2: p \Rightarrow [^ p] in v c \ast}.
\]
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 \( \Rightarrow[^ block ] \) in the pipes
  - easy-to-repair proofs
Thanks for listening!

Questions?