SESSION TYPES IN COQ

ORNELA DARDHA

UNIVERSITY OF GLASGOW

BEHAPI WORKSHOP

PRAGUE, 06/04/2019

HOW THIS WORK CAME TO BE…

- MSc student project during 3 months of summer 2018 (completed successfully by **Eric Dilmore**)
- "Session types in Coq" project proposal/call:

" *Session types* are a type formalism which allow specification and verification of protocols among software components in distributed systems *[…]*

The *π-calculus* is a model of computation *[…]*

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs.

The goal of this project is to formalise the session π-calculus in Coq, including syntax, semantics and type system. Finally, the project will include certification of fundamental theorems, of type preservation and type safety for the session π-calculus. "

THE Π-CALCULUS

- A **computational model** for *communication* and *concurrency*
- Channels/names (*x,y*…) and processes (*P,Q*…) are key concepts.
- Communication occurs between two processes in *parallel composition*

$$
\begin{array}{lll}
P, Q ::= & \mathbf{0} & \text{inaction} \\
& (\boldsymbol{\nu} xy)P & \text{scope restriction} \\
& \overline{u}\langle v \rangle.P & \text{output} \\
& u(x).P & \text{input} \\
& u \triangleleft l_j.P & \text{selection} \\
& u \triangleright \{l_i : P_i\}_{i \in I} & \text{branching} \\
& P \mid Q & \text{parallel composition}
\end{array}
\qquad
\begin{array}{lll}
u, v ::= & x & \text{name} \\
& * & \text{unit}
\end{array}
$$

(BINARY) SESSION TYPES

- *Session types* model communication protocols among distributed systems components
- Specify *type*, *direction* and *order* of data exchanged.

$$T ::= \quad \texttt{Unit} \quad \text{unit type} \qquad S \quad \text{session type}$$

$$S ::= \quad \texttt{End} \quad \text{termination} \qquad !T.S \quad \text{send} \qquad ?T.S \quad \text{receive} \qquad \&\{l_i : S_i\}_{i \in I} \quad \text{branch} \qquad \oplus\{l_i : S_i\}_{i \in I} \quad \text{select}$$

COQ PROOF ASSISTANT

- "*Coq is a proof assistant. It means that it is designed to develop mathematical proofs, and especially to write formal specifications, programs, and proofs that programs comply to their specifications.*"

- The Coq Proof Assistant provides
  - *Gallina*: a functional, dependently-typed programming language
  - A *tactics* language: for constructing proofs

COQ EXAMPLE

- Gallina's type system has two basic types: `Prop` and `Set`
  - `Prop` for propositions, i.e. well-formed propositions are of *type* `Prop`.
  - `Set` for data structures and functions over them

`Type` is a super type of `Prop` and `Set`

```
Inductive bool : Set :=
  | true : bool
  | false : bool.

Inductive seq : nat -> Set :=
  | niln : seq 0
  | consn : forall n : nat, nat -> seq n -> seq (S n).
```

PROJECT DESIGN

- Crucial choices: name binding and typing environments → the path taken influences the proofs!
- Name binding (restriction and input):
  - String names
  - **De Bruijn indices**
  - Locally nameless
  - Parametric Higher-Order Abstract Syntax (PHOAS)
- Type environments:
  - Partial functions: name → types
  - Association list
  - **List of optional types**

STRING NAMES

Straightforward approach of using strings names for channel names. `(new x y) (x ! Unit; P0 ||| y ? z; P0)`

- **Pros**
  - Immediate visual symmetry of π-calculus theory and its representation in Coq
  - Greater confidence of correctness of implementation
- **Cons**
  - Difficulties of "fresh names"
    - *Alpha-renaming* or *Barendregt convention* are possible, but need lemmas etc.
  - Difficulties of correctly implementing to type environments

DE BRUIJN INDICES (OUR CHOICE!)

- Name representation using `nats`: name n refers to the n[th] -nearest binding

$$(\boldsymbol{\nu} xy)(\overline{x}\langle * \rangle.\mathbf{0} \mid y(z).\overline{y}\langle z \rangle).\mathbf{0} \quad \Rightarrow \quad (\boldsymbol{\nu})(\overline{1}\langle * \rangle.\mathbf{0} \mid 0().\overline{1}\langle 0 \rangle).\mathbf{0}.$$

- **Pros**
  - Solves the problem of fresh names (no alpha-renaming)
  - Renaming becomes mechanical and systematic (*lift all free variable* indices when new bindings are introduced)
  - Allows for list-like, as opposed to map-like structure for type environments

- **Cons**
  - Visual disconnection from standard π-calculus processes
  - Shifting variables upon expanding/removing a scope → increased effort in scope expansion

LOCALLY NAMELESS

- Different convention for free and bound names: de Bruijn indices for bound and *atoms* for free names.

- **Pros**
  - Same as de Bruijn
  - More readability for free variables
  - Substitution is easier due to division of bound/free names

- **Cons**
  - More cases to analyse (free names)
  - Variable shifting replaced by opening/closing scopes
  - Typechecking is harder due to opening/closing of scopes

PARAMETRIC HIGHER-ORDER ABSTRACT SYNTAX (PHOAS)

- The HOAS uses the metalanguage itself to represent names
- The PHOAS: the `name` type becomes a *variable*, postponing the actual type of names until we need them.

```
forall x : name,
  PInput (VName x) (fun y : name => POutput (VName y) VUnit P0)
```

- **Pros**
  - Names are always fresh, as they are handled by Coq (!)
  - Visually-natural method of writing processes.
  - Seems more adequate and is left as **future work**

- **Cons**
  - More difficult to implement than de Bruijn indices…

TYPE ENVIRONMENTS

- **Type environments** are associations of *names* to *types*.
- Several representations of type environments in Coq, but we want to satisfy the following properties:
  1. A **powerful inductive principle**: important for writing proofs about type environment *split*
  2. **Commutative insert function**: removing and re-adding an element with a new value, while split is still valid

- We'll discuss the choice in terms of these properties.

PARTIAL FUNCTIONS: NAMES → TYPES

- Most basic implementation: names in the environment produce the corresponding type; names outside the environment produce `None`.

- **Pros**
  - Simple implementation
  - Provides a *commutative insert function* through functional *extensionality*

- **Cons**
  - Difficult for the induction principle; a list of stored values needed

ASSOCIATION LIST

- One step from partial functions, is the association list: `list (name * type)`
  - `insert` function adds to the top of the list
  - `lookup` function traverses the list until it finds a pair, or returns `None`

- **Pros**
  - Simple implementation
  - Provides an *inductive principle*

- **Cons**
  - Insert function is not commutative

LIST OF OPTIONAL TYPES (OUR CHOICE!)

- When names are restricted to `nat`, we can use the names themselves to index a simple list of types.

E.g., `[Some Unit; None; Some End]`

Types: 0 as `Unit`; 2 as `End` and `None` is placeholder.

- **Pros**
  - Good inductive principle (it's just a list!)
  - Insert function is commutative
  - Simplifies proofs associated with environment *split*

IMPLEMENTATION

- Project outcome: machine-checkable representation of the π-calculus and binary session types:
  - Process terms
  - Session types
  - Reduction rules
  - Typing environments
  - Typing rules
  - Proofs (many only in Coq, rather than in theory) TBC

Π-CALCULUS PROCESS TERMS

- De Bruijn indices: names as `nat`s, bindings implied by the structure

```
Inductive branch : Type :=
  | Branch :> nat -> branch.

Inductive value : Type :=
  | VUnit : value | VName :> nat -> value.

Inductive process : Type :=
  | P0 : process
  | PNew : process -> process (* two bindings *)
  | PComp : process -> process -> process
  | POutput : value -> value -> process -> process
  | PInput : value -> process -> process (* one binding *)
  | PSelect : value -> branch -> process -> process
  | PBranch : value -> ne_list process -> process.
```

NOTATIONS

- Notations can be used in Coq, allowing us to construct π-calculus terms in a more visually natural way.

```
Notation "u ? ; P" := (PInput u P).
Notation "u ! v ; P" := (POutput u v P).
Notation "P ||| Q" := (PComp P Q).
Notation "'(new)' P" := (PNew P).
Notation "u <| n ; P" := (PSelect u n P).
Notation "u |> l" := (PBranch u l).

Example process_example : process :=
  (new) (1 ! VUnit; P0 ||| 0 ? ; P0).
```

STRUCTURAL CONGRUENCE

```
Inductive proc_congruent : process -> process -> Prop :=
  | PCCompCommutative P Q : P ||| Q === Q ||| P
  | PCCompAssociative P Q R :
      (P ||| Q) ||| R === P ||| (Q ||| R)
  | PCComp0 P : P ||| P0 === P
  | PCScopeExpansion P Q :
      (PNew P) ||| Q === PNew (P ||| incr_free 2 Q)
  | PCScope0 :
      PNew P0 === P0
  | PCScopeCommutative P :
      PNew (PNew P) === PNew (PNew (swap 0 2 (swap 1 3 P)))
  | PCScopeSwap P :
      PNew P === PNew (swap 0 1 P)
  | PCInnerNew P Q : P === Q -> PNew P === PNew Q
  | PCInnerOutput P Q v1 v2 :
      P === Q -> POutput v1 v2 P === POutput v1 v2 Q
```

REDUCTION

```
Inductive reduce : process -> process -> Prop :=
  | RComm P Q payload :
      (new) (1 ! payload; P ||| 0 ?; Q) ==>
        (new) (P ||| subst payload 0 Q)
  | RCase P Qs b Q :
      ne_index b Qs = Some Q ->
      (new) (1 <| b; P ||| 0 |> Qs) ==> PNew (P ||| Q)
  | RRes P Q : P ==> Q -> (new) P ==> (new) Q
  | RPar P Q R : P ==> Q -> P ||| R ==> Q ||| R
  | RStruct P P' Q Q' :
      P === P' -> Q' === Q -> P' ==> Q' ->
      P ==> Q
  where "P ==> Q" := (reduce P Q).
```

SESSION TYPES

```
Inductive type : Set :=
  | TUnit : type
  | TSession :> stype -> type
  with stype : Set :=
  | TEnd : stype
  | TSend : type -> stype -> stype
  | TRecv : type -> stype -> stype
  | TSelect : ne_list stype -> stype
  | TBranch : ne_list stype -> stype.
```

```
Notation "'Unit'" := TUnit.
Notation "'End'" := TEnd.
Notation "! T1 ; T2" := (TSend T1 T2).
Notation "? T1 ; T2" := (TRecv T1 T2).
Notation "& bs" := (TBranch bs).
Notation "(+) bs" := (TSelect bs).
```

DUALITY

```
Fixpoint dual (T : stype) : stype :=
  let dual_branches := fix dual_branches bs : ne_list stype :=
    match bs with
    | ne_nil T => ne_nil (dual T)
    | ne_cons T bs' => ne_cons (dual T) (dual_branches bs')
    end in
  match T with
  | End => End
  | ! T1; T2 => ? T1; dual T2
  | ? T1; T2 => ! T1; dual T2
  | (+) bs => & dual_branches bs
  | & bs => (+) dual_branches bs
  end.
```

```
Lemma dual_inverse :
  forall T,
    dual (dual T) = T.
```

TYPE ENVIRONMENTS AND TYPING RULES

```
Definition env := list (option type).
```

```
Reserved Notation "G '|-v' u : T".

Inductive types_value : env -> value -> type -> Prop :=
  | TVName :
      forall G x T,
        un_env (raw_replace x None G) -> lookup x G = Some T ->
        G |-v x : T
  | TVVal :
      forall G,
        un_env G ->
        G |-v VUnit : Unit
  where "G '|-v' u : T" := (types_value G u T).
```

```
Inductive types_process : env -> process -> Prop :=
  | TPInact :
      forall G, un_env G -> G |-p P0
  | TPPar :
      forall G G1 G2 P Q,
        G1 |-p P -> G2 |-p Q -> G ~= G1 + G2 ->
        G |-p P ||| Q
  | TPRes :
      forall (G : env) (T : stype) P,
        G;; TSession (dual T);; TSession T |-p P ->
        G |-p (new) P
  | TPIn :
      forall (G G1 G2 : env) (x : nat) T S P,
        G1 |-v x : ? T ; S ->
        (replace x (TSession S) G2);; T |-p P ->
        G ~= G1 + G2 ->
        G |-p x ? ; P
  | TPOut :
      forall G G1 G2 G3 (x : nat) u T S P,
        G1 |-v x : ! T ; S -> G2 |-v u : T ->
        replace x (TSession S) G3 |-p P ->
        G ~= G1 + G2 + G3 ->
        G |-p x ! u; P
```

TOWARDS TYPE PRESERVATION…

Lemma (Unrestricted weakening)**:** *If* $\Gamma \vdash P$ *and* $\mathrm{un}(T)$ *then* $\Gamma, x : T \vdash P$.

```
Theorem unrestricted_weakening :
  forall P G,
    G |-p P ->
    forall T x, un T -> lookup x G = None ->
    (replace x T G) |-p P.
```

Lemma (Strengthening)**:** *Let* $\Gamma \vdash P$ *and* $x \notin fv(P)$.

1. *If* $\neg\, \mathrm{un}(T)$ *then* $(x : T) \notin \Gamma$.
2. *If* $\Gamma = \Gamma', x : T$ *then* $\Gamma' \vdash P$.

```
Lemma strengthening1 :
  forall G P,
    G |-p P ->
    forall x T, ~Free x P ->
    (~un T -> ~lookup x G = Some T).
Lemma strengthening2 :
  forall G P,
    G |-p P ->
    forall x, ~Free x P ->
    raw_replace x None G |-p P.
```

TOWARDS TYPE PRESERVATION …

Lemma (Preservation for $\equiv$)**:** *If* $\Gamma \vdash P$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q$.

```
Lemma pc_preservation :
  forall P Q,
    P === Q ->
    forall G, G |-p P <-> G |-p Q.
Admitted.
```

Lemma (Preservation for substitution)**:** *If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T \vdash P$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$ *then* $\Gamma \vdash P[v/x]$.

```
Theorem subst_preservation :
  forall G G1 G2 v T P,
    G1 |-v v : T ->
    G2;; T |-p P ->
    G ~= G1 + G2 ->
    G |-p subst v 0 P.
Admitted.
```

TOWARDS TYPE PRESERVATION …

Theorem (Type preservation)**:** *If* $\Gamma \vdash P$ *and* $P \to Q$ *then* $\Gamma \vdash Q$.

```
Theorem type_preservation :
  forall P Q,
    P ==> Q ->
    forall G, G |-p P ->
    G |-p Q.
Admitted.
```

- Proof by induction on `P ==> Q`
- "*Most of the effort required here involves finding the useful information amidst the noise of a screen full of facts. […]*"

CONCLUSION AND FUTURE WORK

- Discussed alternative representations for the design of the formalisation
- We formalised: process terms, session types, type environment, typing rules and some auxiliary lemmas

- Finalise **Type Preservation** and move onto **Type Safety**
- Replication and recursion
- Rewrite code into a *library*
- Explore PHOAS name binding representation
- …

Thank you!

Questions??