Thinking About Mechanizing the Meta-Theory of Session Types

Francisco Ferreira
(joint work with Nobuko Yoshida)

17th Dec
ABCD Meeting - Imperial College London

Engineering the Meta-Theory of Session Types

Francisco Ferreira
(joint work with Nobuko Yoshida)

17th Dec
ABCD Meeting - Imperial College London

"The limits of my language mean the limits of my world."

–Ludwig Wittgenstein

Who Am I?

- I did my **PhD** at **McGill University**, advised by **Brigitte Pientka.**
- I worked with **Higher Order Abstract Syntax.**
- Also on the **meta-theory** of programming languages.

Who Am I?

- I did my **PhD** at **McGill University**, advised by **Brigitte Pientka.**
- I worked with **Higher Order Abstract Syntax.**
- Also on the **meta-theory** of programming languages.
- I worked in the implementation of:

Who Am I?

- I did my **PhD** at **McGill University**, advised by **Brigitte Pientka.**
- I worked with **Higher Order Abstract Syntax.**
- Also on the **meta-theory** of programming languages.
- I worked in the implementation of:
    - Beluga — My supervisor's project on **computational reasoning about LF definitions**.

Who Am I?

- I did my **PhD** at **McGill University**, advised by **Brigitte Pientka.**
- I worked with **Higher Order Abstract Syntax.**
- Also on the **meta-theory** of programming languages.
- I worked in the implementation of:
    - Beluga — My supervisor's project on **computational reasoning about LF definitions**.
    - Babybel — Our project on supporting **HOAS in functional programming languages** (e.g.: OCaml).

Who Am I?

- I did my **PhD** at **McGill University**, advised by **Brigitte Pientka.**
- I worked with **Higher Order Abstract Syntax.**
- Also on the **meta-theory** of programming languages.
- I worked in the implementation of:
  - Beluga — My supervisor's project on **computational reasoning about LF definitions**.
  - Babybel — Our project on supporting **HOAS in functional programming languages** (e.g.: OCaml).
  - Orca — Our project on combining **HOAS and Type Theory.**

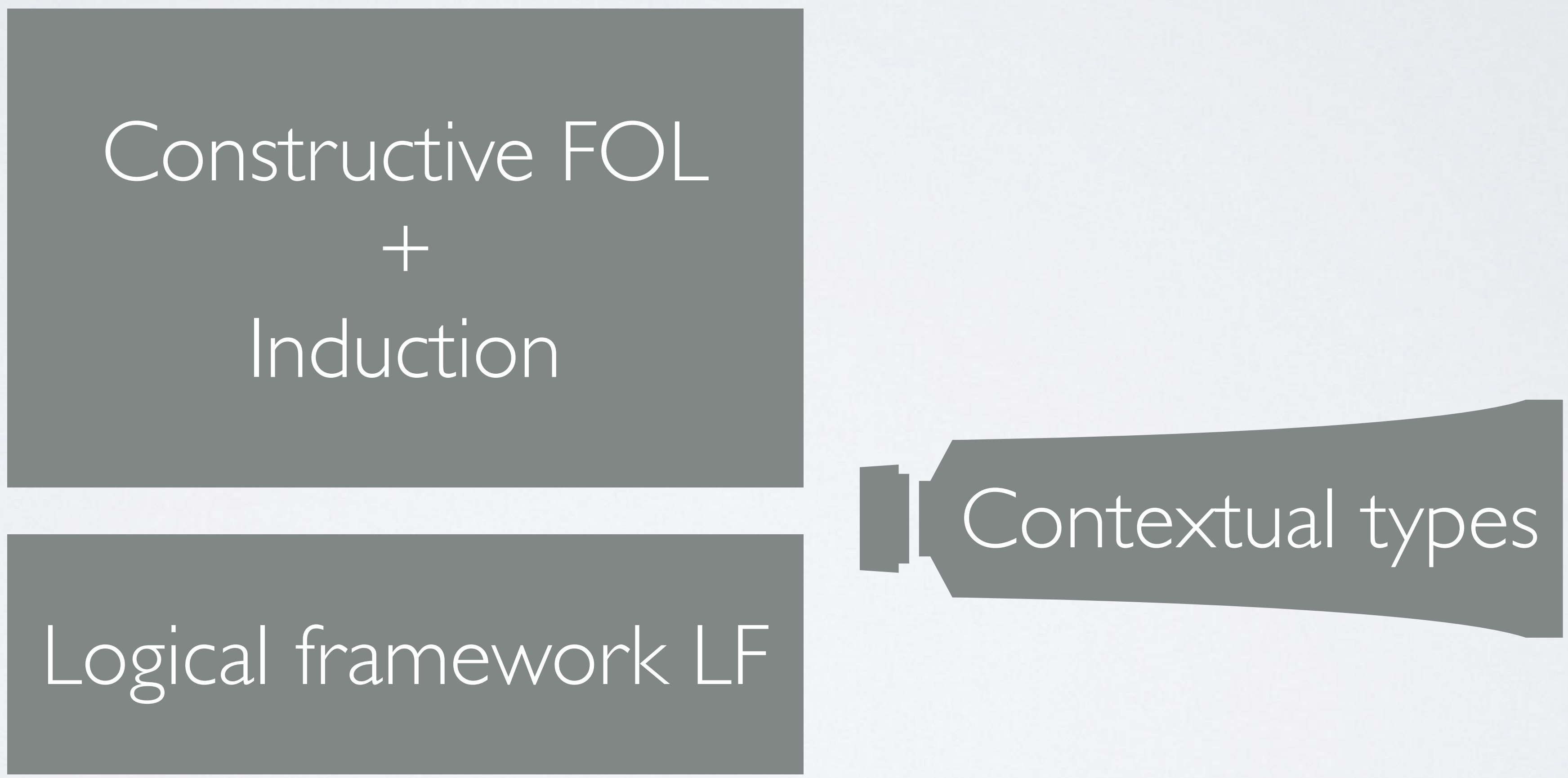
Mechanising the Meta-Theory Session Types

- Names are ubiquitous.
- The binding structure is **quite** rich.
- Channels are handled linearly.
- Names exist besides binders. Names are a first class notion.

The First Step

- Do a case study:
  - Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited, by Yoshida and Vasconcelos.

How Best To Represent Session Types Calculi?



How Best To Represent Session Types Calculi?

Constructive FOL
+
Induction

Nominal Equation Logic

But, Really? Another Proof Assistant?

But, Really? Another Proof Assistant?

- What if we relax the requirement for α-conversion?

But, Really? Another Proof Assistant?

- What if we relax the requirement for α-conversion?
- Work by Ernesto Copello, Maribel Fernandez, et al.
  - Defines a notion of α-compatible relations.
  - Defines a notion of α-structural induction.

But, Really? Another Proof Assistant?

- What if we relax the requirement for α-conversion?

- Work by Ernes Fernandez, et al.

It can be readily implemented in **Agda** and **Coq!**

  - Defines a notion of α-compatible relations.

  - Defines a notion of α-structural induction.

But, Really? Another Proof Assistant?

- What if we relax the requirement for **α**-conversion?
- Work by Erne… …z, et al.
  - Defines a notion of **α**…
  - Defines a notion of **α**-structu… induc…ion.

It can be reac… implement… and **c**…

Induction on judgments is still an "it should be possible" problem in this approach.

Time To Consider Existing Solutions

- Well established work on **Locally Nameless**:
  - Use names for free variables.
  - Use indices for bound variables.
  - Mediate between them with open & close operations.

STLC

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs} \quad t \quad | \quad \mathsf{app}\, t\, t$$

STLC

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs}\quad t \quad | \quad \mathsf{app}\, t\, t$$

STLC

$$t \quad := \quad \text{bvar}\, x \quad | \quad \text{fvar}\, p \quad | \quad \underline{\text{abs} \quad t} \quad | \quad \text{app}\, t\, t$$

STLC

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs} \quad t \quad | \quad \mathsf{app}\, t\, t$$

STLC

$$t \quad := \quad \text{bvar}\, x \quad | \quad \text{fvar}\, p \quad | \quad \text{abs} \quad t \quad | \quad \text{app}\, t\, t$$

$$t^x \quad \equiv \quad \{0 \to x\}\, t \qquad\qquad \backslash^x t \quad \equiv \quad \{0 \leftarrow x\}\, t$$

STLC

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs} \quad t \quad | \quad \mathsf{app}\, t\, t$$

$$t^x \quad \equiv \quad \{0 \to x\}\, t \qquad\qquad \backslash^x t \quad \equiv \quad \{0 \leftarrow x\}\, t$$

$$\frac{\text{ok } E \qquad (x : T) \in E}{E \vdash \mathsf{fvar}\, x : T} \text{ TYPING-VAR} \qquad \frac{E \vdash t_1 : T_1 \to T_2 \qquad E \vdash t_2 : T_1}{E \vdash \mathsf{app}\, t_1\, t_2 : T_2} \text{ TYPING-APP}$$

$$\frac{\forall x \notin L, \quad E, x : T_1 \vdash t^x : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2} \text{ TYPING-ABS}$$

STLC

$$t \quad := \quad \textsf{bvar}\, x \quad | \quad \textsf{fvar}\, p \quad | \quad \textsf{abs} \quad t \quad | \quad \textsf{app}\, t\, t$$

$$t^x \quad \equiv \quad \{0 \to x\}\, t \qquad\qquad {}^{\backslash x}t \quad \equiv \quad \{0 \leftarrow x\}\, t$$

$$\frac{\text{ok } E \qquad (x : T) \in E}{E \vdash \textsf{fvar}\, x : T}\ \text{TYPING-VAR} \qquad \frac{E \vdash t_1 : T_1 \to T_2 \qquad E \vdash t_2 : T_1}{E \vdash \textsf{app}\, t_1\, t_2 : T_2}\ \text{TYPING-APP}$$

$$\frac{\forall x \notin L, \quad E,\ x : T_1 \vdash t^x : T_2}{E \vdash \textsf{abs}\, t : T_1 \to T_2}\ \text{TYPING-ABS}$$

STLC

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs} \quad t \quad | \quad \mathsf{app}\, t\, t$$

$$t^{x} \quad \equiv \quad \{0 \to x\}\, t \qquad\qquad \backslash^{x} t \quad \equiv \quad \{0 \leftarrow x\}\, t$$

$$\frac{\mathrm{ok}\ E \qquad (x : T) \in E}{E \vdash \mathsf{fvar}\, x : T}\ \textsc{typing-var} \qquad \frac{E \vdash t_1 : T_1 \to T_2 \qquad E \vdash t_2 : T_1}{E \vdash \mathsf{app}\, t_1\, t_2 : T_2}\ \textsc{typing-app}$$

$$\frac{\forall x \notin L \qquad E,\ x : T_1 \vdash t^{x} : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2}\ \textsc{typing-abs}$$

STLC

$$t \quad := \quad \mathsf{bvar}\, x \quad | \quad \mathsf{fvar}\, p \quad | \quad \mathsf{abs} \quad t \quad | \quad \mathsf{app}\, t\, t$$

$$t^x \quad \equiv \quad \{0 \to x\}\, t \qquad\qquad {}^{\backslash x}t \quad \equiv \quad \{0 \leftarrow x\}\, t$$

$$\frac{\mathrm{ok}\, E \qquad (x : T) \in E}{E \vdash \mathsf{fvar}\, x : T}\ \textsc{typing-var} \qquad \frac{E \vdash t_1 : T_1 \to T_2 \qquad E \vdash t_2 : T_1}{E \vdash \mathsf{app}\, t_1\, t_2 : T_2}\ \textsc{typing-app}$$

$$\frac{\forall x \notin L, \quad E,\ x : T_1 \vdash t^x : T_2}{E \vdash \mathsf{abs}\, t : T_1 \to T_2}\ \textsc{typing-abs}$$

Open and close should admit several lemmas:

- Opening locally closed terms does not change the term
- Opening and substitution commute
- The interaction of opening and substitutions of variables

$$\frac{\text{ok } E \quad \cdots}{E \vdash \cdots} \text{ TYPING-APP}$$

The Send Receive System and its Cousins the Relaxed and the Revisited System.

Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 171 (2007) 73–93
www.elsevier.com/locate/entcs

Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited:
Two Systems for Higher-Order Session Communication

Nobuko Yoshida[1]
Imperial College London

Vasco T. Vasconcelos[2]
University of Lisbon

The Send Receive System and its Cousins the Relaxed and the Revisited System.

Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 171 (2007) 73–93

Electronic Note
Theoretical Co
Science

www.elsevi

Language Primitives and Type Dis
Structured Communication-Ba
Programming Revisited:
Two Systems for
Higher-Order Session Communi

Nobuko Yoshida[1]
Imperial College London

Vasco T. Vasconcelos[2]
University of Lisbon

Start developing the infrastructure and eventually move on to MPST

A Tale of Three Systems

- We set out to represent the three systems described in the paper:
  - The Honda, Vasconcelos, Kubo system from ESOP'98
  - Its naïve but ultimately unsound extension
  - Its revised system inspired by Gay and Hole in Acta Informatica

The Send Receive System

$P ::= \texttt{request}\ a(k)\ \texttt{in}\ P$	session request
$\mid \texttt{accept}\ a(k)\ \texttt{in}\ P$	session acceptance
$\mid k![\tilde{e}]; P$	data sending
$\mid k?(\tilde{x})\ \texttt{in}\ P$	data reception
$\mid k \triangleleft l; P$	label selection
$\mid k \triangleright \{l_1 : P_1 \talloblong \cdots \talloblong l_n : P_n\}$	label branching
$\mid \texttt{throw}\ k[k']; P$	channel sending
$\mid \texttt{catch}\ k(k')\ \texttt{in}\ P$	channel reception
$\mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional branch
$\mid P \mid Q$	parallel composition
$\mid \texttt{inact}$	inaction
$\mid (\nu u)P$	name/channel hiding
$\mid \texttt{def}\ D\ \texttt{in}\ P$	recursion
$\mid X[\tilde{e}\tilde{k}]$	process variables
$e ::= c$	constant
$\mid e + e' \mid e - e' \mid e \times e \mid \texttt{not}(e) \mid \ldots$	operators
$D ::= X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \texttt{and} \cdots \texttt{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n$	declaration for recursion

The Send Receive System

$P \ ::= \texttt{request}\ a(k)\ \texttt{in}\ P$	session request
$\mid \texttt{accept}\ a(k)\ \texttt{in}\ P$	session acceptance
$\mid k![\tilde{e}]; P$	data sending
$\mid k?(\tilde{x})\ \texttt{in}\ P$	data reception
$\mid k \lhd l; P$	label selection
$\mid k \rhd \{l_1 : P_1 \talloblong \cdots \talloblong l_n : P_n\}$	label branching
$\mid \texttt{throw}\ k[k']; P$	channel sending
$\mid \texttt{catch}\ k(k')\ \texttt{in}\ P$	channel reception
$\mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional branch
$\mid P \mid Q$	parallel composition
$\mid \texttt{inact}$	inaction
$\mid (\nu u)P$	name/channel hiding
$\mid \texttt{def}\ D\ \texttt{in}\ P$	recursion
$\mid X[\tilde{e}\tilde{k}]$	process variables
$e \ ::= c$	constant
$\mid e + e' \mid e - e' \mid e \times e \mid \texttt{not}(e) \mid \ldots$	operators
$D \ ::= X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \texttt{and} \cdots \texttt{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n$	declaration for recursion

α-Conversion for Free

- The original system depends crucially on names

$$(\texttt{throw}\ k[k']; P_1) \mid (\texttt{catch}\ k(k')\ \texttt{in}\ P_2) \ \rightarrow\ P_1 \mid P_2$$

α-Conversion for Free

- The original system depends crucially on names

$$(\texttt{throw}\ k[\underline{k'}]; P_1) \mid (\texttt{catch}\ k(\underline{k'})\ \texttt{in}\ P_2) \ \rightarrow\ P_1 \mid P_2$$

α-Conversion for Free

- The original system depends crucially on names

$$(\texttt{throw}\ k[k'];P_1) \mid (\texttt{catch}\ k(k')\ \texttt{in}\ P_2) \rightarrow P_1 \mid P_2$$

This is a bound variable.

α-Conversion for Free

- The original system depends crucially on names

$$(\texttt{throw}\ k[k']; P_1) \mid (\texttt{catch}\ k(k')\ \texttt{in}\ P_2) \rightarrow P_1 \mid P_2$$

This is a bound variable.

- If α-conversion is built in, this rule collapses to:

$$(\texttt{throw}\ k[k']; P_1) \mid (\texttt{catch}\ k(k'')\ \texttt{in}\ P_2) \rightarrow P_1 \mid P_2[k'/k'']$$

α-Conversion for Free

- The original system depends crucially on names

$$(\texttt{throw}\ k[k'];P_1) \mid (\texttt{catch}$$

ble.

- If α-conversion is built … o:

$$(\texttt{throw}\ k[k'];P_1) \mid (\texttt{catch}\ k(k'')\ \texttt{i}$$

Locally Nameless makes it impossible to express the original system's name handling!

The Typing Judgement

The rule for parallel composition is where the fun begins:

The Typing Judgement

The rule for parallel composition is where the fun begins:

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \qquad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} (\Delta \asymp \Delta') \qquad [\textsc{Conc}]$$

The Typing Judgement

The rule for parallel composition is where the fun begins:

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \qquad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} (\Delta \asymp \Delta') \qquad [\textsc{Conc}]$$

The Typing Judgement

The rule for parallel composition is where the fun begins:

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \qquad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} (\Delta \asymp \Delta') \qquad [\textsc{Conc}]$$

The Typing Judgement

The rule for parallel composition is where the fun begins:

$$\frac{\Theta;\Gamma \vdash P \triangleright \Delta \qquad \Theta;\Gamma \vdash Q \triangleright \Delta'}{\Theta;\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} (\Delta \asymp \Delta') \qquad [\textsc{Conc}]$$

Definition 2.4 (Type algebra) *Typings* Δ_0 *and* Δ_1 *are* compatible, *written* $\Delta_0 \asymp \Delta_1$, *if* $\Delta_0(k) = \Delta_1(k)$ *for all* $k \in \mathrm{dom}(\Delta_0) \cap \mathrm{dom}(\Delta_1)$. *When* $\Delta_0 \asymp \Delta_1$, *the* composition of Δ_0 and Δ_1, *written* $\Delta_0 \circ \Delta_1$, *is given as a typing such that* $(\Delta_0 \circ \Delta_1)(k)$ *is (1)* $\perp$, *if* $k \in \mathrm{dom}(\Delta_0) \cap \mathrm{dom}(\Delta_1)$; *(2)* $\Delta_i(k)$, *if* $k \in \mathrm{dom}(\Delta_i) \setminus \mathrm{dom}(\Delta_{i+1 \bmod 2})$ *for* $i \in \{0, 1\}$; *and (3) undefined otherwise.*

Typing Environments

```
Definition tp_env := {finMap atom_ordType → tp}.

(* lift dual to option *)
Definition option_dual (d : option tp) : option tp :=
  match d with
  | None ⇒ None
  | Some T ⇒ Some (dual T)
  end.

(* compatible envs *)
Definition compatible (D1 D2 : tp_env) : bool :=
  all (fun k ⇒ fnd k D1 == option_dual (fnd k D2))
      (filter (fun k ⇒ k \in supp D1) (supp D2)).

(* composition of envs *)
Definition comp (D1 D2 : tp_env) : tp_env :=
  let: (D1, D12, D2) := split D1 D2 in
  fcat (fcat D1 (update_all_with bot D12)) D2.
```

Typing Environments

```
Definition tp_env := {finMap atom_ordType → tp}.

(* lift dual to option *)
Definition option_dual (d : option tp) : option tp :=
  match d with
  | None ⇒ None
  | Some T ⇒ Some (dual T)
  end.

(* compatible envs *)
Definition compatible (D1 D2 : tp_env) : bool :=
  all (fun k ⇒ fnd k D1 == option_dual (fnd k D2))
      (filter (fun k ⇒ k \in supp D1) (supp D2)).

(* composition of envs *)
Definition comp (D1 D2 : tp_env) : tp_env :=
  let: (D1, D12, D2) := split D1 D2 in
  fcat (fcat D1 (update_all_with bot D12)) D2.
```

Typing Environments

Typing Environments

- Store their assumptions in a unique order **(easy to compare)**
- Only store unique assumptions **(easy to split)**

Typing Environments

- Store their assumptions in a unique order **(easy to compare)**
- Only store unique assumptions **(easy to split)**

This together requires implementing our own LN infrastructure. But it allows for names and linearity.

The Revisited System

- Now we distinguish between the endpoints of channels.
- It can be represented with LN-variables and names.

Two Kinds of Atoms

```
(* variables that can be substituted
   for channels and expressions *)
Inductive var :=
| Free of VA.atom (* a variable waiting to be instantiated *)
| Bound of nat (* a bound variable *)
.


(* The variables for channel names,
   bound in restrictions (Never substituted) *)
Inductive nvar :=
| NFree of NA.atom
| NBound of nat
.
```

Two Kinds of Atoms

```
(* variables that can be substituted
   for channels and expressions *)
Inductive var :=
| Free of VA.atom (* a variable waiting to be instantiated *)
| Bound of nat (* a bound variable *)
.


(* The variables for channel names,
   bound in restrictions (Never substituted) *)
Inductive nvar :=
| NFree of NA.atom
| NBound of nat
.
```

Channels and Expressions

```
(* Channels use both *)
Inductive channel :=
| Ch of (nvar * polarity) %type (* a channel with polarity *)
| Var of var
.

(* Expressions use only variables *)
Inductive exp : Set :=
| tt| ff| ...
| V of var
.
```

Channels and Expressions

```
(* Channels use both *)
Inductive channel :=
| Ch of (nvar * polarity) %type (* a channel with polarity *)
| Var of var
.

(* Expressions use only variables *)
Inductive exp : Set :=
| tt| ff| ...
| V of var
.
```

Channels and Expressions

```
(* Channels use both *)
Inductive channel :=
| Ch of (nvar * polarity) %type (* a channel with polarity *)
| Var of var
.

(* Expressions use only variables *)
Inductive exp : Set :=
| tt| ff| ...
| V of var
.
```

Processes

```
(* processes bind variables and channels,
   but they are in channels and expressions*)
Inductive proc : Set :=
| par : proc → proc → proc
| send : channel → exp → proc → proc
| receive : channel → proc → proc

| throw : channel → channel → proc → proc
| catch : channel → proc → proc

| nu_nm : proc → proc (* hides a name *)
| nu_ch : proc → proc (* hides a channel name *)
| ...
.
```

Binders are “invisible”

Processes

```
(* processes bind variables and channels,
   but they are in channels and expressions*)
Inductive proc : Set :=
| par : proc → proc → proc
| send : channel → exp → proc → proc
| receive : channel → proc → proc

| throw : channel → channel → proc → proc
| catch : channel → proc → proc

| nu_nm : proc → proc (* hides a name *)
| nu_ch : proc → proc (* hides a channel name *)
| ...
.
```

Binders are “invisible”

Processes

```
(* processes bind variables and channels,
   but they are in channels and expressions*)
Inductive proc : Set :=
| par : proc → proc → proc
| send : channel → exp → proc → proc
| receive : channel → proc → proc

| throw : channel → channel → proc → proc
| catch : channel → proc → proc

| nu_nm : proc → proc (* hides a name *)
| nu_ch : proc → proc (* hides a channel name *)
| ...
.
```

Binders are “invisible”

Processes

```
(* processes bind variables and channels,
   but they are in channels and expressions*)
Inductive proc : Set :=
| par : proc → proc → proc
| send : channel → exp → proc → proc
| receive : channel → proc → proc

| throw : channel → channel → proc → proc
| catch : channel → proc → proc

| nu_nm : proc → proc (* hides a name *)
| nu_ch : proc → proc (* hides a channel name *)
| ...
.
```

Binders are “invisible”

But Mechanical Proofs Are..

- Well, very mechanical. We have to be very precise with the theorems.

The typing judgements:

```
Inductive oft_exp (G : sort_env) : exp → sort → Prop :=
...

Inductive oft : sort_env → proc → tp_env → Prop :=
...
```

One of the Substitution Lemmas

Lemma 3.1 (Channel Replacement) *If* $\Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \alpha$, *then* $\Theta; \Gamma \vdash P[\kappa^p/x] \triangleright \Delta \cdot \kappa^p : \alpha$.

Proof. A straightforward induction on the derivation tree for P.

One of the Substitution Lemmas

Lemma 3.1 (Channel Replacement) *If* $\Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \alpha$, *then* $\Theta; \Gamma \vdash P[\kappa^p/x] \triangleright \Delta \cdot \kappa^p : \alpha$.

Proof. A straightforward induction on the derivation tree for P.

Becomes:

```
Theorem ChannelReplacement G P x kp D:
  def (subst_env_ch x (ce kp) D) →
  oft G P D → oft G (s[ x ⇝ (ch kp)]p P) (subst_env_ch x (ce kp) D).
Proof.
(* ... *)
```

One of the Substitution Lemmas

Lemma 3.1 (Channel Replacement) *If* $\Theta;\Gamma \vdash P \triangleright \Delta \cdot x : \alpha$, *then* $\Theta;\Gamma \vdash P[\kappa^p/x] \triangleright \Delta \cdot \kappa^p : \alpha$.

Proof. A straightforward induction on the derivation tree for P.

Becomes:

```
Theorem ChannelReplacement G P x kp D:
  def (subst_env_ch x (ce kp) D) →
  oft G P D → oft G (s[ x ⇝ (ch kp)]p P) (subst_env_ch x (ce kp) D).
Proof.
(* ... *)
```

One of the Substitution Lemmas

Lemma 3.1 (Channel Replacement) *If* $\Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \alpha$, *then* $\Theta; \Gamma \vdash P[\kappa^p/x] \triangleright \Delta \cdot \kappa^p : \alpha$.

Proof. A straightforward induction on the derivation tree for P.

Becomes:

```
Theorem ChannelReplacement G P x kp D:
  def (subst_env_ch x (ce kp) D) → 
  oft G P D → oft G (s[ x ⇝ (ch kp)]p P) (subst_env_ch x (ce kp) D).
Proof.
(* ... *)
```

One of the Substitution Lemmas

Lemma 3.1 (Channel Replacement) *If* $\Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \alpha$*, then* $\Theta; \Gamma \vdash P[\kappa^p/x] \triangleright \Delta \cdot \kappa^p : \alpha$.

Proof. A straightforward induction on the derivation tree for P.

Becomes:

```
Theorem ChannelReplacement G P x kp D:
  def (subst_env_ch x (ce kp) D) →
  oft G P D → oft G (s[ x ⇝ (ch kp)]p P) (su
Proof.
(* ... *)
```

Coq also demanded to be convinced about substituting expressions and various weakening lemmas

Subject Reduction

Theorem 3.3 (Subject Reduction) *If* $\Theta; \Gamma \vdash P \triangleright \Delta$ *with* Δ *balanced and* $P \rightarrow^* Q$*, then* $\Theta; \Gamma \vdash Q \triangleright \Delta'$ *and* Δ' *balanced.*

Subject Reduction

Theorem 3.3 (Subject Reduction) *If* $\Theta; \Gamma \vdash P \triangleright \Delta$ *with* Δ *balanced and* $P \rightarrow^* Q$*, then* $\Theta; \Gamma \vdash Q \triangleright \Delta'$ *and* Δ' *balanced.*

Is straightforward to represent:

```
Theorem SubjectReductionStep G P Q D:
  oft G P D → balanced D → P ⟶ Q → exists D', balanced D' /\ oft G Q D'.
Proof.
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

And Lots of Fun To Prove

```
Lemma SubjectReductionStep' G P Q D D' ka:
  oft G P D → balanced D → P --- ka ---> Q → D ~~~ ka ~~~> D' → oft G Q D'.
(* ... *)

Lemma admissible_label P Q:
  P ⟶ Q → exists ka, P --- ka ---> Q.
(* ... *)

Lemma well_typed_step G P Q D ka:
  oft G P D → P --- ka ---> Q → exists D', D ~~~ ka ~~~> D'.
(* ... *)

Lemma typ_step_preserves_balance D D' ka:
  D ~~~ ka ~~~> D' → balanced D → balanced D'.
(* ... *)
```

Finally:

```
Theorem SubjectReduction G P Q D:
  oft G P D → balanced D → P ⟶ Q → exists D', balanced D' /\ oft G Q D'.
Proof.
  move⇒Hp Hb Hs.

  apply admissible_label in Hs.
  destruct Hs.
  have HH := well_typed_step Hp H.
  destruct HH.
  exists x0.
  split.
  apply: typ_step_preserves_balance ; [apply: H0 | apply: Hb].
  apply: SubjectReductionStep' ;
    [apply: Hp | apply: Hb | apply: H | apply: H0].
Qed.
```

What We Have:

- The definition two systems, the unsound proved with a counter example, and the revised with a proof by induction.
- There are still some lemmas to prove (≈4.5 KLOC so far).
- All using a locally nameless representation
- Some use ssreflect and overloaded-lemmas to simply proofs.
  - More automation using overloaded-lemmas in the future.

What We Have:

- The definition two syst... ...ound proved with a counter example, a... ...proof by induction.
- There are still s... ...5 KLOC so far).
- All using a locally na... ...tion
- Some use ssreflect and overloaded-lemmas to simply proofs.
  - More automation using overloaded-lemmas in the future.

Thanks for your attention. Questions?