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?

- Constructive FOL
- Induction
- Logical framework LF

Contextual types
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 $\alpha$-conversion?
But, Really? Another Proof Assistant?

• What if we relax the requirement for $\alpha$-conversion?

• Work by Ernesto Copello, Maribel Fernandez, et al.
  
  • Defines a notion of $\alpha$-compatible relations.

  • Defines a notion of $\alpha$-structural induction.
But, Really? Another Proof Assistant?

• What if we relax the requirement for \( \alpha \)-conversion?

• Work by Ernesto Copello, Maribel Fernandez, et al.

  • Defines a notion of \( \alpha \)-compatible relations.

  • Defines a notion of \( \alpha \)-structural induction.

  It can be readily implemented in Agda and Coq!
But, Really? Another Proof Assistant?

- What if we relax the requirement for $\alpha$-conversion?
- Work by Ernesto Copello, Maribel Fernandez, et al.
  - Defines a notion of $\alpha$-compatible relations.
  - Defines a notion of $\alpha$-structural induction.
- It can be readily implemented in Agda and Coq!
- 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.
The Locally Nameless Representation

Most issues related to variable bindings can be studied on a language as simple as the \( \lambda \)-calculus. Thus, only the syntax of \( \lambda \)-terms is considered throughout the core of the paper. Support for more advanced binding structures is investigated afterwards (Section 7).

2.1 Named Representations: Raw Terms and Quotiented Terms

The most common representation of \( \lambda \)-terms relies on the use of names: each abstraction and each variable bear a name. The syntax of raw named terms is described by the following grammar.

\[
t ::= \text{var } x \mid \text{abs } x t \mid \text{app } t t
\]

The objects from this grammar are called raw terms because they are not isomorphic to \( \lambda \)-terms. For example, the two raw terms "\text{abs } x (\text{var } x)" and "\text{abs } y (\text{var } y)" are two different objects, although the two \( \lambda \)-terms "\( \lambda x.x \)" and "\( \lambda y.y \)" should be considered equal because the theory of \( \lambda \)-calculus identifies terms that are \( \alpha \)-equivalent. Due to the mismatch between raw terms and \( \lambda \)-terms, there are pieces of reasoning from \( \lambda \)-calculus textbooks that cannot be formalized using raw terms.

In order to obtain a representation of terms truly isomorphic to \( \lambda \)-terms, we need to build a quotient structure, quotienting the set of raw terms with respect to alpha-equivalence. This construction based on a quotient corresponds very closely to the of presentation from standard textbooks on \( \lambda \)-calculus. In practice, though, working formally with a quotient structure is not that straightforward. In order to define a function or a relation on \( \lambda \)-terms, we need to first define it on raw terms, then show it compatible with \( \alpha \)-equivalence, and finally lift it to the quotient structure. For instance, if \( f \) is a unary function on terms in the named representation, then, for \( f \) to be accepted as a definition on \( \lambda \)-terms, we must prove that, for any two alpha-equivalent terms \( t_1 \) and \( t_2 \), the applications \( f(t_1) \) and \( f(t_2) \) yield \( \alpha \)-equivalent results. Lifting definitions to the quotient structure is typically long and tedious. Fortunately, a lot of this work can be automated. For example, Urban's nominal package aims at factoring and automating definitions and proofs about data types involving binders. Yet, at this time, there are still a number of advanced binding structures that are not supported by the nominal package.

2.2 The Locally Named Representation

The locally nameless representation is closely related to the locally named representation, which has been extensively developed by McKinna and Pollack.

This representation syntactically distinguishes between bound variables and free variables. Bound variables are represented using a name, written \( x.F r e e \) variables, also called parameters, are represented using another kind of names, written \( p.A b s t r a c t i o n s \), which always bind "bound variables", carry a bound variable name. The grammar of locally named terms can thus be described as follows.

\[
t ::= \text{bvar } x \mid \text{fvar } p \mid \text{abs } t \mid \text{app } t t
\]

The main interest of the locally named representation is that a bound name and a free name cannot be confused. In particular, one never needs to \( \alpha \)-rename
The Locally Nameless Representation. (The paper by Aydemir et al. contains a survey of binding techniques.) Most issues related to variable bindings can be studied on a language as simple as the $\lambda$-calculus. Thus, only the syntax of $\lambda$-terms is considered throughout the core of the paper. Support for more advanced binding structures is investigated afterwards (Section 7).

2.1 Named Representations: Raw Terms and Quotiented Terms
The most common representation of $\lambda$-terms relies on the use of names: each abstraction and each variable bear a name. The syntax of raw named terms is described by the following grammar.

$$t ::= \text{var } x \mid \text{abs } x t \mid \text{app } tt$$

The objects from this grammar are called raw terms because they are not isomorphic to $\lambda$-terms. For example, the two raw terms "abs $x$ (var $x$)" and "abs $y$ (var $y$)" are two different objects, although the two $\lambda$-terms "$\lambda x.x$" and "$\lambda y.y$" should be considered equal because the theory of $\lambda$-calculus identifies terms that are $\alpha$-equivalent. Due to the mismatch between raw terms and $\lambda$-terms, there are pieces of reasoning from $\lambda$-calculus textbooks that cannot be formalized using raw terms.

In order to obtain a representation of terms truly isomorphic to $\lambda$-terms, we need to build a quotient structure, quotienting the set of raw terms with respect to $\alpha$-equivalence. This construction based on a quotient corresponds very closely to the of presentation from standard textbooks on $\lambda$-calculus.

In practice, though, working formally with a quotient structure is not that straightforward. In order to define a function or a relation on $\lambda$-terms, we need to first define it on raw terms, then show it compatible with $\alpha$-equivalence, and finally lift it to the quotient structure. For instance, if $f$ is a unary function on terms in the named representation, then, for $f$ to be accepted as a definition on $\lambda$-terms, we must prove that, for any two alpha-equivalent terms $t_1$ and $t_2$, the applications $f(t_1)$ and $f(t_2)$ yield $\alpha$-equivalent results. Lifting definitions to the quotient structure is typically long and tedious. Fortunately, a lot of this work can be automated. For example, Urban's nominal package aims to factorize and automate definitions and proofs about data types involving binders. Yet, at this time, there are still a number of advanced binding structures that are not supported by the nominal package.

2.2 The Locally Named Representation
The locally nameless representation is closely related to the locally named representation, which has been extensively developed by McKinna and Pollack.

This representation syntactically distinguishes between bound variables and free variables. Bound variables are represented using a name, written $x$. Frees, also called parameters, are represented using another kind of names, written $p$.

Abstractions, which always bind "bound variables", carry a bound variable name. The grammar of locally named terms can thus be described as follows.

$$t ::= \text{bvar } x \mid \text{fvar } p \mid \text{abs } t \mid \text{app } tt$$

The main interest of the locally named representation is that a bound name and a free name can never confuse. In particular, one never needs to $\alpha$-rename.
The paper by Aydemir et al. [8] contains a survey of binding techniques. Most issues related to variable bindings can be studied on a language as simple as the pure $\lambda$-calculus. Thus, only the syntax of $\lambda$-terms is considered throughout the core of the paper. Support for more advanced binding structures is investigated afterwards (Section 7).

2.1 Named Representations: Raw Terms and Quotiented Terms

The most common representation of $\lambda$-terms relies on the use of names: each abstraction and each variable bear a name. The syntax of raw named terms is described by the following grammar.

\[ t ::= \text{var } x \mid \text{abs } x t \mid \text{app } t t \]

The objects from this grammar are called raw terms because they are not isomorphic to $\lambda$-terms. For example, the two raw terms "\text{abs } x (\text{var } x)" and "\text{abs } y (\text{var } y)" are two different objects, although the two $\lambda$-terms "\text{\lambda } x . x" and "\text{\lambda } y . y" should be considered equal because the theory of $\lambda$-calculus identifies terms that are $\alpha$-equivalent. Due to the mismatch between raw terms and $\lambda$-terms, there are pieces of reasoning from $\lambda$-calculus textbooks that cannot be formalized using raw terms. In order to obtain a representation of terms truly isomorphic to $\lambda$-terms, we need to build a quotient structure, quotienting the set of raw terms with respect to alpha-equivalence. This construction based on a quotient corresponds very closely to the presentation from standard textbooks on $\lambda$-calculus. In practice, though, working formally with a quotient structure is not that straightforward. In order to define a function or a relation on $\lambda$-terms, we need to first define it on raw terms, then show it compatible with $\alpha$-equivalence, and finally lift it to the quotient structure. For instance, if $f$ is a unary function on terms in the named representation, then, for $f$ to be accepted as a definition on $\lambda$-terms, we must prove that, for any two alpha-equivalent terms $t_1$ and $t_2$, the applications $f(t_1)$ and $f(t_2)$ yield $\alpha$-equivalent results. Lifting definitions to the quotient structure is typically long and tedious. Fortunately, a lot of this work can be automated. For example, Urban's nominal package [44] aims to factorize and automate definitions and proofs about data types involving binders. Yet, at this time, there are still a number of advanced binding structures that are not supported by the nominal package.

2.2 The Locally Named Representation

The locally nameless representation is closely related to the locally named representation, which has been extensively developed by McKinna and Pollack [24]. This representation syntactically distinguishes between bound variables and free variables. Bound variables are represented using a name, written $x$. Free variables, also called parameters, are represented using another kind of names, written $p$.

Abstractions, which always bind "bound variables", carry a bound variable name. The grammar of locally named terms can thus be described as follows.

\[ t ::= \text{bvar } x \mid \text{fvar } p \mid \text{abs } x t \mid \text{app } t t \]

The main interest of the locally named representation is that a bound name and an free name can never confuse. In particular, one never needs to $\alpha$-rename.
The Locally Nameless Representation

Most issues related to variable bindings can be studied on \( \lambda \)-calculus. Thus, only the syntax of \( \lambda \)-terms is considered throughout the core of the paper. Support for more advanced binding structures is investigated afterwards (Section 7).

2.1 Named Representations: Raw Terms and Quotiented Terms

The most common representation of \( \lambda \)-terms relies on the use of names: each abstraction and each variable bear a name. The syntax of raw named terms is described by the following grammar.

\[
t : = \text{var } x \mid \text{abs } x t \mid \text{app } tt
\]

The objects from this grammar are called raw terms because they are not isomorphic to \( \lambda \)-terms. For example, the two raw terms "\( \text{abs } x (\text{var } x) \)" and "\( \text{abs } y (\text{var } y) \)" are two different objects, although the two \( \lambda \)-terms "\( \lambda x . x \)" and "\( \lambda y . y \)" should be considered equal because the theory of \( \lambda \)-calculus identifies terms that are \( \alpha \)-equivalent. Due to the mismatch between raw terms and \( \lambda \)-terms, there are pieces of reasoning from \( \lambda \)-calculus textbooks that cannot be formalized using raw terms.

In order to obtain a representation of terms truly isomorphic to \( \lambda \)-terms, we need to build a quotient structure, quotienting the set of raw terms with respect to \( \alpha \)-equivalence. This construction based on a quotient corresponds very closely to the presentation from standard textbooks on \( \lambda \)-calculus.

In practice, though, working formally with a quotient structure is not that straightforward. In order to define a function or a relation on \( \lambda \)-terms, we need to first define it on raw terms, then show it compatible with \( \alpha \)-equivalence, and finally lift it to the quotient structure. For instance, if \( f \) is a unary function on terms in the named representation, then, for \( f \) to be accepted as a definition on \( \lambda \)-terms, we must prove that, for any two \( \alpha \)-equivalent terms \( t_1 \) and \( t_2 \), the two applications \( f(t_1) \) and \( f(t_2) \) yield \( \alpha \)-equivalent results. Lifting definitions to the quotient structure is typically long and tedious. Fortunately, a lot of this work can be automated. For example, Urban's nominal package \([44]\) aims to factorizing and automating definitions and proofs about data types involving binders. Yet, at this time, there are still a number of advanced binding structures that are not supported by the nominal package.

2.2 The Locally Named Representation

The locally nameless representation is closely related to the locally named representation, which has been extensively developed by McKinna and Pollack \([24]\).

This representation syntactically distinguishes between bound variables and free variables. Bound variables are represented using a name, written \( x \). Free variables, also called parameters, are represented using another kind of names, written \( p \).

Abstractions, which always bind "bound variables", carry a bound variable name. The grammar of locally named terms can thus be described as follows.

\[
t : = \text{bvar } x \mid \text{fvar } p \mid \text{abs } x t \mid \text{app } tt
\]

The main interest of the locally nameless representation is that a bound name and a free name can never be confused. In particular, one never needs to \( \alpha \)-rename.

STLC

\[
t : = \text{bvar } x \mid \text{fvar } p \mid \text{abs } t \mid \text{app } tt
\]
STLC

\[
t \; : \; : = \; bvar \; x \; \mid \; fvar \; p \; \mid \; \text{abs} \; t \; \mid \; \text{app} \; t \; t
\]

\[
t^x \; \equiv \; \{0 \rightarrow x\} \; t
\]

\[
\{x\} \; t \; \equiv \; \{0 \leftarrow x\} \; t
\]
STLC

\[
\begin{align*}
t & := \text{bvar } x & \text{fvar } p & \text{abs } t & \text{app } tt \\
\end{align*}
\]

\[
\begin{align*}
t^x & \equiv \{0 \to x\} t & \backslash x t & \equiv \{0 \leftarrow x\} t \\
\end{align*}
\]

\[
\begin{align*}
\text{ok } E & \quad (x : T) \in E \quad \text{E \vdash } f\text{var } x : T & \quad E \vdash t_1 : T_1 \to T_2 & \quad E \vdash t_2 : T_1 \quad \text{E \vdash } \text{app } t_1 t_2 : T_2 \\
\frac{}{E \vdash } & \quad \text{TYing-VAR} & \quad \text{TYing-APP} & \\
\forall x \notin L, \quad E, x : T_1 \vdash t^x : T_2 & \quad \text{E \vdash } \text{abs } t : T_1 \to T_2 \\
\frac{}{E \vdash } & \quad \text{TYing-ABS} \\
\end{align*}
\]
The implementation of variable opening follows a pattern similar to the implementation of variable closing. Its implementation is based on a recursive function, with a bound variable of index $k$.

In practice, though, working formally with a quotient structure is not that straightforward. In particular, one even needs to reason about quotient structures that do not canonize. The paper by Aydemir et al. [365] then, for the sake of presentation from standard textbooks on nominal package (\texttt{Top}), shows it compatible with the named representation, then, for $\lambda$-terms, we need to first define four constructors for variables: one for bound type variables ($\text{typ}_{bvar}$), one for free type variables ($\text{trm}_{bvar}$), one for bound variables ($\text{abs}$), and one for free variables ($\text{fvar}$).

For the sake of presentation of typing and subtyping rules, we introduce the following typing judgments:

- $\text{TYPING-VAR}$: $\text{ok} \; E \quad (x : T) \in E \quad \frac{E \mid fvar \; x : T}{E \mid \text{bvar} \; x}$
- $\text{TYPING-APP}$: $\frac{E \mid t_1 : T_1 \rightarrow T_2 \quad E \mid t_2 : T_1}{E \mid \text{app} \; t_1 \; t_2 : T_2}$
- $\text{TYPING-ABS}$: $\forall x \notin L, \quad \frac{E, \; x : T_1 \mid t^x : T_2}{E \mid \text{abs} \; t : T_1 \rightarrow T_2}$

In particular, all the variables named $x$ are abstracted except those named $y$. For example, the two raw terms "\( \lambda x . x \)" and "\( \lambda y . y \)" are equivalent. Due to the mismatch between raw terms and locally nameless terms, we need to first define the following bindings:

- $t^x \ni \{0 \rightarrow x\} \; t$
- $\backslash x \; t \ni \{0 \leftarrow x\} \; t$
The implementation of variable opening follows a pattern similar to the implementation of variable closing. Its implementation is based on a recursive function, which has been extensively developed by McKinna and Pollack [A. Charguéraud].

The locally named representation is that a bound name and a free type variable be different from the atoms used to represent free type variables. Note that universal types, abstractions and type applications. Environments are made of the empty environment, environments where the free variables named \( k \) have been replaced with a bound variable. The indices of the variables named \( k \) are then incremented each time an abstraction is traversed. When those variables are chosen in such a way that all the bound variables introduced are those that already appear below an abstraction named \( \lambda \).

In order to define a function or a relation on \( \alpha \)-terms, we need to first define

\[
\begin{align*}
\text{ok } E &\quad (x : T) \in E \\
E \vdash \text{fvar } x : T &\quad \text{TYPING-VAR} \\
E \vdash t_1 : T_1 \rightarrow T_2 &\quad E \vdash t_2 : T_1 \\
E \vdash \text{app } t_1 t_2 : T_2 &\quad \text{TYPING-APP} \\
\forall x \notin L &\quad E, x : T_1 \vdash t^x : T_2 \\
E \vdash \text{abs } t : T_1 \rightarrow T_2 &\quad \text{TYPING-ABS}
\end{align*}
\]
STLC

\[
\begin{align*}
t & := \text{bvar } x \mid \text{fvar } p \mid \text{abs } t \mid \text{app } t t \\

\{0 \to x\} t & \equiv t^x \\
{0 \leftarrow x}\ t & \equiv \backslash x t
\end{align*}
\]

\[
\begin{align*}
\text{ok } E & \quad (x : T) \in E \quad \text{TYPING-VAR} & \quad E \vdash t_1 : T_1 \to T_2 & \quad E \vdash t_2 : T_1 \quad \text{TYPING-APP} \\
E \vdash \text{fvar } x : T & \quad \text{TYPING-VAR} & \quad E \vdash \text{app } t_1 t_2 : T_2 & \quad E \vdash \text{abs } t : T_1 \to T_2 \quad \text{TYPING-ABS} \\
\forall x \notin L, \quad E, \ x : T_1 \vdash t^x : T_2 & \quad \text{TYPING-ABS}
\end{align*}
\]
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

\[
\frac{\text{ok } E}{E \vdash T} \quad \text{TYPING-APP}
\]
The Send Receive System and its Cousins the Relaxed and the Revisited System.
The Send Receive System and its Cousins the Relaxed and the Revisited System.

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

\[ P ::= \text{request } a(k) \text{ in } P \]
\[ \quad | \text{accept } a(k) \text{ in } P \]
\[ \quad | k!\lceil \hat{e} \rceil ; P \]
\[ \quad | k?\lceil \hat{x} \rceil \text{ in } P \]
\[ \quad | k \triangleleft l; P \]
\[ \quad | k \triangleright \{l_1 : P_1 \| \cdots \| l_n : P_n\} \]
\[ \quad | \text{throw } k[k']; P \]
\[ \quad | \text{catch } k(k') \text{ in } P \]
\[ \quad | \text{if } e \text{ then } P \text{ else } Q \]
\[ \quad | P | Q \]
\[ \quad | \text{inact} \]
\[ \quad | (\nu u)P \]
\[ \quad | \text{def } D \text{ in } P \]
\[ \quad | X[\check{e}k] \]

\[ e ::= c \]
\[ \quad | e + e' \mid e - e' \mid e \times e \mid \text{not}(e) \mid \ldots \]

\[ D ::= X_1(\check{x}_1\check{k}_1) = P_1 \text{ and } \cdots \text{ and } X_n(\check{x}_n\check{k}_n) = P_n \]

session request  
session acceptance  
data sending  
data reception  
label selection  
label branching  
channel sending  
channel reception  
conditional branch  
parallel composition  
inaction  
name/channel hiding  
recursion  
process variables  
constant  
operators  
declaration for recursion
The Send Receive System

\[ P ::= \text{request } a(k) \text{ in } P \]
\[ | \text{accept } a(k) \text{ in } P \]
\[ | k!\vec{e}; P \]
\[ | k?(\vec{x}) \text{ in } P \]
\[ | k < l; P \]
\[ | k > \{l_1 : P_1 \| \cdots \| l_n : P_n\} \]
\[ | \text{throw } k[k']; P \]
\[ | \text{catch } k(k') \text{ in } P \]
\[ | \text{if } e \text{ then } P \text{ else } Q \]
\[ | P | Q \]
\[ | \text{inact} \]
\[ | (\nu u)P \]
\[ | \text{def } D \text{ in } P \]
\[ | X[\vec{e}k] \]

\[ e ::= c \]
\[ | e + e' | e - e' | e \times e | \text{not}(e) | \ldots \]

\[ D ::= X_1(\vec{x}_1\vec{k}_1) = P_1 \text{ and } \cdots \text{ and } X_n(\vec{x}_n\vec{k}_n) = P_n \]

session request
session acceptance
data sending
data reception
label selection
label branching
channel sending
channel reception
conditional branch
parallel composition
inaction
name/channel hiding
recursion
process variables
constant
operators
declaration for recursion
\(\alpha\)-Conversion for Free

- The original system depends crucially on names

\[
(\text{throw } k[k']; P_1) | (\text{catch } k(k') \text{ in } P_2) \rightarrow P_1 | P_2
\]
\(\alpha\)-Conversion for Free

- The original system depends crucially on names

\[
\text{(throw } k[k'] \text{ in } P_1) \mid (\text{catch } k(k') \text{ in } P_2) \rightarrow P_1 \mid P_2
\]
\textbf{\(\alpha\)-Conversion for Free}

- The original system depends crucially on names

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

This is a bound variable.
\( \alpha \)-Conversion for Free

- The original system depends crucially on names

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

This is a bound variable.

- If \( \alpha \)-conversion is built in, this rule collapses to:

\[(\text{throw } k[k']; P_1) \mid (\text{catch } k(k'') \text{ in } P_2) \rightarrow P_1 \mid P_2[k'/k'']\]
α-Conversion for Free

- The original system depends crucially on names

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

- If α-conversion is built into the system, then:

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

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:

\[
\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta' \\
\Theta; \Gamma \vdash P | Q \triangleright \Delta \circ \Delta' (\Delta \bowtie \Delta')
\]

[Conc]
The Typing Judgement

The rule for parallel composition is where the fun begins:

\[
\frac{\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta'}{(\Delta \asymp \Delta')} \quad [\text{Conc}]
\]
The Typing Judgement

The rule for parallel composition is where the fun begins:

\[
\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta' \\
\frac{}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \quad (\Delta \approx \Delta')
\]

[Conc]
The Typing Judgement

The rule for parallel composition is where the fun begins:

$$\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta' \quad (\Delta \asymp \Delta')$$

[Conc]

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

Definition \( \text{tp_env} \) := \{ \text{finMap atom_ordType} \rightarrow \text{tp} \}.

(* lift dual to option *)
Definition \( \text{option_dual} \) \( (d : \text{option tp}) : \text{option tp} := \)
  match \( d \) with
  | \( \text{None} \) \( \Rightarrow \) \( \text{None} \)
  | \( \text{Some} \ T \) \( \Rightarrow \) \( \text{Some} \ (\text{dual} T) \)
end.

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

(* composition of envs *)
Definition \( \text{comp} \) \( (D1 \ D2 : \text{tp_env}) : \text{tp_env} := \)
  \( \text{let} : \ (D1, D12, D2) := \text{split} \ D1 \ D2 \ \text{in} \)
  \( \text{fcat} \ (\text{fcat} \ D1 \ (\text{update_all_with} \ \text{bot} \ D12)) \ D2. \)
Typing Environments

Definition $tp\_env := \{\text{finMap atom\_ordType} \rightarrow tp\}$.

(* lift dual to option *)
Definition $option\_dual (d : option tp) : option tp :=$
  match $d$ with
  $|$ None $\Rightarrow$ None
  $|$ Some $T \Rightarrow$ Some (dual $T$)
end.

(* compatible envs *)
Definition $compatible (D1 D2 : tp\_env) : bool :=$
  all $(\text{fun } k \Rightarrow \text{find } k \text{ D1} = \text{option\_dual} (\text{find } k \text{ D2}))$
  (filter $(\text{fun } k \Rightarrow k \in \text{supp D1}) (\text{supp D2})$).

(* composition of envs *)
Definition $\text{comp} (D1 D2 : tp\_env) : tp\_env :=$
  let: $(D1, D12, D2) := \text{split D1 D2 in}$
  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

Binders are “invisible”
Processes

Bounders are “invisible”
Processes

Binders are “invisible”
Processes

Binders are “invisible”
But Mechanical Proofs Are...

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

The typing judgements:

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

Inductive oft : sortEnv → 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 $\Theta; \Gamma \vdash P[k^p / x] \triangleright \Delta \cdot k^p : \alpha$.

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) \rightarrow 1
  alt G P D \rightarrow alt G (s[ x \sim (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:

```plaintext
Theorem ChannelReplacement G P x kp D:
  def (subst_env_ch x (ce kp) D) \rightarrow
  oft G P D \rightarrow oft G (s[ x \sim (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:

```plaintext
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 $\Theta; \Gamma \vdash P[k^p/x] \triangleright \Delta \cdot k^p : \alpha$.

**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 => of G (s[ x ~ (ch kp) ]p P) ((sub...)
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 $\Delta$ balanced and $P \rightarrow^* Q$, then $\Theta; \Gamma \vdash Q \triangleright \Delta'$ and $\Delta'$ balanced.
Subject Reduction

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

Is straightforward to represent:

```
Theorem SubjectReductionStep G P Q D:
  oft G P D \rightarrow balanced D \rightarrow P \rightarrow Q \rightarrow exists D', balanced D' \land 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'** \( G \ P \ Q \ D \ D' \ ka: \)
\[
\text{oft } G \ P \ D \rightarrow \text{balanced } D \rightarrow P \rightarrow Q \rightarrow D \rightarrow Q \rightarrow D' \rightarrow \text{oft } G \ Q \ D'.
\]
\((* \ ... \ *)\)

**Lemma admissible_label** \( P \ Q: \)
\[
P \rightarrow Q \rightarrow \text{exists } ka, P \rightarrow Q.
\]
\((* \ ... \ *)\)

**Lemma well_typed_step** \( G \ P \ Q \ D \ ka: \)
\[
\text{oft } G \ P \ D \rightarrow P \rightarrow Q \rightarrow \text{exists } D', D \rightarrow D' \rightarrow \text{exists } D'.
\]
\((* \ ... \ *)\)

**Lemma typ_step_preserves_balance** \( D \ D' \ ka: \)
\[
D \rightarrow \text{balanced } D \rightarrow \text{balanced } D'.
\]
\((* \ ... \ *)\)
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:
\[
oft \ G \ P \ D \to \text{balanced} \ D \to \ P \quad \text{--- ka} \quad \text{---} \to \ Q \to \ D \quad \text{~~~ ka} \quad \text{~~~} \to \ D' \to \ oft \ G \ Q \ D'.
\]
\[(\ast \ \ldots \ \ast)\]

**Lemma admissible_label** $P \ Q$:
\[
P \to Q \to \text{exists} \ ka, \ P \quad \text{--- ka} \quad \text{---} \to Q.
\]
\[(\ast \ \ldots \ \ast)\]

**Lemma well_typed_step** $G \ P \ Q \ D \ ka$:
\[
oft \ G \ P \ D \to P \quad \text{--- ka} \quad \text{---} \to Q \to \text{exists} \ D', \ D \quad \text{~~~ ka} \quad \text{~~~} \to D'.
\]
\[(\ast \ \ldots \ \ast)\]

**Lemma typ_step_preserves_balance** $D \ D' \ ka$:
\[
D \quad \text{~~~ ka} \quad \text{~~~} \to D' \to \text{balanced} \ D \to \text{balanced} \ D'.
\]
\[(\ast \ \ldots \ \ast)\]
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 \text{SubjectReductionStep'} \ G \ P \ Q \ D \ D' \ ka:
\quad \text{oft } G \ P \ D \rightarrow \text{balanced } D \rightarrow P \quad \text{ka} \quad \text{----> } Q \rightarrow D \quad \text{~~~ ka} \quad \text{~~~> } D' \rightarrow \text{oft } G \ Q \ D'.
\quad \text{(* ... *)}

Lemma \text{admissible\_label} \ P \ Q:
\quad P \rightarrow Q \rightarrow \text{exists } \text{ka}, P \quad \text{ka} \quad \text{----> } Q.
\quad \text{(* ... *)}

Lemma \text{well\_typed\_step} \ G \ P \ Q \ D \ ka:
\quad \text{oft } G \ P \ D \rightarrow P \quad \text{ka} \quad \text{----> } Q \rightarrow \text{exists } D', D' \quad \text{~~~ ka} \quad \text{~~~> } D'.
\quad \text{(* ... *)}

Lemma \text{typ\_step\_preserves\_balance} \ D \ D' \ ka:
\quad D \quad \text{~~~ ka} \quad \text{~~~> } D' \rightarrow \text{balanced } D \rightarrow \text{balanced } D'.
\quad \text{(* ... *)}
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 \( G \ P \ Q \ D: \)
\[
\text{oft } G \ P \ D \rightarrow \text{balanced } D \rightarrow P \rightarrow Q \rightarrow \text{exists } D', \text{balanced } D' \setminus \text{oft } G \ Q \ D'.
\]

Proof.

- move\( \Rightarrow \)\( H_p \ H_b \ H_s. \)
- apply admissible_label in \( H_s. \)
- destruct \( H_s. \)
- have \( H_H := \text{well_typed_step } H_p \ H. \)
- destruct \( H_H. \)
- exists \( x_0. \)
- split.
- apply: typ_step_preserves_balance ; [apply: \( H_0 \) | apply: \( H_b \)].
- apply: SubjectReductionStep';
  [apply: \( H_p \) | apply: \( H_b \) | apply: \( H \) | apply: \( H_0 \)].

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 systems, one sound proved with a counter example, and another 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 simplify proofs.
- More automation using overloaded-lemmas in the future.

Thanks for your attention. Questions?