Mechanizing Session-Types: Enforcing linearity without linearity

Brigitte Pientka
McGill University
Montreal, Canada

joint work with
Chuta Sano and Ryan Kavanagh
Mechanizing formal systems together with proofs establishes trust.
Mechanizing formal systems together with proofs establishes trust... and avoids flaws.
Compiler bugs

- are very hard to find and to fix
- every programmer and every program is affected and potentially at risk
  \[\Rightarrow\] over 195 bugs in GCC and LLVM C Compiler [Vu et al. PLDI'14]
Compilers Go Wrong

Compiler bugs

- are very hard to find and to fix
- every programmer and every program is affected and potentially at risk
  \[\Rightarrow\] over 195 bugs in GCC and LLVM C Compiler [Vu et al. PLDI’14]

Grand Challenge: Verifying compiler challenge (T. Hoare)

1970s Challenge proposed, but abandoned
2003 Challenge renewed
Compilers Go Wrong

Compiler bugs

- are very hard to find and to fix
- every programmer and every program is affected and potentially at risk

⇒ [Vu et al. PLDI’14] found over 195 bugs in GCC and LLVM C Compiler

Grand Challenge: Verifying compiler challenge (T. Hoare)

1970s  •  Challenge proposed, but abandoned
2003  •  Challenge renewed
2009  •  CompCert Compiler [X. Leroy]
Mechanized Metatheory for the Masses: The POPLMark Challenge

Brian E. Aydemir¹, Aaron Bohannon¹, Matthew Fairbairn², J. Nathan Foster¹, Benjamin C. Pierce¹, Peter Sewell², Dimitrios Vytiniotis¹, Geoffrey Washburn¹, Stephanie Weirich¹, and Steve Zdancewic¹

¹ Department of Computer and Information Science, University of Pennsylvania
² Computer Laboratory, University of Cambridge

Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

SOFTWARE FOUNDATIONS

The Software Foundations series is a broad introduction to the mathematical underpinnings of reliable software.

The principal novelty of the series is that every detail is one hundred percent formalized and machine-checked: the entire text of each volume, including the exercises, is entirely "proof script" for the Coq proof assistant.

The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers.

No specific background in logic or programming languages is assumed, though a degree of mathematical maturity is helpful.

A one-semester course can present the entire Software Foundations. Each volume of Software Foundations is deeply functional algorithms, or selections from books.

Part 1: Logical Foundations

- Naturals: Natural numbers
- Induction: Proof by induction
- Relations: Inductive definition of relations
- Equality: Equality and equational reasoning
- Isomorphism: Isomorphism and embedding
- Connectives: Conjunction, disjunction, and implication
- Negation: Negation, with intuitionistic and classical logic
- Quantifiers: Universals and existentials
- Decidable: Decidability and decision procedures
But Mechanization of Process Calculi Remain Very Difficult
But Mechanization of Process Calculi Remain Very Difficult

1995


1997

D. Hirschkoff: A full formalisation of $\pi$-calculus theory in the calculus of constructions. TPHOL’97

2001

Christine Röckl, Daniel Hirschkoff, Stefan Berghofer: Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the pi-Calculus and Mechanizing the Theory of Contexts, Fossacs’01
“approaches in the literature to the implementation of the π-calculus had adopted either a direct first-order encoding or had dropped names tout court in favour of de Brujin indexes. The user is hence overwhelmed by technical details and lemmata about equivalence free names operators, substitution functions and so ... out 600 of 800 proved lemmata concern the technical details of index handling.”

F. Honsell, M. Miculan, I. Scagnetto, Pi-calculus in (Co)Inductive Type Theory, Theoretical Computer Science, Vol 253, 2001
Session Types in a Nutshell

- Message passing calculus (like the $\pi$-calculus) with a typing discipline for structured interactions

"Session types are applied to a wide range of problems, and their properties, such as deadlock-freedom, are well studied. These calculi are very expressive, and rather complex, with features like: shared and linear communication channels, name passing, and fresh name generation. Given this complexity, it is not surprising that some innocent looking extensions violated the type safety properties of the calculus in several literature (as pointed out by [23])."

[Castro-Perez, Ferreira, and Yoshida'20]
Recent Efforts on Mechanizing Session Types

- Functional language with session-typed communication [Thiemann’19] de Bruijn

- Type-checking session-typed pi-calculus with Coq [Zalakain’19] combination of linearity/wf checks and explicit contexts

- Π with left-overs: A Mechanization in Agda. [Zalakain and Dardha’21] explicit contexts with left-over constraints

- General framework of mechanizing session-typed process calculi in Coq. [Castro-Perez, Ferreira, and Yoshida’20] locally nameless

- Concurrent Calculi Formalisation Benchmark [Ferreira, et. al. 2023]

It’s been difficult to encode session type systems . . . . The fact that there are different formulations (GV, Multi-Party Session Types, etc.) isn’t helping.
A Logical View of Session Types
“A logic without weakening or contraction” [Girard 87]

\[ \vdash A_1, \ldots, A_n \]

\[
A, B ::= A \otimes B \quad \text{(Tensor; multiplicative conjunction)}
\]

\[
| A \& B \quad \text{(With/and; additive conjunction)}
\]

\[
| A \bowtie B \quad \text{(Par; multiplicative disjunction)}
\]

\[
| A \oplus B \quad \text{(Plus; additive disjunction)}
\]

\[
| 1 \quad \text{(Unit of } \otimes)\]

\[
| \perp \quad \text{(Unit of } \bowtie)\]

Linear negation \( A \perp \)
| Linear Logic | Message Passing Concurrency |
### Linear Propositions as Session Types

**Linear Logic**
- Assumptions
- Linear Propositions
- Sequents
- Proofs
- Cut
- Cut elimination

**Message Passing Concurrency**
- Channels
- Session types
- Process Typing
- Well-Typed Processes
- Parallel Composition
- Communication

[Caires, Pfenning’10][Wadler’12]
Identity and Composition

\[
\frac{\text{fwd} \ x \ y \vdash x : A, y : A^\perp}{\vdash x : A, y : A^\perp} \quad (\text{ID})
\]

\[
\frac{P \vdash \Delta_1, x : A \quad Q \vdash \Delta_2, x : A^\perp}{\nu x : A . (P \parallel Q) \vdash \Delta_1, \Delta_2} \quad (\text{CUT})
\]

- Identity – Forwarding future communication between channels
- Composition – spawns processes \( P \) and \( Q \) that communicate along a bound private channel \( x \).

**Challenge**

- Variable binding
- Splitting context and ensuring channel that no other channels are shared
Internal and External Choice

\[ P \vdash \Delta, x : A \]
\[ \text{inl}^x \quad P \vdash \Delta, x : A \oplus B \] (⊕₁)

\[ P \vdash \Delta, x : B \]
\[ \text{inr}^x \quad P \vdash \Delta, x : A \oplus B \] (⊕₂)

\[ P \vdash \Delta, x : A \]
\[ Q \vdash \Delta, x : B \]
\[ \text{case } x (P, Q) \vdash \Delta, x : A \& B \] (&)

- Send a "left" or "right" choice over \( x \) and continue with \( P \)
- Based on the received choice choose process \( P \) or \( Q \)

Challenges

- "Update" the type associated with the channel
Type Judgements - Classical Processes (CP) [Wadler’12]

Channel Transmission

Duality of (Multiplicative) Par and Tensor

\[(A \otimes B)^\perp = A^\perp \otimes B^\perp\]

\[
\frac{P \vdash \Delta_1, y : A}{\text{out } x y; (P \parallel Q) \vdash \Delta_1, \Delta_2, x : A \otimes B}
\]

\[
\frac{Q \vdash \Delta_2, x : B}{(\otimes) \quad P \vdash \Delta, x : B, y : A}
\]

\[
\frac{\text{inp } x y; P \vdash \Delta, x : A \otimes B}{(\otimes)}
\]

- **Out**: Sends a channel name \(y\) across the channel \(x\), and spawns concurrent processes \(P\) and \(Q\) that provide channel \(y\) and \(x\)
- **In**: Receives a channel over \(x\), binds it to a fresh name \(y\), and proceeds as \(P\).

Challenges

- "Update" the type associated with the channel \(x\)
- Variable binding
- Splitting context to ensure channels are not shared between \(P\) and \(Q\)
Termination

\[
\begin{align*}
\text{close } x & \vdash x : 1 \quad (1) \\
\text{wait } x; P & \vdash \Delta, x : \bot \quad (\bot)
\end{align*}
\]

- Termination and Waiting for termination

Challenges

- Ensure that there are no other left-over channels
- Consume channel \( x \)
Reductions and Commuting Conversions

Reduction and Principal Reduction:

\[
\nu x : A. (\text{fwd} \times y \parallel Q) \Rightarrow_{CP} \left[ y / x \right] Q \quad (\beta_{\text{FWD}})
\]

\[
\nu x : A \oplus B. (\text{inl}^x \, P' \parallel \text{case} \times (Q_1, Q_2)) \Rightarrow_{CP} \nu x : A. (P' \parallel Q_1) \quad (\beta_{\text{INL}})
\]

Commuting Conversion:

\[
\nu z : C. (\text{inl}^x \, P' \parallel Q) \Rightarrow_{CP} \text{inl}^x \nu z : C. (P' \parallel Q) \quad (\kappa_{\text{INL}})
\]

Congruence Rules including Associativity and Communicativity of parallel process composition
Theorem: Type Preservation

If \( P \vdash \Delta \) and \( P \Rightarrow_{CP} Q \), then \( Q \vdash \Delta \).
Two main challenges

• How to deal with contexts?
• How to deal with channel names?

They are somewhat related ...
Goal: Leverage HOAS

- Leverage binding infrastructure for modelling channel dependencies in processes (inherit $\alpha$-renaming, substitution operation, easy checking for variable dependencies, etc.)

- Leverage (ambient) context and hypothetical judgments (inherit substitution property, context management including weakening, contraction, uniqueness)

Problem

We have a linear context and we re-use channel names
Goal: Leverage HOAS

- Leverage binding infrastructure for modelling channel dependencies in processes (inherit $\alpha$-renaming, substitution operation, easy checking for variable dependencies, etc.)

- Leverage (ambient) context and hypothetical judgments (inherit substitution property, context management including weakening, contraction, uniqueness)

Solution

1. Linear Context:
   - Adapt idea from Crary [ICFP’12] to session types
   - Define and use linearity predicate to ensure a channel name occurs only once
   - If a channel name occurs once, any assumption involving it can only be used once

2. Continuation Channels to avoid re-using channel names
“You have to listen to the logical framework, as it were, and take its advice in guiding you towards a better way to formulate your system. We learned this lesson many years ago when we first invented LF — the exercise of formalizing a logic in LF does wonders for the logic.”

Bob Harper’s post to the POPLmark-list 2 May 2006
A Structural Process Calculus (SCP)

[Sano, Kavanagh, Pientka’23]
Making the continuation channel explicit in SCP

Previous rule for $A \oplus B$ (Plus; additive disjunction)

$$
P \vdash \Delta, x : A \quad (\oplus_1) \quad P \vdash \Delta, x : B \quad (\oplus_2)
$$

Restating $\text{inl}^x P$ and $\text{inr}^x P$ with an explicit continuation channel $w$

$$
P \vdash \Gamma, x : A \oplus B, w : A \quad [\oplus_1] \quad P \vdash \Gamma, x : A \oplus B, w : B \quad [\oplus_2]
$$
Making the continuation channel explicit in SCP

Previous rule for $A \oplus B$ (Plus; additive disjunction)

\[
P \vdash \Delta, x : A \\
inl^x P \vdash \Delta, x : A \oplus B \quad (\oplus_1) \\
inr^x P \vdash \Delta, x : A \oplus B \quad (\oplus_2)
\]

Restating \( \text{inl}^x P \) and \( \text{inr}^x P \) with an explicit continuation channel \( w \)

\[
P \vdash \Gamma, x : A \oplus B, w : A \\
\text{inl } x; \ w.P \vdash \Gamma, x : A \oplus B \quad [\oplus_1] \\
P \vdash \Gamma, x : A \oplus B, w : B \\
\text{inr } x; \ w.P \vdash \Gamma, x : A \oplus B \quad [\oplus_2]
\]

Using explicit continuation channel also for case \( x (P, Q) \)

\[
P \vdash \Gamma, x : A \& B, w : A \\
Q \vdash \Gamma, x : A \& B, w : B \\
\text{case } x (w.P, \ w.Q) \vdash \Gamma, x : A \& B \quad [\&]
\]
Ensuring internal bindings are linear in SCP

**Idea:** Check linearity for fresh channels

\[
\frac{P \parallel \Gamma, x : A \quad \text{lin}(x, P) \quad Q \parallel \Gamma, x : A^\perp \quad \text{lin}(x, Q)}{\nu x: A. (P \parallel Q) \parallel \Gamma} \quad \text{[Cut]}
\]
Ensuring internal bindings are linear in SCP

Idea: Check linearity for fresh channels

\[
\frac{P \parallel \Gamma, x : A \quad \text{lin}(x, P) 
\quad Q \parallel \Gamma, x : A^\perp \quad \text{lin}(x, Q)}{
u x : A. (P \parallel Q) \parallel \Gamma} \quad \text{[Cut]}
\]

How does this work for when we reach a channel name?

\[
\frac{\text{close } x \parallel \Gamma, x : 1 \quad [1]}{\text{wait } x; P \parallel \Gamma, x : \perp \quad [\perp]}
\]
Localize Linearity - Linearity Predicate

\( fn(P) \) - “The set of free channel names in \( P \).”

\( \text{lin}(x, P) \) - “Channel \( x \) and its continuations are used linearly in \( P \).”
Localize Linearity - Linearity Predicate

$\text{fn}(P)$ - “The set of free channel names in $P$.”

$\text{lin}(x, P)$ - “Channel $x$ and its continuations are used linearly in $P$.”

\[
\begin{align*}
\text{lin}(x, \text{close } x) & \quad \text{Linl} \\
\text{lin}(x, \text{close } x) & \quad \text{L}_\text{close} \\
\text{lin}(w, P) \quad x \notin \text{fn}(P) & \quad \text{Linl} \\
\text{lin}(x, \text{inl } x; w.P) & \quad \text{Linl}
\end{align*}
\]
Localize Linearity - Linearity Predicate

\[ \text{fn}(P) - \text{“The set of free channel names in } P.\text{”} \]
\[ \text{lin}(x, P) - \text{“Channel } x \text{ and its continuations are used linearly in } P.\text{”} \]

\[
\frac{\text{lin}(x, \text{close } x)}{L_{\text{close}}} \quad \frac{\text{lin}(w, P) \ x \notin \text{fn}(P)}{L_{\text{inl}}}
\]

\[
\frac{\text{lin}(z, P) \ z \notin \text{fn}(Q)}{L_{\nu_1}} \quad \frac{\text{lin}(z, P) \ z \notin \text{fn}(P)}{L_{\nu_2}}
\]

Note: We can check \( x \notin \text{fn}(P) \) exploiting the powers of higher-order unification in HOAS systems where we check and encode variable dependencies.
**Localize Linearity - Linearity Predicate**

\[\text{fn}(P) - \text{"The set of free channel names in } P\text{."}\]
\[\text{lin}(x, P) - \text{"Channel } x \text{ and its continuations are used linearly in } P\text{."}\]

\[
\begin{align*}
\text{lin}(x, \text{close } x) & \quad L_{\text{close}} \\
\text{lin}(w, P) & \quad x \not\in \text{fn}(P) \quad L_{\text{lin}}
\end{align*}
\]

\[
\begin{align*}
\text{lin}(z, P) & \quad z \not\in \text{fn}(Q) \quad L_{\nu_1} \\
\text{lin}(z, \nu x:A.(P \parallel Q)) & \quad \text{lin}(z, \nu x:A.(P \parallel Q)) \quad L_{\nu_2}
\end{align*}
\]

Note: We can check \( x \not\in \text{fn}(P) \) exploiting the powers of higher-order unification in HOAS systems where we check and encode variable dependencies.
Key Principle to Allow for Extensions

• **Does the construct bind any new linear channels?**
  If yes, then the typing judgment must check their linearity.

• **Does the construct requires the absence of other linear assumptions?**
  If yes, then there should be no congruence rules for the linearity predicate.

• **Does the construct use a continuation channel?**
  If yes, then the linearity predicate should check that the continuation channel is used linearly. Otherwise, the linearity predicate should be an axiom.

• **Are linear channels shared between subterms composed by the construct?**
  If they are not shared, then the linearity predicate must ensure that no sharing occurs.
Encoding SCP in Beluga
The Tip of the Iceberg: Beluga [POPL’08, IJCAR’10, POPL’12, CADE’15, ICFP’16, ...]

Proofs as Functional Programs

Main Proof

Terms $t ::= [\Psi \vdash M] \mid \ldots$

Types $T ::= [\Psi \vdash A] \mid \ldots$

Contextual Logical Framework LF

Renaming Scope Binding Hypothesis Variables
Substitution Context Eigenvariables
Derivation Tree

$\Gamma \vdash t : T$

$\Gamma; \Psi \vdash M : A$

PPDP Test of Time Award in 2018
Encoding Processes using (weak) HOAS

Processes \( P, Q \) ::= \( \text{fwd} \ x \ y \mid \text{close} \ x \mid \text{wait} \ x; P \mid \nu x:A.(P \parallel Q) \mid \text{inl} \ x; w.P \mid \text{inr} \ x; w.P \mid \text{case} \ x \ (w.P, w.Q) \mid \text{out} \ x; (y.P \parallel w.Q) \mid \text{inp} \ x \ (w.y.P) \)

**LF Representation**

| f\text{wd} | : name \rightarrow name \rightarrow proc. |
| close | : name \rightarrow proc. |
| wait | : name \rightarrow proc \rightarrow proc. |
| p\text{comp} | : tp \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow \text{proc}. |
| in\text{l} | : name \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow \text{proc.} |
| in\text{r} | : name \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow \text{proc.} |
| case | : name \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow \text{proc.} |
| inp | : name \rightarrow (\text{name} \rightarrow \text{name} \rightarrow \text{proc}) \rightarrow \text{proc.} |
| out | : name \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow (\text{name} \rightarrow \text{proc}) \rightarrow \text{proc.} |
Representing Session Typing in LF

\[
\text{wtp: proc} \rightarrow \text{type}
\]

\[
\text{wtp}_\text{fwd} : \text{dual} \ A \ A' \\
\rightarrow \{X:\text{name}\}\text{hyp} \ X \ A \\
\rightarrow \{Y:\text{name}\}\text{hyp} \ Y \ A' \\
\rightarrow \text{wtp} \ (\text{fwd} \ X \ Y).
\]

\[
\text{wtp}_\text{close} : \{X:\text{name}\}\text{hyp} \ X \ 1 \\
\rightarrow \text{wtp} \ (\text{close} \ X).
\]

\[
\text{wtp}_\text{pcomp} : \text{dual} \ A \ A' \\
\rightarrow (\{x:\text{name}\} \ \text{hyp} \ x \ A \rightarrow \text{wtp} \ (P \ x)) \\
\rightarrow (\{x:\text{name}\} \ \text{hyp} \ x \ A' \rightarrow \text{wtp} \ (Q \ x)) \\
\rightarrow \text{linear} \ P \rightarrow \text{linear} \ Q \\
\rightarrow \text{wtp} \ (\text{pcomp} \ A \ P \ Q).
\]

\[
P \vdash x_1 : A_1, \ldots, x_n : A_n
\]

\[
\text{fwd} \ x \ y \vdash \Gamma, x : A, y : A' \perp \quad [\text{Id}]
\]

\[
\text{close} \ x \vdash \Gamma, x : 1 \quad [1]
\]

\[
P \vdash \Gamma, x : A \quad \text{lin}(x, P) \quad Q \vdash \Gamma, x : A' \perp \quad \text{lin}(x, Q) \quad \nu x : A. (P \parallel Q) \vdash \Gamma \quad [\text{Cut}]
\]
Encoding the Linearity Predicate

\[
\text{linear} : (\text{name} \rightarrow \text{proc}) \rightarrow \text{type}.
\]

\[
\text{l\_close} : \text{linear} (\lambda x. \text{close} x).
\]

\[
\text{l\_wait} : \text{linear} (\lambda x. \text{wait} x P).
\]

\[
\text{l\_inl} : \text{linear} P
\]
\[
\rightarrow \text{linear} (\lambda x. \text{inl} x P).
\]

\[
\text{l\_out} : \text{linear} Q
\]
\[
\rightarrow \text{linear} (\lambda x. \text{out} x P Q).
\]

\[
\begin{align*}
\text{lin}(x, P) & \quad L_{\text{close}} \\
\text{lin}(x, \text{close} x) & \\
\hline
\text{lin}(x, \text{wait} x; P) & L_{\text{wait}} \\
x \notin \text{fn}(P) & \\
\text{lin}(w, P) \quad x \notin \text{fn}(P) & L_{\text{inl}} \\
\text{lin}(x, \text{inl} x; w.P) & \\
x \notin \text{fn}(P) \cup \text{fn}(Q) & \\
\text{lin}(w, Q) \quad x \notin \text{fn}(P) \cup \text{fn}(Q) & L_{\text{out}} \\
\text{lin}(x, \text{out} x; (y.P \parallel w.Q)) & \\
\end{align*}
\]
For example:

\[
\nu x : A \oplus B . (\text{inl} \ x; \ w . P \parallel \text{case} \ x \ (w . Q, \ w . R)) \Rightarrow_{SCP} \nu w : A . (P \parallel Q)
\]  

is translated to:

\[
\beta_{\text{inl}} : \text{step} \ (p\text{comp} \ (A \oplus B) \ (\lambda x . \text{inl} \ x \ P) \ (\lambda x . \text{case} \ x \ Q \ R)) \ (p\text{comp} \ A \ P \ Q).
\]
Implementing Type Preservation
The Tip of the Iceberg: Beluga [POPL’08, IJCAR’10, POPL’12, CADE’15, ICFP’16, ...]

Proofs as Functional Programs

Terms $t ::= [\Psi \vdash M] | \ldots$

Types $T ::= [\Psi \vdash A] | \ldots$

Contextual Logical Framework LF

Renaming Scope Binding
Hypothesis Variables
Substitution Context
Eigenvariables
Derivation Tree

$\Gamma \vdash t : T$

$\Gamma; \Psi \vdash M : A$

PPDP Test of Time Award in 2018
Main Property – Revisited in Beluga

**Theorem: Type Preservation of CP**

If $P \vdash \Delta$ and $P \Rightarrow_{CP} Q$, then $Q \vdash \Delta$.

\[
\text{rec wtp.s : (Γ : ctx) } [Γ \vdash \text{wtp } P ] \rightarrow [Γ \vdash \text{step } P Q ] \rightarrow [Γ \vdash \text{wtp } Q ]
\]

and

\[
\text{rec lin.s : (Γ : ctx) } [Γ, x:\text{name}, h: \text{hyp } x A[] \vdash \text{wtp } P[..,x] ] \rightarrow [Γ \vdash \text{linear } (\lambda x. P[..,x])] \\
\rightarrow [Γ, x:\text{name} \vdash \text{step } P[..,x] Q[..,x] ] \\
\rightarrow [Γ \vdash \text{linear } (\lambda x. Q[..,x])]
\]

- Uses contextual objects
- Abstract over the set of channel names and assumptions about channels
Summary of SCP

- Unrestricted context and linearity predicate precisely characterize linearity.
- Continuation channels make channel dependencies explicit.
- Equivalence between CP and SCP
Summary of SCP

- Unrestricted context and linearity predicate precisely characterize linearity.
- Continuation channels make channel dependencies explicit.
- Equivalence between CP and SCP
Summary of SCP

- Unrestricted context and linearity predicate precisely characterize linearity.
- Continuation channels make channel dependencies explicit.
- Equivalence between CP and SCP

\[
\text{lin}(x, P) \quad \text{for all } x \in \Gamma
\]
Mechanizations of Sub-Structural Systems in Beluga

- SCP [C. Sano, R. Kavanagh]
  Type Preservation
  About 10 lemmas, in particular strengthening lemmas that would be difficult to state in a system like Twelf; (1+1)x2 main theorems for reductions and equivalences

- CP with explicit contexts and continuation channels [D. Zackon]
  Equivalence to SCP
  Type Preservation
  approx. 90 lemmas just about context management in CP

- CP with explicit contexts and updating of channel names [D. Zackon]
  Equivalence of between two formulations of CP
What’s next?
**What’s next?**

**ToDo 1:** Exploring how reusable the infrastructure for explicitly managing contexts is in other sub-structural settings

**ToDo 2:** Mechanize deadlock freedom for SCP (and CP) (Seems relatively straightforward)

**ToDo 3:** Add more type constructors such as exponentials !A and ?A to the mechanization

**ToDo 4:** Mechanize other theorems about session types

**ToDo 5:** Mechanize other session type systems [Vasconcelos’12] and Concurrent Calculi Formalisation Benchmark [Ferreira, et. al. 2023]
Lesson 1: Explicit linearity predicate is surprisingly effective.

Lesson 2: HOAS is a surprisingly effective way to re-think and encode sub-structural systems (channel name binding, context management, free variable condition checks).

Lesson 3: Apply these ideas to other sub-structural systems such as for example Quiper (Quantum Programming Language)

Lesson 4: The approach should also scale to systems such as Coq where we can take advantage of existing infrastructure for contexts and variable bindings for intuitionistic systems (i.e. re-use the Auto-Subst library; still need to implement free variable checks)

Lesson 5: Most session types are simply typed – can we provide stronger static guarantees about processes? (ongoing work with Ryan Kavanagh)