Mechanizing Session-Types using a Structural View: Enforcing Linearity without Linearity

CHUTA SANO, McGill University, Canada
RYAN KAVANAGH, McGill University, Canada
BRIGITTE PIENTKA, McGill University, Canada

Session types employ a linear type system that ensures that communication channels cannot be implicitly copied or discarded. As a result, many mechanizations of these systems require modeling channel contexts and carefully ensuring that they treat channels linearly. We demonstrate a technique that localizes linearity conditions as additional predicates embedded within type judgments, which allows us to use structural typing contexts instead of linear ones. This technique is especially relevant when leveraging (weak) higher-order abstract syntax to handle channel mobility and the intricate binding structures that arise in session-typed systems.

Following this approach, we mechanize a session-typed system based on classical linear logic and its type preservation proof in the proof assistant Beluga, which uses the logical framework LF as its encoding language. We also prove adequacy for our encoding. This shows the tractability and effectiveness of our approach in modelling substructural systems such as session-typed languages.

CCS Concepts: • **Theory of computation → Logic and verification**; *Process calculi.*

Additional Key Words and Phrases: linear logic, concurrency, session types, verification, logical framework

ACM Reference Format:
Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. 2023. Mechanizing Session-Types using a Structural View: Enforcing Linearity without Linearity. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 235 (October 2023), 26 pages. https://doi.org/10.1145/3622810

1 INTRODUCTION

The π-calculus [Milner 1980] is a well-studied formalism for message passing concurrency. Although there have been many efforts to mechanize variants of the π-calculus by encoding their syntax and semantics in proof assistants, mechanization remains an art. For example, process calculi often feature rich binding structures and semantics such as channel mobility, and these must be carefully encoded to respect α-equivalence and to avoid channel name clashes.

Even harder to mechanize are session-typed process calculi, in part because they treat communications channels linearly. *Session types* [Honda 1993; Honda et al. 1998] specify interactions on named communication channels, and linearity ensures that communication channels are not duplicated or discarded. As a result, session types can be used to statically ensure safety properties such as session fidelity or deadlock freedom. However, mechanizing linear type systems adds another layer of complexity; most encodings of linear type systems encode contexts explicitly: they

Authors' addresses: Chuta Sano, School of Computer Science, McGill University, 3480 rue University, Montréal, QC, H3A 0E9, Canada, chuta.sano@mail.mcgill.ca; Ryan Kavanagh, School of Computer Science, McGill University, 3480 rue University, Montréal, QC, H3A 0E9, Canada, rkavanagh@cs.mcgill.ca; Brigitte Pientka, School of Computer Science, McGill University, 3480 rue University, Montréal, QC, H3A 0E9, Canada, bpientka@cs.mcgill.ca.

2475-1421/2023/10-ART235
https://doi.org/10.1145/3622810

develop some internal representation of a collection of channels, for example, a list, implement relevant operations on it, and then prove lemmas such as α-equivalence and substitution. Though explicit encodings have led to successful mechanizations [Castro-Perez et al. 2020; Jacobs et al. 2022; Thiemann 2019; Zalakain and Dardha 2021], they make it cumbersome to formalize metatheoretic results like subject reduction.

Higher-order abstract syntax [Pfenning and Elliott 1988] (HOAS) relieves us from the bureaucracy of explicitly encoded contexts. With this approach, variable abstractions are identified with functions in the proof assistant or the host language. Thus, we can obtain properties of bindings in the host language for free, such as the aforementioned α-equivalence and substitution lemmas. This technique had been studied in process calculi without modern linear session types by Röckl, Hirschkoff, and Berghofer [Röckl et al. 2001] in Isabelle/HOL and by Despeyroux [Despeyroux 2000] in Coq. However, HOAS has rarely been used to encode linear systems, and it has not yet been applied to mechanize session-typed languages. This is because most HOAS systems treat contexts structurally while session-typed systems require linear contexts. Consequently, naively using HOAS to manage channel contexts would not guarantee that channels are treated linearly. This would in turn make it difficult or impossible to prove metatheoretic properties that rely on linearity, such as deadlock freedom.

In our paper, we develop a technique to bridge the gap between structural and linear contexts. We use this technique to mechanize a subset of Wadler's Classical Processes (CP) [Wadler 2012]. CP is a well-studied foundation for investigating the core ideas of concurrency due to its tight relation with linear logic. For our mechanization, we first introduce *Structural Classical Processes* (SCP), a system whose context is structural. This calculus encodes linearity using a technique heavily inspired by the one Crary [2010] used to give a HOAS encoding of the linear λ-calculus. The key idea is to define a predicate

$$\mathsf{lin}(x,\ P)$$

for some process P that uses a channel x. This predicate can informally be read as "channel x is used linearly in P," and it serves as a localized well-formedness predicate on the processes. We embed these additional proof obligations within type judgments for rules that introduce channel bindings. Thus, well-typed processes use all of their internally bound names linearly, and we further give a bijection between CP and SCP typing derivations to show that these linearity predicates precisely capture the notion of linear contexts.

We then mechanize SCP in Beluga [Pientka and Dunfield 2010] using weak HOAS. The mechanization is mostly straightforward due to the strong affinity SCP has with LF, and we prove adequacy of our encoding with respect to SCP. This adequacy result is compatible with our prior bijection result between CP and SCP, meaning our encoding is also adequate with respect to CP. Finally, we mechanize type preservation in our encoding in a very elegant manner, taking advantage of the various properties we obtain for free from a HOAS encoding such as renaming, variable dependencies that are enforced via higher-order unification, etc.

Contributions. We describe a structural approach to mechanizing session-types and their metatheory without relying on the substructural properties of the session type system, by using explicit linearity check for processes. In particular:

- We introduce an on-paper system equivalent to a subset of Wadler's Classical Processes (CP) [Wadler 2012], which we call Structural Classical Processes (SCP). This system uses a structural context as opposed to a linear context but still captures the intended properties of linearity using linearity predicates. SCP is well-suited to a HOAS-style encoding as we demonstrate in this paper, but it is also well-suited to other styles of mechanizations given that it does not require any context splits.

- We define a linearity predicate inspired by Crary [2010] for the linear λ-calculus. By doing so, we demonstrate the scalability of Crary's technique to richer settings.
- We encode processes and session types using weak HOAS in the logical framework LF. Our encoding illustrates how we leverage HOAS/LF and its built-in higher-order unification to model channel bindings and hypothetical session type derivations as intuitionistic functions.
- We prove the equivalence of CP and SCP and then show that our encoding of SCP in Beluga is adequate, i.e., that there exist bijections between all aspects of SCP and their encodings. We therefore show that our encoding of SCP is adequate with respect to CP as well. Given that adequacy for session typed systems is quite difficult, we believe that the techniques presented in SCP is a useful baseline for more complex systems.
- We encode and mechanize SCP in Beluga and prove (on paper) that the encoding is adequate. We further mechanize a subject reduction proof of SCP to illustrate how metatheoretic proofs interact with our linearity predicates.

The full mechanization of SCP in Beluga is available as an artifact [**?**].

2 CLASSICAL PROCESSES (CP)

We present a subset of Wadler's Classical Processes (CP), making minor syntactic changes to better align with our later development. CP is a proofs-as-processes interpretation of classical linear logic. It associates to each proof of a classical, linear (one-sided) sequent

$$\vdash A_1, \ldots, A_n$$

a process P that communicates over channels $x_1, \ldots, x_n$:

$$P \vdash x_1 : A_1, \ldots, x_n : A_n.$$

We interpret linear propositions $A_1, \ldots, A_n$ as session types that specify the protocol that P must follow when communicating on channels $x_1, \ldots, x_n$, respectively. Table 1 summarizes the operational interpretation of the standard linear connectives without exponentials and quantifiers:

Type	Action
1	Send a termination signal and then terminate
$\bot$	Receive a termination signal
$A \otimes B$	Send a channel of type A and proceed as B
$A \parr B$	Receive a channel of type A and proceed as B
$A \oplus B$	Send a "left" or "right" and then proceed as A or B accordingly
$A \,\&\, B$	Receive a "left" or "right" and then proceed as A or B accordingly

Table 1. Interpretation of propositions in linear logic as session types on channels in CP

Logical negation induces an involutory notion of duality on session types, where two types are dual if one can be obtained from the other by exchanging sending and receiving. This duality will be used in process composition: we can safely compose a process P communicating on $x : A$ with a process Q communicating on $x : B$ whenever A and B are dual. We write $A^\bot$ for the dual of A; it is inductively defined on the structure of A:

$$1^\bot = \bot \qquad \bot^\bot = 1$$
$$(A \otimes B)^\bot = A^\bot \parr B^\bot \qquad (A \parr B)^\bot = A^\bot \otimes B^\bot$$
$$(A \,\&\, B)^\bot = A^\bot \oplus B^\bot \qquad (A \oplus B)^\bot = A^\bot \,\&\, B^\bot$$

2.1 Type Judgments

Since each inference rule in linear logic corresponds to a process construct, we define the syntax of the processes alongside the type judgments.

Identity and process composition. The identity rule globally identifies two channels x and y. The duality between the types A and $A^\perp$ ensures that this identification only occurs between channels with compatible protocols.

$$\frac{}{\mathsf{fwd}\ x\ y \vdash x : A, y : A^\perp}\ (\textsc{Id})$$

The process composition $\nu x{:}A.(P \parallel Q)$ spawns processes P and Q that communicate along a bound private channel x. Its endpoints in P and Q have type A and $A^\perp$, respectively. Linearity ensures that no other channels are shared between P and Q.

$$\frac{P \vdash \Delta_1, x : A \quad Q \vdash \Delta_2, x : A^\perp}{\nu x{:}A.(P \parallel Q) \vdash \Delta_1, \Delta_2}\ (\textsc{Cut})$$

Channel transmission. The two multiplicative connectives $\otimes$ and $\parr$ correspond to sending and receiving a channel, respectively. The process $\mathsf{out}\ x\ y; (P \parallel Q)$ sends a channel name y across the channel x, and spawns concurrent processes P and Q that provide x and y, respectively.

$$\frac{P \vdash \Delta_1, y : A \quad Q \vdash \Delta_2, x : B}{\mathsf{out}\ x\ y; (P \parallel Q) \vdash \Delta_1, \Delta_2, x : A \otimes B}\ (\otimes)$$

The process $\mathsf{inp}\ x\ y; P$ receives a channel over x, binds it to a fresh name y, and proceeds as P.

$$\frac{P \vdash \Delta, x : B, y : A}{\mathsf{inp}\ x\ y; P \vdash \Delta, x : A \parr B}\ (\parr)$$

Internal and external choice. The two additive connectives $\oplus$ and $\&$ respectively specify internal and external choice. Internal choice is implemented by processes $x[\mathsf{inl}]; P$ and $x[\mathsf{inr}]; P$ that respectively send a "left" and "right" choice across x.

$$\frac{P \vdash \Delta, x : A}{x[\mathsf{inl}]; P \vdash \Delta, x : A \oplus B}\ (\oplus_1) \qquad \frac{P \vdash \Delta, x : B}{x[\mathsf{inr}]; P \vdash \Delta, x : A \oplus B}\ (\oplus_2)$$

External choice is implemented by a case analysis on a received choice:

$$\frac{P \vdash \Delta, x : A \quad Q \vdash \Delta, x : B}{\mathsf{case}\ x\ (P,\ Q) \vdash \Delta, x : A \,\&\, B}\ (\&)$$

Contrary to previous rules, the context Δ in the conclusion is not split between premisses. This does not violate linearity because only one of the branches will be taken.

Termination. The multiplicative units 1 and $\perp$ specify termination and waiting for termination, respectively.

$$\frac{}{\mathsf{close}\ x \vdash x : 1}\ (1) \qquad \frac{P \vdash \Delta}{\mathsf{wait}\ x; P \vdash \Delta, x : \perp}\ (\perp)$$

2.2 Reductions and Type Preservation

Cut elimination in classical linear logic corresponds to reduction rules for CP processes and therefore reduces parallel compositions of form $\nu x{:}A.(P \parallel Q)$. For example, if $P = \mathsf{fwd}\ x\ y$, then we have the reduction rule

$$\frac{}{\nu x{:}A.(\mathsf{fwd}\ x\ y \parallel Q) \Rightarrow_{CP} [y/x]Q}\ (\beta_{\textsc{Fwd}})$$

Other reduction rules are categorized into *principal* reductions, where both P and Q are attempting to communicate over the same channel, *commuting conversions*, where we can push the cut inside

P, and *congruence* rules. We treat all other processes, e.g., $\mathsf{inp}\ x\ y; P$, as stuck processes waiting to communicate with an external agent.

An example of a principal reduction occurs with the composition of $P = x[\mathsf{inl}]; P'$ and $Q = \mathsf{case}\ x\ (Q_1, Q_2)$. After communication, the left process continues as P' and the right process as Q_1, since the "left" signal was sent by P.

$$\frac{}{\nu x{:}A \oplus B.(x[\mathsf{inl}]; P' \parallel \mathsf{case}\ x\ (Q_1, Q_2)) \Rightarrow_{CP} \nu x{:}A.(P' \parallel Q_1)}\ (\beta_{\text{INL}})$$

An example of a commuting conversion occurs when $P = x[\mathsf{inl}]; P'$ and the abstracted channel is some z such that $x \neq z$. In this case, we push the cut inside P.

$$\frac{}{\nu z{:}C.(x[\mathsf{inl}]; P' \parallel Q) \Rightarrow_{CP} x[\mathsf{inl}]; \nu z{:}C.(P' \parallel Q)}\ (\kappa_{\text{INL}})$$

Finally, the congruence rules enable reduction under cuts. We follow Wadler's formulation and do not provide congruence rules for other process constructs. Such rules would eliminate internal cuts and do not correspond to the intended notion of computation, analogously to not permitting reduction under λ-abstractions.

$$\frac{P \Rightarrow_{CP} P'}{\nu x{:}A.(P \parallel Q) \Rightarrow_{CP} \nu x{:}A.(P' \parallel Q)}\ (\beta_{\text{CUT1}}) \qquad \frac{Q \Rightarrow_{CP} Q'}{\nu x{:}A.(P \parallel Q) \Rightarrow_{CP} \nu x{:}A.(P \parallel Q')}\ (\beta_{\text{CUT2}})$$

We close these rules under structural equivalences $P \equiv Q$, which says that parallel composition is commutative and associative:

$$\frac{}{\nu x{:}A.(P \parallel Q) \equiv \nu x{:}A^{\perp}.(Q \parallel P)}\ (\equiv_{\text{COMM}})$$

$$\frac{}{\nu y{:}B.(\nu x{:}A.(P \parallel Q) \parallel R) \equiv \nu x{:}A.(P \parallel \nu y{:}B.(Q \parallel R))}\ (\equiv_{\text{ASSOC}})$$

For ($\equiv_{\text{ASSOC}}$), it is implicit that the process P does not depend on the channel y.

For later developments, we define the closure explicitly as a reduction rule:

$$\frac{P \equiv Q \quad Q \Rightarrow_{CP} R \quad R \equiv S}{P \Rightarrow_{CP} S}\ (\beta_{\equiv})$$

which also requires adding reflexitivity and transitivity to $\equiv$.

Theorem 2.1 (Type Preservation of CP). *If $P \vdash \Delta$ and $P \Rightarrow_{CP} Q$, then $Q \vdash \Delta$.*

3 STRUCTURAL CLASSICAL PROCESSES (SCP)

We introduce *Structural Classical Processes* (SCP). SCP is a reformulation of Classical Processes using a structural context, i.e., in which weakening and contraction hold. The property we would like to enforce is that the context can only grow as we move upwards in a typing derivation. This property makes SCP well-suited for mechanizations and in particular HOAS encodings since no complex operations such as context splitting are necessary. Of course, simply adopting structural rules on top of CP is insufficient because linearity is needed to prove its safety theorems. Instead, we use local linearity predicates to enforce linearity on a global level. These linearity checks are given by a judgment $\mathsf{lin}(x, \mathsf{P})$ that informally means "x occurs linearly in the process P".

SCP's syntax is similar to CP's. In particular, we use the same syntax for session types and the same notion of duality. However, SCP's process syntax explicitly tracks the continuation channels that are left implicit in CP's typing rules (and other on-paper systems). To illustrate, contrast the CP process $x[\mathsf{inl}]; P$ with the corresponding SCP process $\mathtt{inl}\ x;\ w.\mathsf{P}$ and their associated typing rules:

$$\frac{P \vdash \Delta, x : A}{x[\mathsf{inl}]; P \vdash \Delta, x : A \oplus B}\ (\oplus_1) \qquad \frac{\mathsf{P} \Vdash \Gamma, x : A \oplus B, w : A}{\mathtt{inl}\ x;\ w.\mathsf{P} \Vdash \Gamma, x : A \oplus B}\ [\oplus_1]$$

In ($\oplus_1$), the assumption $x : A \oplus B$ in the conclusion is replaced by $x : A$ in the premise, violating our principle that we may only grow contexts. SCP respects the principle thanks to two changes. First, the syntax $\texttt{inl}\ x;\ w.\mathsf{P}$ binds a name w in P for the continuation channel of x. This in turn lets us grow the context in the premise of $[\oplus_1]$ with an assumption $w : A$, while keeping the assumption $x : A \oplus B$. Our linearity predicate ensures that the continuation channel w is used instead of x in P, making these modifications safe. We explain SCP typing judgments below.

SCP is a faithful structural encoding of CP: we give a bijection between well-typed CP processes and well-typed linear SCP processes. Accordingly, we encode SCP instead of CP in LF, and we rely on our equivalence proof to mediate between CP and our LF mechanization of SCP.

3.1 Type Judgments

We write $\mathsf{P} \Vdash \Gamma$ for SCP typing judgments to differentiate them from CP typing judgments $P \vdash \Delta$. The context Γ is structural: it enjoys the weakening, contraction, and exchange properties. Intuitively, it represents the ambient LF context.

Identity and Cut. Axioms use arbitrary contexts Γ to allow for weakening:

$$\frac{}{\texttt{fwd}\ x\ y \Vdash \Gamma, x : A, y : A^{\perp}}\ [\texttt{Id}]$$

We write $vx{:}A.(\mathsf{P} \parallel \mathsf{Q})$ for the composition of P and Q along a private, bound channel x. Contrary to the typing rule (Cut) in CP, the cut rule in SCP does not split contexts. This is because contexts can only grow as we move upwards in SCP typing derivations.

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \quad \mathsf{lin}(x,\ \mathsf{P}) \quad \mathsf{Q} \Vdash \Gamma, x : A^{\perp} \quad \mathsf{lin}(x,\ \mathsf{Q})}{vx{:}A.(\mathsf{P} \parallel \mathsf{Q}) \Vdash \Gamma}\ [\texttt{Cut}]$$

This rule illustrates a general design principle of SCP: we must check that any channel introduced in the continuation of a process is used linearly. In particular, $[\texttt{Cut}]$ checks that P and Q use the free channel x linearly.

Choices. The choice rules explicitly track continuation channels. In particular, the processes $\texttt{inl}\ x;\ w.\mathsf{P}$ and $\texttt{inr}\ x;\ w.\mathsf{P}$ bind the name w in P. This name stands in for the continuation channel of x after it has transmitted a left or right label. The rules $[\oplus_1]$ and $[\oplus_2]$ grow the context and ensure that w has the appropriate type in P. We remark that these two rules do not preclude x and w from both appearing in P. However, this will be ruled out by our linearity predicate, which checks that x and its continuation channels are used linearly in $\texttt{inl}\ x;\ w.\mathsf{P}$ or $\texttt{inr}\ x;\ w.\mathsf{P}$. The treatment of continuation channels in the rule $[\&]$ is analogous.

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \oplus B, w : A}{\texttt{inl}\ x;\ w.\mathsf{P} \Vdash \Gamma, x : A \oplus B}\ [\oplus_1] \qquad \frac{\mathsf{P} \Vdash \Gamma, x : A \oplus B, w : B}{\texttt{inr}\ x;\ w.\mathsf{P} \Vdash \Gamma, x : A \oplus B}\ [\oplus_2]$$

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \,\&\, B, w : A \quad \mathsf{Q} \Vdash \Gamma, x : A \,\&\, B, w : B}{\texttt{case}\ x\ (w.\mathsf{P},\ w.\mathsf{Q}) \Vdash \Gamma, x : A \,\&\, B}\ [\&]$$

Channel Transmission. The channel transmission rules follow the same principles as the identity and cut rules. In particular, they do not split channel contexts between processes, and they check that freshly introduced channels are used linearly. The names y and w are bound in $\texttt{out}\ x;\ (y.\mathsf{P} \parallel w.\mathsf{Q})$

and in $\mathtt{inp}\ x\ (w.y.\mathsf{P})$.

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \otimes B, y : A \quad \mathsf{lin}(y,\ \mathsf{P}) \quad \mathsf{Q} \Vdash \Gamma, x : A \otimes B, w : B}{\mathtt{out}\ x;\ (y.\mathsf{P} \parallel w.\mathsf{Q}) \Vdash \Gamma, x : A \otimes B}\ [\otimes]$$

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \parr B, w : B, y : A \quad \mathsf{lin}(y,\ \mathsf{P})}{\mathtt{inp}\ x\ (w.y.\mathsf{P}) \Vdash \Gamma, x : A \parr B}\ [\parr]$$

Termination. The rules for termination are analogous:

$$\frac{}{\mathtt{close}\ x \Vdash \Gamma, x : 1}\ [1] \qquad \frac{\mathsf{P} \Vdash \Gamma}{\mathtt{wait}\ x; \mathsf{P} \Vdash \Gamma, x : \bot}\ [\bot]$$

3.2 Linearity Predicate

We now define the predicate $\mathsf{lin}(x,\ \mathsf{P})$. It syntactically checks that a free channel x and its continuations occur linearly in P. This judgment is generic relative to an implicit context of channel names that can be freely renamed, and we assume that this implicit context contains the free names $\mathrm{fn}(\mathsf{P})$ of the process P. The linearity predicate $\mathsf{lin}(x,\ \mathsf{P})$ is inductively defined by the following rules, which we informally group into two categories. The first category specifies when a process uses its principal channels linearly. The axioms in this category are:

$$\frac{}{\mathsf{lin}(x,\ \mathtt{fwd}\ x\ y)}\ L_{\mathtt{fwd1}} \quad \frac{}{\mathsf{lin}(y,\ \mathtt{fwd}\ x\ y)}\ L_{\mathtt{fwd2}} \quad \frac{}{\mathsf{lin}(x,\ \mathtt{close}\ x)}\ L_{\mathtt{close}} \quad \frac{x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{wait}\ x;\ \mathsf{P})}\ L_{\mathtt{wait}}$$

For process constructs whose principal channel x would persist in CP, we must check that its continuation channel w is used linearly in its continuation process **and** that the original channel x does not appear in the continuation, thereby capturing the property that w is the continuation of x.

$$\frac{\mathsf{lin}(w,\ \mathsf{Q}) \quad x \notin \mathrm{fn}(\mathsf{P}) \cup \mathrm{fn}(\mathsf{Q})}{\mathsf{lin}(x,\ \mathtt{out}\ x;\ (y.\mathsf{P} \parallel w.\mathsf{Q}))}\ L_{\mathtt{out}} \quad \frac{\mathsf{lin}(w,\ \mathsf{P}) \quad x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{inp}\ x\ (w.y.\mathsf{P}))}\ L_{\mathtt{inp}}$$

$$\frac{\mathsf{lin}(w,\ \mathsf{P}) \quad x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{inl}\ x;\ w.\mathsf{P})}\ L_{\mathtt{inl}} \quad \frac{\mathsf{lin}(w,\ \mathsf{P}) \quad x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{inr}\ x;\ w.\mathsf{P})}\ L_{\mathtt{inr}}$$

$$\frac{\mathsf{lin}(w,\ \mathsf{P}) \quad \mathsf{lin}(w,\ \mathsf{Q}) \quad x \notin \mathrm{fn}(\mathsf{P}) \cup \mathrm{fn}(\mathsf{Q})}{\mathsf{lin}(x,\ \mathtt{case}\ x\ (w.\mathsf{P},\ w.\mathsf{Q}))}\ L_{\mathtt{case}}$$

These rules do not check the linearity of freshly bound channels, for example, of the channel y in channel output or channel input. This is because the predicate only checks the linearity of free channels and their continuations. Although this predicate does not check the linearity of fresh channels such as y, our type system ensures their linear use in well-typed processes.

The second category of rules are congruence cases in which we check the linearity of non-principal channels. We implicitly assume throughout that z is distinct from any bound name:

$$\frac{\mathsf{lin}(z,\ \mathsf{P})}{\mathsf{lin}(z,\ \mathtt{wait}\ x;\ \mathsf{P})}\ L_{\mathtt{wait2}} \quad \frac{\mathsf{lin}(z,\ \mathsf{P}) \quad z \notin \mathrm{fn}(\mathsf{Q})}{\mathsf{lin}(z,\ \mathtt{out}\ x;\ (y.\mathsf{P} \parallel w.\mathsf{Q}))}\ L_{\mathtt{out2}} \quad \frac{\mathsf{lin}(z,\ \mathsf{Q}) \quad z \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(z,\ \mathtt{out}\ x;\ (y.\mathsf{P} \parallel w.\mathsf{Q}))}\ L_{\mathtt{out3}}$$

$$\frac{\mathsf{lin}(z,\ \mathsf{P})}{\mathsf{lin}(z,\ \mathtt{inp}\ x\ (w.y.\mathsf{P}))}\ L_{\mathtt{inp2}} \quad \frac{\mathsf{lin}(z,\ \mathsf{P})}{\mathsf{lin}(z,\ \mathtt{inl}\ x;\ w.\mathsf{P})}\ L_{\mathtt{inl2}} \quad \frac{\mathsf{lin}(z,\ \mathsf{P})}{\mathsf{lin}(z,\ \mathtt{inr}\ x;\ w.\mathsf{P})}\ L_{\mathtt{inr2}}$$

$$\frac{\mathsf{lin}(z,\ \mathsf{P}) \quad \mathsf{lin}(z,\ \mathsf{Q})}{\mathsf{lin}(z,\ \mathtt{case}\ x\ (w.\mathsf{P},\ w.\mathsf{Q}))}\ L_{\mathtt{case2}} \quad \frac{\mathsf{lin}(z,\ \mathsf{P}) \quad z \notin \mathrm{fn}(\mathsf{Q})}{\mathsf{lin}(z,\ \nu x{:}A.(\mathsf{P} \parallel \mathsf{Q}))}\ L_{\nu 1} \quad \frac{\mathsf{lin}(z,\ \mathsf{Q}) \quad z \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(z,\ \nu x{:}A.(\mathsf{P} \parallel \mathsf{Q}))}\ L_{\nu 2}$$

When checking that z appears linearly in processes whose context would be split by the typing rules in CP, namely, in channel output and parallel composition, we ensure that z appears in at

most one of the subprocesses. This lets us use our linearity predicate to mimic context splitting in the presence of structural ambient contexts.

Example 3.1. There exists a well-typed SCP process that is not linear, to wit,

$$\dfrac{\dfrac{\dfrac{}{\texttt{close}\ x \Vdash x : 1, y : \bot}\ [1]}{\texttt{wait}\ y;\ \texttt{close}\ x \Vdash x : 1, y : \bot}\ [\bot]}{\texttt{wait}\ y;\ \texttt{wait}\ y;\ \texttt{close}\ x \Vdash x : 1, y : \bot}\ [\bot]$$

However, it is not the case that $\mathsf{lin}(y,\ \texttt{wait}\ y;\ \texttt{wait}\ y;\ \texttt{close}\ x)$. Indeed, the only rule with a conclusion of this form is $L_{\texttt{wait}}$, but it is subject to the side condition $y \notin \mathrm{fn}(\texttt{wait}\ y;\ \texttt{close}\ x)$.

3.3 Equivalence of CP and SCP

We establish a correspondence between CP and SCP typing derivations. Because CP and SCP use slightly different process syntax, we first define an encoding $\varepsilon(P)$ and a decoding $\delta(\mathsf{P})$ that maps a process in CP to SCP and SCP to CP respectively. We give several representative cases:

$$\begin{aligned}
\varepsilon(\texttt{fwd}\ x\ y) &= \texttt{fwd}\ x\ y & \delta(\texttt{fwd}\ x\ y) &= \texttt{fwd}\ x\ y\\
\varepsilon(\nu x{:}A.(P \parallel Q)) &= \nu x{:}A.(\varepsilon(P) \parallel \varepsilon(Q)) & \delta(\nu x{:}A.(\mathsf{P} \parallel \mathsf{Q})) &= \nu x{:}A.(\delta(\mathsf{P}) \parallel \delta(\mathsf{Q}))\\
\varepsilon(\texttt{inp}\ x\ y; P) &= \texttt{inp}\ x\ (x.y.\varepsilon(P)) & \delta(\texttt{inp}\ x\ (w.y.\mathsf{P})) &= \texttt{inp}\ x\ y; [x/w]\delta(\mathsf{P})\\
\varepsilon(x[\texttt{inl}]; P) &= \texttt{inl}\ x;\ x.\varepsilon(P) & \delta(\texttt{inl}\ x;\ w.\mathsf{P}) &= x[\texttt{inl}]; [x/w]\delta(\mathsf{P})
\end{aligned}$$

The bijection between well-typed processes is subtle because we must account for different structural properties in each system and slight differences in the process syntax. For example, the judgment $\texttt{close}\ x \Vdash \Gamma, x : 1$ is derivable in SCP for any Γ, whereas the judgment $\texttt{close}\ x \vdash \Gamma, x : 1$ is derivable in CP only if Γ is empty. The key insight is that the bijection holds only if the SCP process uses each channel in its context linearly. This restriction to linear SCP processes is unproblematic because we only ever consider such processes in our development.

Before stating the equivalence theorem, we introduce two lemmas that we use in its proof. Both lemmas are proved by induction on the derivation of the typing judgment.

Lemma 3.2 (Weakening). *If* $\mathsf{P} \Vdash \Gamma$, *then* $\mathsf{P} \Vdash \Gamma, x : A$.

Lemma 3.3 (Strengthening). *If* $\mathsf{P} \Vdash \Gamma, x : A$ *and* $x \notin \mathrm{fn}(\mathsf{P})$, *then* $\mathsf{P} \Vdash \Gamma$.

Notation. We write $\mathsf{lin}(\Delta,\ \mathsf{P})$ as shorthand for $\forall x \in \mathrm{dom}(\Delta).\mathsf{lin}(x,\ \mathsf{P})$.

The equivalence theorem shows that we can not only faithfully embed CP processes in SCP but also their typing derivations. Indeed, Theorem 3.4 states that each CP derivation determines the typing derivation of a linear SCP process and that each typing derivation of a linear SCP process can be obtained by weakening a CP typing derivation. This structure-preserving embedding of CP derivations in SCP is given by induction on the derivation. The general strategy is that we interleave the CP derivation with the appropriate linearity checks. We give a more detailed overview of the encoding and decoding maps alongside the proof of the following theorem in Appendix A of an extended version of this paper [**?**].

Theorem 3.4 (Adequacy). *The function* δ *is left inverse to* ε, *i.e.,* $\delta(\varepsilon(P)) = P$ *for all CP processes* P. *The syntax-directed nature of* ε *and* δ *induces functions between CP typing derivations and typing derivations of linear SCP processes:*

(1) If $\mathcal{D}$ *is a derivation of* $P \vdash \Delta$, *then there exists a derivation* $\varepsilon(\mathcal{D})$ *of* $\varepsilon(P) \Vdash \Delta$, *and* $\mathsf{lin}(\Delta,\ \varepsilon(P))$ *and* $\delta(\varepsilon(\mathcal{D})) = \mathcal{D}$.

(2) If $\mathcal{D}$ is a derivation of $\mathsf{P} \Vdash \Gamma, \Delta$ where $\mathrm{fn}(\mathsf{P}) = \mathrm{dom}(\Delta)$ and $\mathsf{lin}(\Delta,\ \mathsf{P})$, then there exists a derivation $\delta(\mathcal{D})$ of $\delta(\mathsf{P}) \vdash \Delta$, and $\varepsilon(\delta(\mathsf{P})) = \mathsf{P}$. Moreover, $\mathcal{D}$ is the result of weakening the derivation $\varepsilon(\delta(\mathcal{D}))$ of $\mathsf{P} \Vdash \Delta$ by Γ.

3.4 Reduction and Type Preservation

The dynamics of SCP is given by translation to and from CP. In particular, we write $\mathsf{P} \Rightarrow_{SCP} \mathsf{Q}$ whenever $\delta(\mathsf{P}) \Rightarrow_{CP} Q$ and $\varepsilon(Q) = \mathsf{Q}$ for some CP process Q. This translation satisfies the usual type-preservation property:

Lemma 3.5. *If $\mathsf{P} \Vdash \Delta$ and $\mathsf{lin}(\Delta,\ \mathsf{P})$, then $\mathrm{fn}(\mathsf{P}) = \mathrm{dom}(\Delta)$.*

Proof. By induction, $\mathsf{lin}(x,\ \mathsf{P})$ implies $x \in \mathrm{fn}(\mathsf{P})$, so $\mathsf{lin}(\Delta,\ \mathsf{P})$ implies $\mathrm{dom}(\Delta) \subseteq \mathrm{fn}(\mathsf{P})$. For the opposite inclusion, $\mathsf{P} \Vdash \Delta$ implies $\mathrm{dom}(\Delta) \supseteq \mathrm{fn}(\mathsf{P})$ by induction, so $\mathrm{fn}(\mathsf{P}) = \mathrm{dom}(\Delta)$. □

Theorem 3.6 (Subject Reduction). *If $\mathsf{P} \Vdash \Delta$, $\mathsf{lin}(\Delta,\ \mathsf{P})$, and $\mathsf{P} \Rightarrow_{SCP} \mathsf{Q}$, then $\mathsf{Q} \Vdash \Delta$ and $\mathsf{lin}(\Delta,\ \mathsf{Q})$.*

Proof. Assume $\mathsf{P} \Vdash \Delta$, $\mathsf{lin}(\Delta,\ \mathsf{P})$, and $\mathsf{P} \Rightarrow_{SCP} \mathsf{Q}$. Then $\mathrm{fn}(\mathsf{P}) = \mathrm{dom}(\Delta)$ by Lemma 3.5. Adequacy (Theorem 3.4) implies $\delta(\mathsf{P}) \vdash \Delta$. By the assumption $\mathsf{P} \Rightarrow_{SCP} \mathsf{Q}$, there exists a Q such that $\delta(\mathsf{P}) \Rightarrow_{CP} Q$ and $\varepsilon(Q) = \mathsf{Q}$. Subject reduction for CP (Theorem 2.1) implies $Q \vdash \Delta$, so $\mathsf{Q} \Vdash \Delta$ and $\mathsf{lin}(\Delta,\ \mathsf{Q})$ by adequacy again. □

We could instead directly prove Theorem 3.6 by induction on the reduction. This direct proof is mechanized as Theorem 6.4.

Since we mechanize SCP, it is convenient to have the reduction and equivalence rules expressed directly in SCP. We show some such rules below. They are obtained by translating the rules in section 2.2 (the second congruence rule for cut omitted).

$$\frac{}{vx{:}A.(\mathtt{fwd}\ x\ y \parallel \mathsf{Q}) \Rightarrow_{SCP} [y/x]\mathsf{Q}}\ [\beta_{\mathtt{fwd}}] \qquad \frac{\mathsf{P} \Rightarrow_{SCP} \mathsf{P}'}{vx{:}A.(\mathsf{P} \parallel \mathsf{Q}) \Rightarrow_{SCP} vx{:}A.(\mathsf{P}' \parallel \mathsf{Q})}\ [\beta_{\mathtt{cut1}}]$$

$$\frac{}{vx{:}A \oplus B.(\mathtt{inl}\ x;\ w.\mathsf{P} \parallel \mathtt{case}\ x\ (w.\mathsf{Q}_1,\ w.\mathsf{Q}_2)) \Rightarrow_{SCP} vw{:}A.(\mathsf{P} \parallel \mathsf{Q}_1)}\ [\beta_{\mathtt{inl1}}]$$

$$\frac{}{vz{:}C.(\mathtt{inl}\ x;\ w.\mathsf{P} \parallel \mathsf{Q}) \Rightarrow_{SCP} \mathtt{inl}\ x;\ w.vz{:}C.(\mathsf{P} \parallel \mathsf{Q})}\ [\kappa_{\mathtt{inl}}]$$

We obtain SCP's structural equivalence in a similar manner: $\mathsf{P} \equiv \mathsf{Q}$ whenever $\delta(\mathsf{P}) \equiv \delta(\mathsf{Q})$. We show two cases of this direct translation.

$$\frac{}{vx{:}A.(\mathsf{P} \parallel \mathsf{Q}) \equiv vx{:}A^{\perp}.(\mathsf{Q} \parallel \mathsf{P})}\ [\equiv_{\mathtt{comm}}] \qquad \frac{}{vy{:}B.(vx{:}A.(\mathsf{P} \parallel \mathsf{Q}) \parallel \mathsf{R}) \equiv vx{:}A.(\mathsf{P} \parallel vy{:}B.(\mathsf{Q} \parallel \mathsf{R}))}\ [\equiv_{\mathtt{assoc}}]$$

4 ENCODING SCP IN LF

We now encode each component of SCP in the logical framework LF. Throughout this section, we make liberal modifications to the working code for presentation/readability purposes.

4.1 Types

We encode session types in LF by defining the LF type `tp : type`. The type constants for this type correspond to the type constructors in SCP.

```
1 : tp.              % termination ("provider")     ⅋ : tp → tp → tp. % channel input
⊥ : tp.              % termination ("client")       & : tp → tp → tp. % receive choice
⊗ : tp → tp → tp. % channel output                 ⊕ : tp → tp → tp. % send choice
```

We use the LF type family `dual : tp → tp → type` to represent duality as a relation between two types. The constants of this type family correspond to the equational definition of duality. In particular, `dual A A'` encodes $A = A^\perp$ (where $A^\perp = A'$).

```
D1 : dual 1 ⊥.
D⊥ : dual ⊥ 1.
D⊗ : dual A A' → dual B B'
   → dual (A ⊗ B) (A' ⅋ B').
```

```
D⅋ : dual A A' → dual B B'
   → dual (A ⅋ B) (A' ⊗ B').
D& : dual A A' → dual B B'
   → dual (A & B) (A' ⊕ B').
D⊕ : dual A A' → dual B B'
   → dual (A ⊕ B) (A' & B').
```

4.2 Processes

We give an encoding of processes by interpreting all channel bindings as intuitionistic functions in LF. First, we define channel names as the type family `name`. Unlike in the functional setting where everything is an expression, in the process calculus setting, channels and processes are distinct. This leads to a so-called weak-HOAS encoding [Despeyroux et al. 1995]. We then introduce the predicate `proc`, standing for processes.

```
name : type.    % channel names
proc : type.    % process
```

We first encode `fwd` x y, `close` x, and `wait` x; `P`, which introduce no channel bindings. The former requires two names and the latter two require one name.

```
fwd : name → name → proc.  % fwd x y
close : name → proc.       % close x
wait : name → proc → proc. % wait x; P
```

The processes `inl` x; w.P, `inr` x; w.P, and `case` x (w.P, w.Q) first require some name x. They then bind a fresh continuation channel w to the continuation processes P and Q. We therefore encode the continuation processes as intuitionistic functions `name → proc`.

```
inl : name → (name → proc) → proc. % x.inl; w.P
inr : name → (name → proc) → proc. % x.inr; w.P
choice : name → (name → proc) → (name → proc) → proc. % case x (w.P, w.Q)
```

Channel output `out` x; (y.P ‖ w.Q) binds the channel y to the process P and the continuation channel w to the process Q. We therefore encode y.P and w.Q as intuitionistic functions `name → proc`:

```
out : name → (name → proc) → (name → proc) → proc. % out x y; (y.P || w.Q)
```

Similarly, channel input `inp` x (w.y.P) binds two channels to P: the continuation channel w and the received channel y. We therefore encode P as a function with two channel names as input.

```
inp : name → (name → name → proc) → proc. % inp x; (w.y.P)
```

Parallel composition $\nu x{:}A.(\mathsf{P} \parallel \mathsf{Q})$ takes some session type A and binds a fresh x to both P and Q, so we encode both processes as functions.

```
pcomp : tp → (name → proc) → (name → proc) → proc. % νx:A. (P || Q)
```

4.3 Linearity Predicate

On paper, we inductively defined a predicate $\mathsf{lin}(x,\ \mathtt{P})$ that checks if x occurs "linearly" in a process P. This predicate clearly respects renaming – if $\mathsf{lin}(x,\ \mathtt{P})$ and y is fresh with respect to P, then $\mathsf{lin}(y,\ [y/x]\mathtt{P})$. We encode this predicate in LF as a type family over functions from names x to processes P. Inhabitants of this family correspond to functions that produces a process that treats its input channel linearly.

```
linear : (name → proc) → type.
```

Unlike our encodings of types and duality, processes can depend on assumptions of the form `x1 : name , ... , xn : name` that are stored in the so-called *ambient* context. In fact, in Beluga, we always consider an object with respect to the context in which it is meaningful. In the on-paper definition of linearity (see section 3.2) we left this context implicit and only remarked that the set of free names $\mathsf{fn}(\mathtt{P})$ of a process P is a subset of this ambient context of channel names. However, when we encode the linearity predicate in LF, we need to more carefully quantify over channel names as we recursively analyze the linearity of a given process.

Intuitively, we define the constructors for linearity by pattern matching on various process constructors. By convention, we will use capital letters for metavariables that are implicitly quantified at the outside. These metavariables describe closed LF terms; in particular when the metavariables stand for processes, it requires that the processes *not* depend on any local, internal bindings. We heavily exploit this feature in our encoding to obtain side conditions of the form $x \notin \mathsf{fn}(P)$ for free.

We begin by translating the axioms in section 3.2:

```
l_fwd1 : linear (λx. fwd x Y).
l_fwd2 : linear (λx. fwd Y x).
l_close : linear (λx. close x).
l_wait : linear (λx. wait x P).
```

$$\frac{}{\mathsf{lin}(x,\ \mathsf{fwd}\ x\ y)}\ L_{\mathsf{fwd1}} \qquad \frac{}{\mathsf{lin}(y,\ \mathsf{fwd}\ x\ y)}\ L_{\mathsf{fwd2}}$$

$$\frac{}{\mathsf{lin}(x,\ \mathsf{close}\ x)}\ L_{\mathsf{close}} \qquad \frac{x \notin \mathsf{fn}(\mathtt{P})}{\mathsf{lin}(x,\ \mathsf{wait}\ x;\ \mathtt{P})}\ L_{\mathsf{wait}}$$

Here, `Y : name` in both `l_fwd1` and `l_fwd2` are implicitly quantified at the outside and cannot depend on the input channel i.e.. $x \neq Y$. Similarly, the metavariable `P : proc` in `l_wait` cannot depend on the input channel x, satisfying the condition that $x \notin \mathsf{fn}(\mathtt{P})$.

The remaining principal cases must continue to check for linearity in the continuation process. Consider the principal case for channel output:

```
% where Q : (name → proc)
l_out : linear Q → linear (λx. out x P Q).
```

$$\frac{\mathsf{lin}(w,\ \mathtt{Q}) \quad x \notin \mathsf{fn}(\mathtt{P}) \cup \mathsf{fn}(\mathtt{Q})}{\mathsf{lin}(x,\ \mathsf{out}\ x;\ (y.\mathtt{P} \parallel w.\mathtt{Q}))}\ L_{\mathsf{out}}$$

The premise $\mathsf{lin}(w,\ \mathtt{Q})$ corresponds to the input `linear Q` for this constructor because we encode Q as a function `name → proc`. The additional condition that x does not appear in P and Q follows because `P` and `Q` are metavariables, meaning they cannot depend on the internally bound `x : name`.

The encoding of the principal case for channel input requires a bit more care. Recall the on-paper rule:

$$\frac{\mathsf{lin}(w,\ \mathtt{P}) \quad x \notin \mathsf{fn}(\mathtt{P})}{\mathsf{lin}(x,\ \mathsf{inp}\ x\ (w.y.\mathtt{P}))}\ L_{\mathsf{inp}}$$

Following the strategy for channel output, we would like to continue checking that the continuation channel w appears linearly in P by requiring it as an input in our encoding. But since we encode P as a two argument function `name → name → proc`, we cannot simply say

```
l_inp : linear P
     → linear (λx. inp x P). % WRONG
```

Instead, what we need as our premise is the fact that P is linear with respect to some input w given *any* y. To check this, we universally quantify over `y` using the syntax `{y : name}`:

```
l_inp : ({y:name} linear (λw. P w y))
     → linear (λx. inp x P).
```

The condition that x does not appear in P again follows from the fact that P must be closed.

The other principal cases are standard translations, which we present in a less verbose manner. The continuation channels are checked in the same style as in channel output.

```
l_inl : linear P → linear (λx. inl x P).
l_inr : linear P → linear (λx. inr x P).
l_choice : linear P → linear Q
        → linear (λx. choice x P Q).
```

$$\frac{\mathsf{lin}(w,\ \mathsf{P}) \quad x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{inl}\ x;\ w.\mathsf{P})}\ L_{\mathsf{inl}} \qquad \frac{\mathsf{lin}(w,\ \mathsf{P}) \quad x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{inr}\ x;\ w.\mathsf{P})}\ L_{\mathsf{inr}}$$

$$\frac{\mathsf{lin}(w,\ \mathsf{P}) \quad \mathsf{lin}(w,\ \mathsf{Q}) \quad x \notin \mathrm{fn}(\mathsf{P}) \cup \mathrm{fn}(\mathsf{Q})}{\mathsf{lin}(x,\ \mathtt{case}\ x\ (w.\mathsf{P},\ w.\mathsf{Q}))}\ L_{\mathsf{case}}$$

The congruence cases follow similar ideas except with complex bindings as in the principal case for input. The simplest case is the encoding of wait:

```
l_wait2 : linear P → linear (λz. wait X (P z)).
```

$$\frac{\mathsf{lin}(z,\ \mathsf{P})}{\mathsf{lin}(z,\ \mathtt{wait}\ x;\ \mathsf{P})}\ L_{\mathsf{wait2}}$$

Here, it is important to recognize that `(P z)` is of type `proc` according to the `wait` constructor, meaning P is of type `name → proc`. Therefore, requiring `linear P` corresponds to checking $\mathsf{lin}(z,\ \mathsf{P})$.

The congruence case for input is perhaps the most extreme instance of this complex binding:

```
l_inp2 : ({w:name}{y:name} linear (λz. P z w y))
      → linear (λz. inp X (P z)).
```

$$\frac{\mathsf{lin}(z,\ \mathsf{P})}{\mathsf{lin}(z,\ \mathtt{inp}\ x\ (w.y.\mathsf{P}))}\ L_{\mathsf{inp2}}$$

Here, `(P z)` is of type `name → name → proc`, so we check for linearity of `z` by requiring it to be linear with any `w` and `y`.

Next, we consider the congruence cases for parallel composition.

```
l_pcomp1 : ({x:name} linear (λz. P x z))
        → linear (λz. (pcomp A (λx. P x z) Q)).
l_pcomp2 : ({x:name} linear (λz. Q x z))
        → linear (λz. pcomp A P (λx. Q x z)).
```

$$\frac{\mathsf{lin}(z,\ \mathsf{P}) \quad z \notin \mathrm{fn}(\mathsf{Q})}{\mathsf{lin}(z,\ \nu x{:}A.(\mathsf{P} \parallel \mathsf{Q}))}\ L_{\nu 1}$$

$$\frac{\mathsf{lin}(z,\ \mathsf{Q}) \quad z \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(z,\ \nu x{:}A.(\mathsf{P} \parallel \mathsf{Q}))}\ L_{\nu 2}$$

Since `Q` is a metavariable in `l_pcomp1`, it must be closed with respect to `z`, so it satisfies the condition $z \notin \mathrm{fn}(\mathsf{Q})$. The condition $z \notin \mathrm{fn}(\mathsf{P})$ in `l_pcomp2` is satisfied for the same reason.

We summarize the remaining cases below.

```
l_out2 : ({y:name} linear (λz. P z y))
      → linear (λz. out X (P z) Q).
l_inl2 : ({x':name} linear (λz. P z x'))
      → linear (λz. inl X (P z)).
l_choice2 : ({x':name} linear (λz. P z x'))
         → ({x':name} linear (λz. Q z x'))
         → linear (λz. choice X (P z) (Q z)).
```

```
l_out3 : ({x':name} linear (λz. Q z x'))
      → linear (λz. out X P (Q z)).
l_inr2 : ({x':name} linear (λz. P z x'))
      → linear (λz. inr X (P z)).
```

4.4 Type Judgments

To encode session typing, we follow the encoding for the sequent calculus in the logical framework LF (see for example [Harper et al. 2009]). Since type judgments depend on assumptions of the form

$x : A$, we introduce the type family `hyp : name → tp → type` to associate a channel name with a session type. We then encode the type judgment $P \Vdash \Gamma$ as a judgment on a process: `wtp : proc → type` with ambient assumptions of the form `x1 : name, h1 : hyp x1 A1, ..., xn : name, hn : hyp xn An` which represent Γ. Note that the use of these assumptions is unrestricted, but the linearity predicate ensures that if an assumption is used, then it is used linearly. As an example, we could encode the rule

$$\frac{}{\mathsf{close}\ x \Vdash \Gamma, x : 1}\ [1]$$

in an obvious manner:

```
wtp_close : {X:name}hyp X 1 → wtp (close X).
```

To establish `wtp (close X)`, we must have an assumption `hyp X 1`. While it is not strictly necessary to explicitly quantify over the channel name `X`, doing so makes encoding the metatheory easier.

Forwarding requires two channels of dual type:

```
wtp_fwd : dual A A'
       → {X:name} hyp X A → {Y:name} hyp Y A'
       → wtp (fwd X Y).
```

$$\frac{}{\mathsf{fwd}\ x\ y \Vdash \Gamma, x : A, y : A^{\perp}}\ [\mathsf{Id}]$$

We encode this rule by requiring a duality relation between two session types A and A' alongside corresponding hypotheses that X and Y are of type A and A' respectively.

The encoding of parallel composition requires a similar trick for duality.

```
wtp_pcomp : dual A A'
  → ({x:name} hyp x A → wtp (P x))
  → ({x:name} hyp x A' → wtp (Q x))
  → linear P → linear Q
  → wtp (pcomp A P Q).
```

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \quad \mathsf{lin}(x,\ \mathsf{P}) \quad \mathsf{Q} \Vdash \Gamma, x : A^{\perp} \quad \mathsf{lin}(x,\ \mathsf{Q})}{\nu x{:}A.(\mathsf{P} \parallel \mathsf{Q}) \Vdash \Gamma}\ [\mathsf{Cut}]$$

We encode the premise $\mathsf{P} \Vdash \Gamma, x : A$ as a function that takes some `x : name` and assumption `hyp x A` to prove that `(P x)` is well-typed. A different reading of this premise is simply as "for all `x : name`, assuming `hyp x A`, we show that `wtp (P x)`". The premise $\mathsf{lin}(x, \mathsf{P})$ corresponds to `linear P` since `P` is of type `name → proc`, and the remaining two premises follow the same idea.

Continuation channels are simply treated as bindings in the same way we treat cut. For instance:

```
wtp_inl : {X:name} hyp X (A ⊕ B)
        → ({w:name} hyp w A → wtp (P w))
        → wtp (inl X P).
```

$$\frac{\mathsf{P} \Vdash \Gamma, x : A \oplus B, w : A}{\mathsf{inl}\ x;\ w.\mathsf{P} \Vdash \Gamma, x : A \oplus B}\ [\oplus_1]$$

The first two inputs to the constructor is a name `X` and a hypothesis that `X` is of type `A ⊕ B`. The next input is that the continuation process `(P w)` is well-typed given an assumption `w : name` and `hyp w A`, corresponding to the premise of the $[\oplus_1]$ rule.

The remaining cases follows a similar pattern. Linearity is checked for the freshly bound channels on channel output and input as in the typing for parallel composition. We defer the full encoding to the attached artifact.

4.5 Reductions and Structural Equivalence

We model both reductions $\mathsf{P} \Rightarrow_{SCP} \mathsf{Q}$ and structural equivalences $\mathsf{P} \equiv \mathsf{Q}$ as relations.

```
step : proc → proc → type.
equiv : proc → proc → type.
```

The encoding is fairly simple. For example, consider

```
βfwd : step (pcomp A (λx. fwd x Y) Q) (Q Y).
```

$$\frac{}{\nu x{:}A.(\mathsf{fwd}\ x\ y \parallel \mathsf{Q}) \Rightarrow_{SCP} [y/x]\mathsf{Q}}\ [\beta_{\mathsf{fwd}}]$$

Since `Y : name` and `Q : name → proc`, we rely on the LF application `(Q Y)` to accomplish the object-level substitution $[y/x]Q$.

We write congruence rules by requiring the inner process to step under some arbitrary `x : name`:

```
βcut1 : ({x:name} step ((P x) (P' x)))
      → step (pcomp A P Q) (pcomp A P' Q).
```

$$\frac{\mathsf{P} \Rightarrow_{SCP} \mathsf{P}'}{vx{:}A.(\mathsf{P} \parallel \mathsf{Q}) \Rightarrow_{SCP} vx{:}A.(\mathsf{P}' \parallel \mathsf{Q})}\ [\beta_{\mathtt{cut1}}]$$

Principal rules, such as

$$\frac{}{vx{:}A \oplus B.(\mathtt{inl}\ x;\ w.\mathsf{P} \parallel \mathtt{case}\ x\ (w.\mathsf{Q}_1,\ w.\mathsf{Q}_2)) \Rightarrow_{SCP} vw{:}A.(\mathsf{P} \parallel \mathsf{Q}_1)}\ [\beta_{\mathtt{in11}}]$$

can be encoded straightforwardly:

```
βinl : step (pcomp (A ⊕ B) (λx. inl x P) (λx. choice x Q R))
           (pcomp A P Q).
```

The names of the bound channels x and w are not explicit since the metavariables `P`, `Q`, and `R` are all functions `name → proc` and can take an arbitrary name.

The remaining reduction rules and structural equivalences are similarly encoded. Since there are no interesting cases to discuss, we defer the complete presentation to the included artifact.

5 ADEQUACY OF THE ENCODING

In this section we prove adequacy for each component of our encoding of SCP. Since the proofs are verbose, we mainly focus on stating the right adequacy lemmas while giving a high-level overview on the proof strategy for the more complex lemmas. We give more details of the proof in Appendix B of an extended version of this paper [?].

5.1 Notation

We use the sequent $\Gamma \vdash_{LF} M : \tau$ to refer to judgments within LF. For instance, $\vdash_{LF} M : \mathrm{tp}$ asserts that the LF term M is of type tp under no assumptions. Similarly, $\Gamma \vdash_{LF} D : \mathrm{wtp}\ P$ asserts that the LF term D is of type $\mathrm{wtp}\ P$ where P is some LF term of type proc. Informally, D in this context would correspond to a typing derivation. We also work with LF *canonical* forms, essentially the $\beta\eta$ normal forms of a given type, as is standard in adequacy statements.

5.2 Session Types and Duality

Adequacy for the encoding of session types can be shown with the obvious translation function $\ulcorner - \urcorner$ that maps session types A to LF terms $\ulcorner A \urcorner$ of type `tp`.

Lemma 5.1 (Adequacy of tp). *There exists a bijection between the set of session types and canonical LF terms M such that $\vdash_{LF} M : \mathrm{tp}$.*

Adequacy of duality is also easy to show once stated properly. Since there is a slight difference between the on-paper definition of duality as a unary function and the LF encoding of duality as a relation, we state adequacy for the encoding of duality as follows.

Lemma 5.2 (Adequacy of dual).

(1) For any session type A, there exists a unique LF canonical form D such that $\vdash_{LF} D : \mathrm{dual}\ \ulcorner A \urcorner\ \ulcorner A^{\perp} \urcorner$.
(2) For any LF canonical form D such that $\vdash_{LF} D : \mathrm{dual}\ \ulcorner A \urcorner\ \ulcorner A' \urcorner$, $A' = A^{\perp}$.

5.3 Processes

Adequacy of the process encoding also follows naturally from our encoding. In particular, all channel bindings, which we encode as intuitionistic functions, precisely match the process syntax of SCP. We can therefore define a translation $\ulcorner - \urcorner$ from processes in SCP to LF normal forms and its decoding $\lfloor - \rfloor$ in the obvious manner.

Definition 5.3. The encoding of name sets to an LF context is given as follows:

$$\ulcorner x_1, \ldots, x_n \urcorner = x_1{:}\mathsf{name}, \ldots, x_n{:}\mathsf{name}$$

Lemma 5.4 (Adequacy of proc). *For each SCP process* P*, there exists a unique canonical LF form* $\ulcorner \mathrm{fn}(\mathsf{P}) \urcorner \vdash_{LF} \ulcorner \mathsf{P} \urcorner : \mathsf{proc}$ *and* $\lfloor \ulcorner \mathsf{P} \urcorner \rfloor = \mathsf{P}$*. Conversely, if* $\Gamma \vdash_{LF} M : \mathsf{proc}$ *is a canonical LF form, then* $\lfloor M \rfloor$ *is an SCP process,* $\ulcorner \lfloor M \rfloor \urcorner = M$*, and* $\ulcorner \mathrm{fn}(\lfloor M \rfloor) \urcorner \subseteq \Gamma$*.*

The context $\ulcorner \mathrm{fn}(\mathsf{P}) \urcorner$ captures the required assumptions to construct a LF term corresponding to a given process. For example, an encoding of $\mathtt{fwd}\ x\ y$ corresponds to the LF term $x{:}\mathsf{name}, y{:}\mathsf{name} \vdash_{LF} \mathtt{fwd}\ x\ y : \mathsf{proc}$. Indeed, $\ulcorner \mathrm{fn}(\mathtt{fwd}\ x\ y) \urcorner = x{:}\mathsf{name}, y{:}\mathsf{name}$, allowing the `fwd` constructor to be applied with the assumptions $x{:}\mathsf{name}$ and $y{:}\mathsf{name}$.

Unfortunately, we cannot give a clean bijection result due to weakening in LF derivations. For example, there is a derivation of $\Gamma, x{:}\mathsf{name}, y{:}\mathsf{name} \vdash_{LF} \mathtt{fwd}\ x\ y : \mathsf{proc}$ for any Γ, and such derivations all correspond to the SCP process $\mathtt{fwd}\ x\ y$. Therefore, we only require that the overall context include the free names for the converse direction. This weaker statement does not affect later developments since weakening in LF does not change the structure of the derivation. This phenomenon repeats for later adequacy results due to weakening.

5.4 Linearity

We define an encoding $\ulcorner - \urcorner$ that maps derivations of linearity predicates in SCP of form $\mathsf{lin}(x,\ \mathsf{P})$ to LF canonical forms of type $\mathsf{linear}\ (\lambda x.\ulcorner \mathsf{P} \urcorner)$. Similarly, we define a decoding $\lfloor - \rfloor$ that maps LF canonical forms of type $\mathsf{linear}\ M$, where M is of type $\mathtt{name} \to \mathtt{proc}$, to derivations of $\mathsf{lin}(x,\ \lfloor M\ x \rfloor)$.

Lemma 5.5 (Adequacy of linear). *For each derivation* $\mathcal{D}$ *of* $\mathsf{lin}(x,\ \mathsf{P})$*, there exists a unique canonical LF term* $L = \ulcorner \mathcal{D} \urcorner$ *such that* $\ulcorner \mathrm{fn}(\mathsf{P}) \setminus x \urcorner \vdash_{LF} L : \mathsf{linear}\ \lambda x.\ulcorner \mathsf{P} \urcorner$ *and* $\lfloor L \rfloor = \mathcal{D}$*. Conversely, if* $\Gamma \vdash_{LF} L : \mathsf{linear}\ M$ *is a canonical LF form, then* $\lfloor L \rfloor$ *is a derivation of* $\mathsf{lin}(x,\ \lfloor M\ x \rfloor)$ *and* $\ulcorner \mathrm{fn}(\lfloor M\ x \rfloor) \setminus x \urcorner \vdash_{LF} \ulcorner \lfloor L \rfloor \urcorner : \mathsf{linear}\ M$ *where* $\ulcorner \mathrm{fn}(\lfloor M\ x \rfloor) \urcorner \subseteq \Gamma$*.*

Here, the encoding of the context is slightly tricky because we define the linearity predicate on paper using the syntax $\mathsf{lin}(x,\ \mathsf{P})$, meaning $x \in \mathrm{fn}(\mathsf{P})$. In LF however, since we encode the linearity predicate `linear : (name → proc) → type` over intuitionistic functions taking some name x, we must use the context $\ulcorner \mathrm{fn}(\mathsf{P}) \setminus x \urcorner$ when encoding an on-paper derivation of some linearity predicate. More informally, we establish a correspondence between derivations of $\mathsf{lin}(x,\ \mathsf{P})$ and LF canonical forms of $\mathsf{linear}\ (\lambda x.\ulcorner \mathsf{P} \urcorner)$ under an LF context *without* the assumption $x{:}\mathsf{name}$.

At a high level, the proof of this lemma mostly involves ensuring that the various $x \notin \mathrm{fn}(\mathsf{P})$ conditions are fulfilled by our higher-order encoding and vice versa. For example, the encoding of

$$\frac{\mathsf{lin}(w,\ \mathsf{P}) \quad x \notin \mathrm{fn}(\mathsf{P})}{\mathsf{lin}(x,\ \mathtt{inl}\ x;\ w.\mathsf{P})}\ L_{\mathtt{inl}}$$

is `l_inl : linear M → linear (λx. inl x M)`, and in particular, `M` is a metavariable, meaning it cannot depend on the internally bound x, satisfying the side condition of $x \notin \mathrm{fn}(\mathsf{P})$.

5.5 Type Judgments

To establish a relation between SCP type judgments $\mathsf{P} \Vdash \Gamma$ and LF derivations of $\mathtt{wtp}\ulcorner \mathsf{P} \urcorner$, we must define a context mapping of typing assumptions $\Gamma = x_1 : A_1, \ldots, x_n : A_n$.

Definition 5.6. A context encoding $\ulcorner \Gamma \urcorner$ is defined by introducing LF assumptions x:name, h:hyp $x\ \ulcorner A \urcorner$ for each typing assumption in Γ:

$$\ulcorner x_1 : A_1, \ldots, x_n : A_n \urcorner = x_1\text{:name}, h_1\text{:hyp}\ x_1\ \ulcorner A_1 \urcorner, \ldots, x_n\text{:name}, h_n\text{:hyp}\ x_n\ \ulcorner A_n \urcorner$$

We define an encoding $\ulcorner - \urcorner$ and decoding $\lfloor - \rfloor$ of type derivations in our adequacy statement.

Lemma 5.7 (Adequacy of wtp). *There exists a bijection between typing derivations in SCP of form $\mathsf{P} \Vdash \Gamma$ and LF canonical forms D such that $\ulcorner \Gamma \urcorner \vdash_{LF} D : \mathrm{wtp}\ \ulcorner \mathsf{P} \urcorner$*

The proof mostly involves appealing to previous adequacy lemmas and is otherwise fairly straightforward. In fact, the proof for the linearity predicate is more involved due to the implicit implementation of the free name side-conditions using higher-order encoding. This is not too surprising: the design of SCP was heavily motivated by a desire for a system more amenable to mechanization in LF. Furthermore, we have a bijection for type judgments because type judgments in SCP also have weakening, making the adequacy statement very clean.

5.6 Reductions and Structural Equivalences

Adequacy of reductions is easy to show; most rules are axioms, so we simply appeal to the adequacy of the underlying processes. The congruence cases are very simple and follows from the appropriate induction hypotheses. Adequacy of structural equivalence is similarly easy to show.

The adequacy statements are unfortunately slightly cumbersome for the same reason as Lemma 5.4 and Lemma 5.5 since weakening in LF does not allow for a clean bijection. Again, we want to emphasize that this does not change the structure of the derivations of both **step** and **equiv**.

Lemma 5.8 (Adequacy of step). *For each SCP reduction S of $\mathsf{P} \Rightarrow_{SCP} \mathsf{Q}$, there exists a unique canonical LF derivation $\ulcorner \mathrm{fn}(\mathsf{P}) \urcorner \vdash_{LF} \ulcorner S \urcorner : \mathrm{step}\ \ulcorner \mathsf{P} \urcorner\ \ulcorner \mathsf{Q} \urcorner$ and $\lfloor \ulcorner S \urcorner \rfloor = S$. Conversely, if $\Gamma \vdash_{LF} D : \mathrm{step}\ M\ N$ is a canonical LF form, then $\lfloor D \rfloor$ is a derivation of a reduction $\lfloor M \rfloor \Rightarrow_{SCP} \lfloor N \rfloor$, $\ulcorner \lfloor D \rfloor \urcorner = D$, and $\ulcorner \mathrm{fn}(\lfloor M \rfloor) \urcorner \subseteq \Gamma$.*

Lemma 5.9 (Adequacy of equiv). *For each SCP structural equivalence S of $\mathsf{P} \equiv \mathsf{Q}$, there exists a unique canonical LF derivation $\ulcorner \mathrm{fn}(\mathsf{P}) \urcorner \vdash_{LF} \ulcorner S \urcorner : \mathrm{equiv}\ \ulcorner \mathsf{P} \urcorner\ \ulcorner \mathsf{Q} \urcorner$ and $\lfloor \ulcorner S \urcorner \rfloor = S$. Conversely, if $\Gamma \vdash_{LF} D : \mathrm{equiv}\ M\ N$ is a canonical LF derivation, then $\lfloor D \rfloor$ is a derivation of a structural equivalence $\lfloor M \rfloor \equiv \lfloor N \rfloor$, $\ulcorner \lfloor D \rfloor \urcorner = D$, and $\ulcorner \mathrm{fn}(\lfloor M \rfloor) \urcorner \subseteq \Gamma$.*

5.7 Adequacy With Respect to CP

Since we establish a bijection between SCP and our encoding and there exists a bijection between CP and SCP when restricted to well-typed and linear processes, we also conclude that our encoding is adequate with respect to CP when restricted to well-typed and linear processes (in the encoding).

Definition 5.10. An encoding map $\varepsilon_\circ$ of processes and typing derivations in CP to LF is defined by the composition of the encoding ε of CP to SCP with the encoding $\ulcorner - \urcorner$ of SCP to LF, i.e., $\varepsilon_\circ = \ulcorner \varepsilon(-) \urcorner$. Similarly, a decoding map $\delta_\circ$ of processes and typing derivation in LF to CP is defined by the composition of the decoding $\lfloor - \rfloor$ of LF to SCP with the decoding δ of SCP to CP, i.e., $\delta_\circ = \delta(\lfloor - \rfloor)$.

Corollary 5.11. *The encoding function $\varepsilon_\circ$ is left inverse to $\delta_\circ$ and*

(1) If $\mathcal{D}$ is a derivation of $P \vdash \Delta$ where $\Delta = x_1\text{:}A_1, \ldots, x_n\text{:}A_n$, then there exists a collection of LF canonical forms $\{W, L_1, \ldots, L_n\}$ such that

- $W = \varepsilon_\circ(\mathcal{D})$ *such that* $\ulcorner \Delta \urcorner \vdash_{LF} W : \mathsf{wtp}\ \varepsilon_\circ(P)$
- $\ulcorner \mathrm{fn}(P) \setminus x_i \urcorner \vdash_{LF} L_i : \mathsf{linear}\ \lambda x_i.\varepsilon_\circ(P)$ *for* $1 \leq i \leq n$
- $\delta_\circ(\varepsilon_\circ(\mathcal{D})) = \mathcal{D}$

(2) If $\{W, L_1, \ldots, L_n\}$ *is a collection of LF derivations such that*

- $\Gamma \vdash_{LF} W : \mathsf{wtp}\ M$ *where* $\Gamma = \{x_1{:}\mathsf{name}, h_1{:}\mathsf{hyp}\ x_1\ \ulcorner A_1 \urcorner, \ldots, x_n{:}\mathsf{name}, h_n{:}\mathsf{hyp}\ x_n\ \ulcorner A_n \urcorner\}$
- $\Gamma \setminus \{x_i{:}\mathsf{name}, h_i{:}\mathsf{hyp}\ x_i\ \ulcorner A_i \urcorner\} \vdash_{LF} L_i : \mathsf{linear}\ \lambda x_i.M$ *for* $1 \leq i \leq n$

then there exists a derivation $\delta_\circ(W)$ *of* $\delta_\circ(M) \vdash \Delta$ *and* $\varepsilon_\circ(\delta_\circ(M)) = M$ *such that* $\Gamma = \ulcorner \Delta \urcorner$.

6 MECHANIZING THE TYPE PRESERVATION PROOF

In the previous sections, we focused our attention to the encoding of SCP and its adequacy, which were purely done in the logical framework LF. Now, we give a brief overview of our mechanization of type preservation in the proof assistant Beluga. Mechanizations in Beluga involve encoding the syntax and semantics of the object language in the *LF Layer* and then manipulating LF terms in the *Computational Layer* using contextual types to characterize derivation trees together with the context in which they make sense [Cave and Pientka 2012; Nanevski et al. 2008; Pientka 2008; Pientka and Dunfield 2008]. The contextual types enable clean statements of various strengthening statements, which comprise the majority of the lemmas used in the type preservation proof.

Since the computational layer in Beluga is effectively a functional programming language, inductive proofs of metatheorems are (terminating) recursive functions that manipulate LF objects. For presentation purposes, we assume no familiarity with the computational layer of Beluga and explain the lemmas and theorems informally in words. We defer to the accompanying artifact for the implementation details of all the lemmas and theorems below.

6.1 Lemmas of dual

Due to our encoding of duality as a relation between two types, we must prove symmetry and uniqueness. The encoding of symmetry is a recursive function `dual_sym` that takes as input a closed LF object of type dual A A' and outputs a closed LF object of type dual A' A. The encoding of uniqueness takes two closed LF objects of type dual A A' and dual A A'' and outputs a proof that $A' = A''$. To encode the equality of session types $A' = A''$, we follow the standard technique of defining an equality predicate `eq`: `tp` → `tp` → **`type`** over session types with reflexivity as its constructor.

```
% Symmetricity and Uniqueness
rec dual_sym : [ ⊢ dual A A' ] → [ ⊢ dual A' A] =
/ total 1 /
fn d ⇒
case d of
| [ ⊢ D1] ⇒ [ ⊢ D⊥]
| [ ⊢ D⊗  Dl Dr] ⇒
  let [ ⊢ l] = dual_sym [ ⊢ Dl] in
  let [ ⊢ r] = dual_sym [ ⊢ Dr] in
  [ ⊢ D⅋  l r]
| ...

rec dual_uniq : [ ⊢ dual A A' ] → [ ⊢ dual A A''] → [ ⊢ eq A' A''] = ...
```

The use of the contextual box with no assumptions `[ ⊢ ...]` captures closed objects. The contextual variables (or metavariables) `A` and `A'` are implicitly quantified at the outside. The implementations of the two functions pattern match on the input with appropriate recursive calls for the binary type constructors, corresponding to the usual induction proofs for these lemmas. We show only

one base case and one recursive case to give the flavour of how proofs are written as recursive programs. The totality annotation checks that the program is covering and that all recursive calls on the first (explicit) argument are structurally smaller and decreasing.

6.2 Strengthening Lemmas

Next, we encode strengthening lemmas for contextual LF terms of various types. First, we present them informally below using LF-like syntax, using $\vdash$ instead of $\vdash_{LF}$ and omitting LF term names for economical purposes:

Lemma 6.1 (Strengthening Lemmas).

(1) If `Γ, z : name, h : hyp z C` $\vdash$ `hyp X A` *and* $z \neq X$, *then* Γ $\vdash$ `hyp X A`.
(2) If `Δ, z : name` $\vdash$ `linear λx.P` *and* $z \notin \mathrm{fn}(P)$, *then* Δ $\vdash$ `linear λx.P`.
(3) If `Γ, z : name, h : hyp z C` $\vdash$ `wtp P` *and* $z \notin \mathrm{fn}(P)$, *then* Γ $\vdash$ `wtp P`.
(4) If `Δ, z : name` $\vdash$ `step P Q` *and* $z \notin \mathrm{fn}(P)$, *then* $z \notin \mathrm{fn}(Q)$ *and* Δ $\vdash$ `step P Q`.
(5) If `Δ, z : name` $\vdash$ `equiv P Q` *and* $z \notin \mathrm{fn}(P)$, *then* $z \notin \mathrm{fn}(Q)$ *and* Δ $\vdash$ `equiv P Q`.

where Γ *consists of assumptions of form* `x1 : name, h1 : hyp x1 A1, . . . , xn : name, hn : hyp xn An` *and* Δ *consists of assumptions of form* `x1 : name, . . . , xn : name`.

The use of different contexts Γ and Δ in these statements mostly indicate the spirit of the judgments that we strengthen. Linearity for instance should not depend on typing assumptions, so we use Δ. In practice, picking the right kind of context to use proved immensely useful in simplifying the final type preservation proof. In particular, we found that it is more convenient to weaken the final two lemmas regarding `step` and `equiv` by stating them under the richer context Γ.

To encode Δ and Γ in Beluga, we first define *context schemas*. In our case, we are interested in contexts containing assumptions of names, i.e., Δ, and assumptions of names alongside their types for the typing judgments, i.e., Γ:

```
schema nctx = name;
schema ctx = some [A:tp] block x:name, h:hyp x A;
```

In the statement of our lemma, we exploit the full power of contextual variables to cleanly state the strengthening lemmas. For instance, we encode the side-condition that $z \neq X$ in the strengthening of `hyp X A` by requiring that `X` does not depend on `z`:

```
rec str_hyp : (Γ:ctx) [Γ, z:name, h:hyp z C[] ⊢ hyp X[..] A[]] → [Γ ⊢ hyp X A[]] = ...
```

We first implicitly abstract over the context Γ specifying what kind of context we are working in. Further, contextual variables such as `X` or `A` are associated with a substitution. By default, they are associated with the identity substitution which can be omitted by the user. However, Beluga also allows us to associate contextual variables with more interesting substitutions. The weakening substitution on the name `X[..]` ensures that `X` only depends on Γ and not `z` or `h`, which indeed captures the requirement $z \neq X$. The empty substitutions on the session types `A[]` and `C[]` indicate that they do not depend on anything, i.e., they are closed. We encode the requirement that $z \notin \mathrm{fn}(P)$ in the strengthening lemmas for linearity and typing using a similar technique:

```
rec str_lin : (Δ:nctx) [Δ, z:name ⊢ linear λy. P[.., y]] → [Δ ⊢ linear λy. P] = ...
rec str_wtp : (Γ:ctx) [Γ, z:name, h:hyp z C[] ⊢ wtp P[..]] → [Γ ⊢ wtp P] = ...
```

The substitutions associated with the variable `P` in `P[.., y]` and `P[..]` encode that the process `P` does not depend on the assumption `z` that we want to strengthen out, properly capturing the side-condition of $z \notin \mathrm{fn}(P)$ in both lemmas. Indeed, `str_wtp` turns out to be a mechanization of Lemma 3.3. The proofs of these lemmas are straightforward and are given by pattern matching on the input.

The final two strengthening lemmas are a bit different because of the additional free-name condition in the conclusions. Suppose we naively follow the prior attempts:

```
rec str_step : (Γ : ctx) [Γ, x:name ⊢ step P[..] Q] → [Γ ⊢ step P Q] = ...
```

Unfortunately, the conclusion `Γ ⊢ step P Q` is not well-typed since `Q` as used in the premise depends on `Γ, x:name` whereas `Q` as used in the conclusion only depends on `Γ`. If we change the premise to `[Γ, x:name ⊢ step P[..] Q[..]` to require that `Q` only depends on `Γ`, then the lemma is not strong enough. Indeed, encoding the strengthening lemma actually requires an existential; we must say that there exists some process `Q'` such that `Γ ⊢ step P Q'` and `Q = Q'`. However, since LF does not have sigma types, we must further encode this existential using a data structure `Result`, whose only constructor takes the process `Q'`, a proof that `Q = Q'`, and a proof that `step P Q'`. As before, we define equality of processes `eq_proc` as a relation with only the reflexivity constructor.

```
inductive Result : (Γ : ctx){P : [Γ ⊢ proc]}{Q : [Γ, x:name ⊢ proc]} → ctype =
| Res : {Q' : [Γ ⊢ proc]}
     → [Γ, x:name ⊢ eq_proc Q Q'[..]]
     → [Γ ⊢ step P Q']
     → Result [Γ ⊢ P] [Γ, x:name ⊢ Q];
```

We can now state the lemma using this data structure:

```
rec str_step : (Γ : ctx) [Γ, x:name ⊢ step P[..] Q] → Result [Γ ⊢ P] [Γ, x:name ⊢ Q] = ...
```

We follow an analogous procedure for strengthening structural equivalences and prove the two lemmas simultaneously via mutual recursion.

6.3 Auxiliary Lemmas

We prove two additional lemmas to aid in the type preservation proof. The first lemma states that $\mathsf{lin}(x, \mathsf{P})$ implies $x \in \mathrm{fn}(\mathsf{P})$. We however work with its contrapositive since we do not directly encode $\mathrm{fn}(\mathsf{P})$.

Lemma 6.2 (Linearity requires usage). *If $x \notin \mathrm{fn}(\mathsf{P})$, then* `Γ ⊢ linear (λx.P)` *is not derivable.*

We encode the contradiction in the lemma using the standard LF technique of defining a type `imposs` without any constructors. The encoding of the lemma is therefore a function that takes as input `[Δ ⊢ linear (λx. P[..])]` and outputs some `imposs`. The substitution `P[..]` indicates that the process does not depend on the input name x which properly captures the premise $x \notin \mathrm{fn}(\mathsf{P})$.

```
imposs : type.
% no constructor for imposs
rec lin_name_must_appear : (Δ : nctx) [Δ ⊢ linear (λx. P[..])] → [ ⊢ imposs] = ...
```

Next, we show that structural equivalence preserves both linearity and typing. To state preservation for linearity, we have to reconcile the fact that linearity is defined parametric to some channel name, so we must extend the context of `equiv` with an additional name.

Lemma 6.3 (Structural Equivalence preserves linearity and typing).

(1) If `Γ, x:name ⊢ equiv P Q` *and* `Γ ⊢ linear λx.P`, *then* `Γ ⊢ linear λx.Q`.
(2) If `Γ ⊢ equiv P Q` *and* `Γ ⊢ wtp P`, *then* `Γ ⊢ wtp Q`.

Although the first lemma can in spirit be stated under a context of names Δ, we used the more general context of names and types Γ to better suit our type preservation proof.

```
rec lin_s_equiv : (Γ : ctx) [Γ, x:name ⊢ equiv P Q]
                       → [Γ ⊢ linear (λx. P)]
                       → [Γ ⊢ linear (λx. Q)] = ...
```

```
rec wtp_s_equiv : (Γ : ctx) [Γ ⊢ equiv P Q]
                          → [Γ ⊢ wtp P]
                          → [Γ ⊢ wtp Q] = ...
```

Note that our proof shows that linearity is preserved for any given (free) channel x, meaning that the on-paper predicate $\mathsf{lin}(\Delta,\ \mathtt{P})$ is also preserved by structural equivalence.

6.4 Type Preservation

Finally, we are ready to state the main theorem. To state preservation of linearity, we extend the contexts of other judgments appropriately in the same manner as for `equiv`.

Theorem 6.4 (Type Preservation).

(1) If Γ, x : `name` ⊢ `step P Q` *and* Γ, x : `name`, h : `hyp x A` ⊢ `wtp P` *and* Γ ⊢ `linear` λx.P, *then* Γ ⊢ `linear` λx.Q.

(2) If Γ ⊢ `step P Q` *and* Γ ⊢ `wtp P`, *then* Γ ⊢ `wtp Q`.

The encodings for these statements are very similar to the encodings for Lemma 6.3:

```
rec lin_s : (Γ : ctx) [Γ, x:name, h:hyp x A[] ⊢ wtp P[..,x]]
                    → [Γ, x:name ⊢ step P Q]
                    → [Γ ⊢ linear (λx. P)]
                    → [Γ ⊢ linear (λx. Q)] = ...
and rec wtp_s : (Γ : ctx) [Γ ⊢ wtp P]
                    → [Γ ⊢ step P Q]
                    → [Γ ⊢ wtp Q] = ...
```

The implementations for both functions proceed by case analysis on the term of type `[Γ, x:name ⊢ step P Q]`. Preservation of linearity is perhaps the more interesting part of this theorem. For instance, consider the case $[\beta_{\mathsf{inl}1}]$:

$$\nu x{:}A \oplus B.(\mathsf{inl}\ x;\ w.\mathtt{P} \parallel \mathsf{case}\ x\ (w.\mathtt{Q}_1,\ w.\mathtt{Q}_2)) \Rightarrow_{SCP} \nu w{:}A.(\mathtt{P} \parallel \mathtt{Q}_1)$$

To show that linearity of some free channel z is preserved under this reduction, we must check for the case where z appears in the left process or in the right process by pattern matching on the linearity assumption.

```
rec lin_s : (Γ : ctx) [Γ, x:name, h:hyp x A[] ⊢ wtp P[..,x] ]
                    → [Γ, x:name ⊢ step P Q]
...
=
/ total 2 /
fn tpP ⇒ fn sPQ ⇒ fn linP ⇒
case sPQ of
...
| [g, z:name ⊢ βinl1] ⇒
  (case linP of
% z appears on the left - the linearity must be the congruence case for inl
  | [g ⊢ l_pcomp1 (λx. l_inl2 (λw.linP'))] ⇒
    [g ⊢ l_pcomp1 (λw. linP'[..,w,w])]
% z appears on the right - the linearity must be the congruence case for the 'case' construct
  | [g ⊢ l_pcomp2 (λx. l_choice2 (λw. linP') (λw._))] ⇒
    [g ⊢ l_pcomp2 (λw. linP'[..,w,w])]
  )
```

The first `w` in the substitution `linP '[..,w,w]` correspond to substituting w for x, which may seem like a violation of linearity. However, for well-typed processes, the linearity predicate for x will ensure that x is no longer used in the inner process, meaning this substitution does not lead to duplication of w and is safe.

The implementation for `wtp_s` is mostly bureaucratic and involves using many of the prior strengthening lemmas to ensure that the communicated channel x can be safely removed from the context.

One interesting observation is that although preservation of typing does not require any assumptions about linearity, preservation of linearity does require the assumption that the original process is well-typed. This is primarily due to the reduction rule $[\beta_{\mathtt{fwd}}]$:

$$vx{:}A.(\mathtt{fwd}\ x\ y \parallel \mathtt{Q}) \Rightarrow_{SCP} [y/x]\mathtt{Q}$$

Here, if we want to show that the linearity of channel y is preserved, we need to know that `Q` treats x linearly, or $\mathsf{lin}(x,\ \mathtt{Q})$. We can only obtain this from the assumption that the original process is well-typed since x in process `Q` is not a continuation channel of y in `P`.

7 RELATED WORK

The linearity predicate that we develop in this paper is based on Crary's mechanization of the linear λ-calculus in Twelf [Crary 2010]. Adapting his ideas to the session-typed setting was non-trivial due to the many differences between the two systems, such as channel mobility, the distinction between names and processes, and continuation channels. Our bijection proof between CP and SCP is similar to Crary's adequacy proof of his encoding, where he showed that typing derivations of linear λ-calculus expressions were in bijection with typing derivations in the encoding alongside a proof of linearity for each free variable. Indeed, this side condition is analogous to our criterion that $\mathsf{lin}(\Delta,\ \mathtt{P})$.

7.1 HOAS Mechanizations

Röckl, Hirschkoff, and Berghofer [Röckl et al. 2001] encode the untyped π-calculus in Isabelle/HOL and prove that their encoding is adequate. Much of their technical development concerns eliminating *exotic terms*. To do so, they introduce local well-formedness conditions, similar in spirit to how we use the linearity predicates to eliminate non-linear processes. In LF, such exotic terms do not typically arise, as there is a bijection between the canonical representation in LF and its on-paper counterpart. Moreover, they do not encode any process reductions or mechanize any metatheorems.

Despeyroux [2000] gives a HOAS encoding of a typed π-calculus in Coq and uses it to mechanize a proof of subject reduction. This encoding is less involved than ours because their type system is very simple and, in particular, does not involve linearity. Thus, they did not need to account for complex operations on contexts. Furthermore, they do not discuss the adequacy of the encoding.

Tiu and Miller [2010] give a weak HOAS encoding of the finite π-calculus together with its operational semantics using the late transition system within a logic that contains the ∇ quantifier for encoding generic judgments and definitions. They then specify a bisimulation for late transition systems and show that it is reflexive and transitive. Tiu and Miller prove that their encoding is adequate. However, their system does need to deal with linearity and is also not typed and hence does not face the same challenges as ours.

The closest existing literature to our work is by Zalakain [2019], who uses parametric HOAS [Chlipala 2008] to mechanize a session-typed process calculus in Coq. They use a global linearity predicate as a well-formedness condition and directly encode the $x \notin \mathrm{fn}(P)$ style side conditions as a predicate. They further prove that linearity is preserved under all reductions except those using the structural

equivalence $P \mid Q \equiv Q \mid P$, which corresponds to $[\equiv_{\mathtt{comm}}]$ in our setting. This equivalence is problematic in their setting because of interactions between their linearity predicate, scope expansion, and parallel composition. They do not discuss the adequacy of their encoding. We instead localize the linearity predicates within type judgments and leverage higher-order encoding to obtain some side conditions "for free". As in their setting, we prove subjection reduction for linearity but also for typing, obtaining the usual type preservation result. Furthermore, the structural equivalence rule $\nu x{:}A.(\mathtt{P} \parallel \mathtt{Q}) \equiv \nu x{:}A^{\perp}.(\mathtt{Q} \parallel \mathtt{P})$ presents no notable difficulties in our setting.

7.2 Other Approaches to Mechanizing Session Types and Typed Process Calculi

Gay [2001] uses Isabelle/HOL to give one of the first mechanizations of a linearly typed process calculus and its reduction relation. Bindings are handled via de Bruijn indexing and linearity is enforced by modeling a linear context with relevant operations. Interestingly, he does not directly encode processes in Isabelle/HOL. Instead, he mechanizes a λ-calculus with constants as a metalanguage and then encodes channel bindings in the process calculus through λ-abstractions in the metalanguage in a HOAS-like manner.

Thiemann [2019] mechanizes a functional language with session-typed communication in Agda. He too uses de Bruijn indexing to handle binding and directly implements linear contexts. The system is intrinsically typed, meaning subject reduction is obtained "for free". However, the encoding is operational in nature, and for example, the operational semantics depends on a "scheduler" that globally identifies channels and performs communication. Showing adequacy of the encoding is therefore quite complicated because of the disconnect between the on-paper theory and the actual implementation, which the author mentions.

Zalakain and Dardha model contexts using leftover typing in Agda [Zalakain and Dardha 2021]. This technique avoids context splits by modifying type judgments to add an additional output context, making explicit what resources are not used by a given process in a type judgment. However, their approach still requires proving certain metatheorems about their leftover typing and still embeds some form of linearity. It is therefore not well-suited for a HOAS-style encoding in LF, although it is less clear what are the trade-offs between their approach and our approach in non-HOAS settings. They also make no mention of adequacy.

Castro-Perez, Ferreira, and Yoshida [Castro-Perez et al. 2020] use a locally nameless representation to develop a general framework of mechanizing session-typed process calculi in Coq. They observe that a naïve usage of locally nameless representations cannot handle higher-order communication, i.e., channel transmission. To encode such communications, they employ a strategy to syntactically distinguish between different forms of channel bindings, working with four sets of channel names. Our approach encodes all forms of channel bindings via intuitionistic functions over the same set of names in LF and handles higher-order communication.

7.3 HOAS with Linearity

Perhaps one natural approach to a HOAS encoding of a linear system like session types is to use a logical framework with direct support for linear implications. Unfortunately, these systems are far less understood, and implementations of such systems are often preliminary.

Concurrent LF [Schack-Nielsen and Schürmann 2008] is an extension of the logical framework LF to support the specification of linear and even concurrent formal systems. Its implementation, Celf, has been used to encode systems such as the untyped π-calculus [Cervesato et al. 2002]. Although encoding a session-typed system certainly seems plausible in Celf, it remains unclear how to encode metatheoretic proofs such as subject reduction.

LINCX [Georges et al. 2017] is a proof environment that follows in the footsteps of Beluga. Instead of specifying formal systems in LF as in Beluga, one specifies formal systems in linear LF in

LINCX. Metatheoretic proofs are then implemented as recursive functions over linear contextual objects. This framework should in principle be capable of representing session-type systems and their metatheory more directly, but there is presently no implementation for it.

Linear Hybrid [Felty 2019; Felty et al. 2021] is designed to support the use of higher-order abstract syntax for representing and reasoning about formal systems, and it is implemented in the Coq Proof Assistant. To support representation of linear systems it implements a linear specification logic in Coq. Felty and collaborators have used this framework to, for example, encode the type system of a quantum λ-calculus with linear typing and its metatheoretic properties. It would be interesting to see how to use this framework to specify session types together with their metatheory.

8 CONCLUSION

We demonstrate a higher-order encoding and mechanization of CP, a session-typed process calculus. Our main technique is using linearity predicates that act as well-formedness conditions on processes. In particular, this lets us encode linearity without relying on linear contexts which are difficult to work with in mechanizations and which are not well-suited for HOAS-style encodings. We decomposed our encoding in two steps: an on-paper formulation of SCP using linearity predicates, and a mechanization of SCP in Beluga.

Our development of SCP, which arose as a byproduct of our mechanization, provides a foundation for mechanizing session-typed process calculi in settings with structural contexts. We prove that CP is fully embedded in SCP and furthermore, that the restriction imposed by the linearity predicates captures the fragment of SCP that correspond to CP. More precisely, we prove that there is a structure-preserving bijection between the processes and typing derivations in CP and those in SCP when we subject SCP to the condition that it treats its free names linearly.

We then mechanize SCP in Beluga and prove the adequacy of our encoding, thereby showing that our encoding is adequate with respect to CP. As we demonstrate through our mechanization, SCP particularly synergizes with a HOAS encoding over Beluga, which utilizes contextual type theory, allowing for side-conditions related to free names to be encoded "for free".

In general however, using an SCP-like presentation has the benefit of using intuitionistic contexts, which are better understood and easier to work with in proof assistants. Whether the encoding style implicitly uses an intuitionistic context like for LF is not particularly important; even an encoding style that explicitly models a context can benefit from this approach. Our development of SCP shows how to shift the work required for linear context management to local side conditions, or linearity predicates, which we believe leads to a more tractable way to both encode and reason with linearity. Although our approach is certainly heavily inspired by the constraints imposed by LF and HOAS, SCP is still a promising system to mechanize over CP using other proof assistants and encoding styles such as de Bruijn or locally nameless. In particular, Zalakain's encoding [Zalakain 2019] of a similar session-typed system using parametric HOAS gives strong evidence that an SCP-style calculus extends well to Coq.

It is however important to acknowledge that this approach comes at the cost of managing linearity predicates and free names in processes. Although these were easy to work with in our setting (in particular, managing free names was obtained for free from higher-order unification), it would be interesting to understand more clearly the costs and benefits from the additional side conditions compared to dealing with linear contexts in the context of other proof assistants and encoding styles.

8.1 Towards More Complex Language Constructs

We illustrated how linearity predicates could be used to mechanize a fragment of Wadler's CP [Wadler 2012], and it is natural to ask whether this technique scales to the full system. It

is also natural to ask whether this technique scales to more complex extensions of session-typed systems, such as notions of sharing [Balzer and Pfenning 2017; Rocha and Caires 2021], equi-recursion [Gay and Hole 2005], and integrations with functional languages [Gay and Vasconcelos 2010; Toninho et al. 2013]. We believe that linearity predicates are a mechanization technique that is sufficiently robust and scalable to handle these richer language constructs. To guide future applications of our approach, we sketch the key patterns and principles for its application to new program constructs:

(1) Determine if the construct binds any new linear channels. If so, then its typing judgments must check their linearity. In our development, this is illustrated by the typing rules `[⅋]`, `[⊗]`, and `[Cut]`.
(2) Determine if the construct requires the absence of other linear assumptions. If so, then there should be no congruence rules for the linearity predicate. In our development, this is illustrated by the linearity predicates for close x and fwd $x\ y$.
(3) Determine if the construct uses a continuation channel. If so, then the linearity predicate should check that the continuation channel is used linearly. Otherwise, the linearity predicate should be an axiom. These two cases are respectively illustrated by $L_{\texttt{inl}}$ and $L_{\texttt{wait}}$.
(4) Determine if linear channels are shared between subterms composed by the construct. If they are not shared, then the linearity predicate must ensure that no sharing occurs. This is illustrated by $L_{\nu 1}$ and $L_{\nu 2}$.

With regard to extending our mechanization to the entirety of CP, we believe that its polymorphic constructors $\forall$ and $\exists$ will pose no technical challenges. Indeed, they operationally correspond to receiving and sending types, and types are treated in an unrestricted manner. Therefore, they do not interact with linearity in an interesting way.

However, the exponentials ! and ? may be more challenging to mechanize. Channels of type $?A$ are not treated linearly: they may be dropped or copied. Intuitively, this means that we should *not* check for linearity of channels of type $?A$. In Crary's encoding of the linear λ-calculus, there was only one syntactical construct that bound assumptions of type $?\tau$, making this easy to do. In contrast, CP channels of type $?A$ can arise from many sources, such as inputs from channels of form $(?A) \parr B$, as channel continuations of any connective such as $?A \oplus ?B$. This means that we cannot determine solely from the syntax of processes whether a bound channel is of type $?A$. However, we only ever use the linearity predicate to check the linearity of channels whose type is known. We believe that by using this type information and by making the linearity predicate type aware, i.e., of the form $\mathsf{lin}(x{:}A,\ \mathtt{P})$, we can give a sufficiently refined analysis of linearity to support channels of type $?A$.

8.2 Future Work

Our work lays the groundwork for two main directions of future work. The first is to explore the trade-offs encountered when encoding SCP in various proof assistants and mechanization styles. Given that SCP was designed with an LF encoding in mind, it is not entirely clear whether the overhead of linearity predicates and free name conditions is offset by the advantages of working with unrestricted contexts in other settings. Nevertheless, we believe that SCP provides a scalable basis for mechanizations with proofs of adequacy in mind.

The second direction is to extend SCP and its encoding to better understand the scalability of our technique. Although we sketched the general roadmap for such extensions, it is interesting to verify that our technique is indeed scalable and to also understand its limitations. Mechanizing metatheory beyond subject reduction will further elucidate our technique's scalability. For example, we believe that our linearity predicate will be essential to mechanizing a progress theorem for

SCP processes. Progress for SCP processes corresponds to top-level cut elimination. Well-typed linear SCP processes support top-level cut elimination by their correspondence with CP processes (Theorem 3.4) and the fact that CP processes enjoy this same property. This indirect proof sketch is similar to our indirect proof of subject reduction (Theorem 3.6). A direct proof of progress is a natural next metatheorem to mechanize and, based on our preliminary investigations, seems to be relatively straightforward.

DATA-AVAILABILITY STATEMENT

The software containing the encoding of SCP (Section 4) and mechanization of the subject reduction proof (Section 6) is available on Zenodo [?].

ACKNOWLEDGMENTS

This work was funded by the Natural Sciences and Engineering Research Council of Canada (grant number 206263), Fonds de recherche du Québec - Nature et Technologies (grant number 253521), a Tomlinson Doctoral Fellowship awarded to the first author, and Postdoctoral Fellowship from Natural Sciences and Engineering Research Council of Canada awarded to the second author.

We also thank the anonymous reviewers for their valuable comments and feedback.

REFERENCES

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. In *International Conference on Functional Programming (ICFP)*. ACM, 37:1–37:29. Extended version available as Technical Report CMU-CS-17-106R, June 2017.

David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12079)*, Armin Biere and David Parker (Eds.). Springer, 278–285. https://doi.org/10.1007/978-3-030-45237-7_17

Andrew Cave and Brigitte Pientka. 2012. Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. 413–424.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. 2002. *A Concurrent Logical Framework II: Examples and Applications.* Technical Report CMU-CS-02-102. Department of Computer Science, Carnegie Mellon University. Revised May 2003.

Adam J. Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, James Hook and Peter Thiemann (Eds.). ACM, 143–156.

Karl Crary. 2010. Higher-order Representation of Substructural Logics. In *Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)*, P.Hudak and S.Weirich (Eds.). ACM, Baltimore, Maryland, 131–142.

Joëlle Despeyroux. 2000. A Higher-Order Specification of the π-Calculus. In *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–439.

Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. 1995. Higher-Order Abstract Syntax in Coq. In *2nd International Conference on Typed Lambda Calculi and Applications (TLCA '95) (Lecture Notes in Computer Science (LNCS 902))*, Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin (Eds.). Springer, 124–138. https://doi.org/10.1007/BFb0014049

Amy P. Felty. 2019. A Linear Logical Framework in Hybrid (Invited Talk). In *FSCD (LIPIcs, Vol. 131)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:2.

Amy P. Felty, Carlos Olarte, and Bruno Xavier. 2021. A focused linear logical framework and its application to metatheory of object logics. *Math. Struct. Comput. Sci.* 31, 3 (2021), 312–340.

Simon J. Gay. 2001. A Framework for the Formalisation of Pi Calculus Type Systems in Isabelle/HOL. In *International Conference on Theorem Proving in Higher Order Logics.*

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-Calculus. *Acta Informatica* 42, 2–3 (2005), 191–225.

Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming* 20, 1 (Jan. 2010), 19–50.

Aina Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. 2017. LINCX: A Linear Logical Framework with First-Class Contexts. In *26th European Symposium on Programming (ESOP 2017) (Lecture Notes in Computer Science (LNCS*

20201), Hongseok Yang (Ed.). 530–555. https://doi.org/10.1007/978-3-662-54434-1_20

Robert Harper, Dan Licata, William Lovas, Chris Martens, and Robert Simmons. 2009. POPL Tutorial: Mechanizing Metatheory with LF and Twelf. http://twelf.org/wiki/POPL_Tutorial/Saturday

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR 1993)*, E. Best (Ed.). Springer LNCS 715, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, C. Hankin (Ed.). Springer LNCS 1381, 122–138.

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 1 (Jan. 2022), 33 pages. https://doi.org/10.1145/3498662

Robin Milner. 1980. *A Calculus of Communicating Systems.* Springer-Verlag LNCS 92.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *Transactions on Computational Logic* 9, 3 (2008).

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation.* Atlanta, Georgia, 199–208.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08).* 371–382.

Brigitte Pientka and Jana Dunfield. 2008. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08).* 163–173.

Brigitte Pientka and Jana Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description), Vol. 6173. 15–21. https://doi.org/10.1007/978-3-642-14203-1_2

Pedro Rocha and Luís Caires. 2021. Propositions-as-Types and Shared State. *Proc. ACM Program. Lang.* 5, ICFP, Article 79 (Aug. 2021), 30 pages. https://doi.org/10.1145/3473584

Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. 2001. Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the Pi-Calculus and Mechanizing the Theory of Contexts. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)*, F. Honsell and M. Miculan (Eds.). Springer Verlag LNCS 2030, Genova, Italy, 364–378.

Anders Schack-Nielsen and Carsten Schürmann. 2008. Celf - A Logical Framework for Deductive and Concurrent Systems (System Description). In *IJCAR (Lecture Notes in Computer Science, Vol. 5195).* Springer, 320–326.

Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP '19).* Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. https://doi.org/10.1145/3354166.3354184

Alwen Tiu and Dale Miller. 2010. Proof search specifications of bisimulation and modal logics for the pi-calculus. *ACM Trans. Comput. Log.* 11, 2 (2010), 13:1–13:35.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Proceedings of the European Symposium on Programming (ESOP'13)*, M. Felleisen and P. Gardner (Eds.). Springer LNCS 7792, Rome, Italy, 350–369.

Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012).* ACM Press, Copenhagen, Denmark, 273–286.

Uma Zalakain. 2019. *Type-checking session-typed π-calculus with Coq.* Masters Thesis. University of Glasgow. https://www.dcs.gla.ac.uk/~ornela/projects/Uma%20Zalakain.pdf

Uma Zalakain and Ornela Dardha. 2021. π with Leftovers: A Mechanisation in Agda. In *Formal Techniques for Distributed Objects, Components, and Systems*, Kirstin Peters and Tim A. C. Willemse (Eds.). Springer International Publishing, Cham, 157–174.

Received 2023-04-14; accepted 2023-08-27