Thesis Proposal: A Logical Foundation for Session-based Concurrent Computation

Bernardo Toninho
March 30, 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Co-Chair
Luis Caires (New University of Lisbon), Co-Chair
Robert Harper,
Stephen Brooks,
Simon Gay (University of Glasgow),
Vasco Vasconcelos (University of Lisbon)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Keywords: Session Types, Linear Logic, Type Theory, Proofs as Programs
Abstract

Linear logic has long been heralded for its potential of providing a logical basis for concurrency. While over the years many research attempts were made in this regard, a Curry-Howard correspondence between linear logic and concurrent computation was only found recently, bridging the proof theory of linear logic and session-typed process calculus. Building upon this work, we have developed the theory of intuitionistic linear logic as a logical foundation for session-based concurrency, exploring the logical meanings of several concurrency-theoretic phenomena. I propose to show that intuitionistic linear logic is indeed a suitable logical foundation for message-passing concurrency by going beyond the initial interpretation of propositional linear logic as session types and providing an elegant framework capable of expressing and reasoning about several naturally occurring phenomena in such a concurrent setting.
1 Introduction

Over the years, computation systems have evolved from monolithic single-threaded machines to concurrent and distributed environments with multiple communicating threads of execution, for which writing correct programs becomes substantially harder than in the more traditional sequential setting. These difficulties arise as a result of several issues, fundamental to the nature of concurrent and distributed programming, such as: the many possible interleavings of executions, making programs hard to test and debug; resource management, since often concurrent programs must interact with (local or remote) resources in an orderly fashion, which inherently introduce constraints in the level of concurrency and parallelism a program can have; and coordination, since the multiple execution flows are intended to work together to produce some ultimate goal or result, and therefore must proceed in a coordinated effort.

Concurrency theory often tackles these challenges through abstract language-based models of concurrency, such as process calculi, which allow for reasoning about concurrent computation in a precise way, enabling the study of the behavior and interactions of complex concurrent systems. Much like the history of research surrounding the $\lambda$-calculus, a significant research effort has been made to develop type systems for concurrent calculi (the most pervasive being the $\pi$-calculus) that, by disciplining concurrency, impose desirable properties on well-typed programs such as deadlock-freedom or absence of race conditions.

However, unlike the (typed) $\lambda$-calculus which has been known to have a deep connection with intuitionistic logic, commonly known as the Curry-Howard correspondence, no equivalent connection was established between the $\pi$-calculus and logic until quite recently. One may then wonder why is such a connection important, given that process calculi have been studied since the early 1980s, with the $\pi$-calculus being established as the lingua franca of interleaving concurrency calculi in the early 1990s, and given the extensive body of work that has been developed based on these calculi, despite the lack of foundations based on logic.

The benefits of a logical foundation for interleaving concurrency in the style of Curry-Howard are various: on one hand, it entails a form of canonicity of the considered calculus by connecting it with proof theory and logic. Pragmatically, a logical foundation opens up the possibility of employing established techniques from logic to the field of concurrency theory, potentially providing elegant new means of reasoning and representing concurrent phenomena. Fundamentally, a logical foundation allows for a compositional and incremental study of new language features, since the grounding in logic ensures that such extensions do not harm previously obtained results.

Proposed work

My thesis work will defend the following statement:

**Thesis Statement:** Linear logic, specifically in its intuitionistic formulation, is a suitable logical foundation for message-passing concurrent computation, providing an elegant framework in which to express and reason about a multitude of naturally occurring phenomena in such a concurrent setting.

The structure of this proposal is as follows: Section 2 discusses the basic concepts of message-passing concurrency and the existing techniques to reason and discipline this form of concurrency; Section 3 discusses the basic interpretation of intuitionistic linear logic as a language of (binary) session types and its connections to session-typed $\pi$-calculus; Section 4 explores the linear logic foundation by giving an account of (value) dependent session types and parametric polymorphism as extensions to the foundation of Section 3 that arise naturally from logical considerations; Section 5 develops a more concrete programming language based on the logical interpretation, combining functional programs with concurrent session-typed process expressions through a contextual monad and including general recursive types; Section 6 makes full use the logical foundation to develop reasoning techniques for concurrent computation, relating them with known techniques from the concurrency community.

In the remainder of this section I outline my contributions in more detail.

**A logical foundation for session-based concurrency** My work builds on the foundational work of [5] connecting linear logic and session types, using it as a basis to explain a variety of phenomena that arise naturally in message-passing concurrency, developing the idea of using linear logic as a logical foundation for message-passing concurrent computation in general. To support this claim, the interpretation of [5] must be able to scale beyond the confines of...
simple session types (and even the $\pi$-calculus), being able to also account for richer and more sophisticated phenomena such as dependent session types \cite{50}, parametric polymorphism \cite{7}, among others I do not develop in this document (such as concurrent evaluation strategies \cite{51} and asynchronous communication \cite{12}), all the while doing so in a technically elegant way. While claims of elegance are by their very nature subjective, I believe it is possible to support these claims by showing how naturally and easily the interpretation can account for these phenomena that traditionally require very sophisticated technical devices.

In Section 3 I develop the interpretation of linear logic as session types that serves as the basis for my work. It has a few variations from that of \cite{5} in that it does not commit to the $\pi$-calculus \textit{a priori}, developing a proof term assignment that can be used as a language for session-typed communication, but also maintaining its connections to the $\pi$-calculus. The goal of this “dual” development, amongst other more technical considerations, is to further emphasize that the connection of linear logic and session-typed concurrency goes beyond that of having a $\pi$-calculus syntax and semantics. While we may use the $\pi$-calculus assignment when it is convenient to do so (and in fact I do precisely this in Section 6), we are not necessarily tied to the $\pi$-calculus. This adds additional flexibility to the interpretation since it enables a sort of “back and forth” reasoning between $\pi$-calculus and the more faithful proof term assignment that is quite natural in Curry-Howard correspondences.

In Section 4 I address the claims of being able to scale the interpretation to explain a range of relevant phenomena by developing the two particular cases of value dependent types and parametric polymorphism as they arise by considering first and second-order intuitionistic linear logic, respectively. I chose these two specific cases since they are perhaps the most natural extensions to the interpretation that provide a previously unattainable degree of expressiveness. However, the interpretation extends further beyond these two settings. For instance, in \cite{51} the interpretation is used to give a logically motivated account of parallel evaluation strategies on $\lambda$-terms through canonical embeddings of intuitionistic logic in linear logic. One remarkable result is that the resulting embeddings induce a form of sharing (as in futures, relaxing the sequentiality constraints of call-by-value and call-by-need) and copying (as in call-by-name) parallel evaluation strategies on $\lambda$-terms, the latter being reminiscent of Milner’s original embedding of the $\lambda$-calculus in the $\pi$-calculus \cite{31}.

A concurrent programming language A logical foundation must also provide the means of expressing concurrent computation in a natural way that preserves the good properties one obtains from logic. To solidify this point I develop the basis of a concurrent programming language (Section 5) that combines functional and session-based concurrent programs via a monadic embedding of session-typed process expressions in a $\lambda$-calculus. One interesting consequence of this embedding is that it allows for process expressions to send, receive and execute other process expressions, in the sense of higher-order processes \cite{44}, while preserving the property of deadlock-freedom by typing.

For practical purposes, I add recursive types to the language. This breaks the tight connection with logic since it introduces potentially divergent computations, but allows me to showcase additionally interesting programs. To recover the connections with logic I restrict general recursive types to inductive and coinductive types and ensure non-divergence of computation through the introduction of syntactic restrictions on recursive process definitions, similar to those used in dependently typed programming languages with recursion such as Coq \cite{49} and Agda \cite{35}, but with additional subtleties due to the concurrent nature of the language.

Reasoning techniques Another fundamental strength of a logical foundation for concurrent computation is its ability to provide both the ability to express and also reason about such computations. To this end I develop a theory of \textit{linear logical relations} on the $\pi$-calculus assignment for the interpretation, showing how it can be applied to the polymorphic language of Section 4 to develop termination and parametricity results, and for the logically justifiable fragment of the monadic language of Section 5 to prove termination. As a consequence of the linear logical relations on the polymorphic setting I obtain an equivalence relation that is sound and complete with respect to the traditional process calculus equivalence of (typed) \textit{barbed congruence}. Finally, I conjecture on how this equivalence may be extended to the monadic setting, providing a notion of extensional equality for a full dependently typed language.
Concurrency is often divided into two models, shared memory concurrency and message-passing concurrency. The former enables communication between concurrent entities through modification of memory locations that are shared between the entities; whereas in the latter the various concurrently executing components communicate by exchanging messages, either synchronously or asynchronously.

Understanding and reasoning about concurrency is ongoing work in the research community. In particular, many language based techniques have been developed over the years for both models of concurrency. For shared memory concurrency, the premier techniques are those related to (concurrent) separation logic [36], which enables formal reasoning about memory configurations that may be shared between multiple entities. For message-passing concurrency, which is the focus of this work, the most well developed and studied techniques are arguably those of process calculi [9, 44]. Process calculi are a family of formal language that enable the precise description of concurrent systems, by modelling interaction through communication across abstract communication channels. An important and appealing feature of process calculi is their algebraic nature: processes can be manipulated via certain algebraic laws, which also allow for formal reasoning about behavioral equivalence [45, 47, 30].

While many such calculi have been developed over the years, the de facto standard process calculus is the π-calculus [44, 32], a language that allows modelling of concurrent systems that communicate over channels that may themselves be generated dynamically and passed in communication. Typically, the π-calculus also includes replication (i.e. the ability to spawn an arbitrary number of parallel copies of a process), which combined with channel generation and passing makes the language a Turing complete model of concurrent, message-passing computation. The syntax of a π-calculus is given in Fig. 1 (for purposes that will be made clear in Section 3, we extend the typical π-calculus syntax with a linear forwarder \( \langle x \leftrightarrow y \rangle \) reminiscent of the forwarders used in the internal mobility π-calculus [43]). The so-called static fragment of the π-calculus consists of the inactive process \( 0 \), the parallel composition operator \( P \parallel Q \) and the scope restriction \( (\nu y) P \), binding \( y \) in \( P \). The communication primitives are the output and input prefixed processes, \( !x(y).P \) and \( x(y).P \) which send and receive along channel \( x \), respectively, with \( y \) bound in \( P \) in the latter. We restrict general replication to an input guarded form \( !x(y).P \), denoting a process that waits for inputs on \( x \) and subsequently spawns a replica of \( P \). The channel forwarding construct \( [x \leftrightarrow y] \) equates the two channel names \( x \) and \( y \). We also consider (binary) choice \( x.\text{case}(P, Q) \) and the two corresponding selection constructs \( x.\text{inl}; P \) and \( x.\text{inr}; P \).

As in the history of the λ-calculus, many type systems for the π-calculus have been developed over the years. The goal of these type systems is to discipline communication in some way as to avoid certain kinds of errors. While the early type systems for the π-calculus focused on assigning types to values communicated on channels (e.g. the type of channel \( c \) states that only integers can be communicated along \( c \)), and on assigning input and output capabilities to channels (e.g. process \( P \) can only send integers on channel \( c \) and process \( Q \) can only receive), arguably the most important type systems developed for the π-calculus are session types.

2.1 Binary Session Types

The core idea of session types is to structure communication between processes around the concept of a session. A (binary) session consists of a description of the interactive behavior between two components of a concurrent system, with an intrinsic notion of duality: When one component sends, the other receives; when one component offers a choice, the other chooses. Another crucial point is that a session is stateful insofar as it is meant to evolve over time (e.g. “input an integer and afterwards output a string”) until all its codified behavior is carried out. A session type [26, 28] codifies the intended session that must take place over a given communication channel at the type level, equating type checking with a high-level form of communication protocol compliance checking.
The communication idioms that are typically captured in session types are input and output behavior, choice and selection, replication and recursive behavior. It is also common for session type systems to have a form of session delegation (i.e. delegating a session to another process via communication). Moreover, session types can provide additional guarantees on system behavior than just adhering to the ascribed sessions, such as deadlock absence and liveness [13]. We present a syntax for session types in Fig. 2, as well as the intended meaning of each type constructor. We use a slightly different syntax than that of the original literature on session types [26, 28], which conveniently matches the syntax of propositions in (intuitionistic) linear logic. We make this connection precise in Section 3.

While session types are indeed a powerful tool for the structuring of communication-centric programs, their theory is fairly complex, especially in those systems that guarantee the absence of deadlocks, which require sophisticated causality tracking mechanisms. Another issue that arises in the original theory of session types is that it is often unclear how to consider new primitives or language features without harming the previously established soundness results. Moreover, the behavioral theory (in the sense of behavioral equivalence) of session typed systems is also quite intricate, requiring sophisticated bisimilarities which are hard to reason about, and from a technical standpoint, for which establishing the desirable property of contextuality or congruence (i.e. equivalence in any context) is often a challenging endeavour.

3 Linear Logic and Session Types

In the concurrency theory community, linearity has played a key role in the development of typing systems for the π-calculus. Linearity was used in early type systems for the π-calculus [29] as a way of ensuring certain desirable properties. Ideas from linear logic also played a key role in the development of session types, as acknowledged by Honda [26, 28].

Girard’s linear logic [22] arises as an effort to marry the dualities of classical logic and the constructive nature of intuitionistic logic by rejecting the so-called structural laws of weakening (“If I assume something, I can assume it multiple times”) and contraction (“I need not use all assumptions in a proof”). Proof theoretically, this simple restriction turns out to have profound consequences in the meaning of logical connectives. Moreover, the resulting logic is one where assumptions are no longer persistent immutable objects but rather resources that interact, transform and are consumed during inference. Linear logic divides conjunction into two forms, which in linear logic terminology are called additive (usually dubbed “with”, written &) and multiplicative (dubbed “tensor”, and written ⊗), depending on how resources are used to prove the conjunction. Additive conjunction denotes a pair of resources where one must choose which of the elements of the pair one will use, although the available resources must be able to realize (i.e. prove) both elements. Multiplicative conjunction denotes a pair of resources where both resources must be used, since the available resources simultaneously realize both elements and all resources must be consumed in a valid inference (due to the absence of weakening and contraction). In classical linear logic there is also a similar separation in disjunction, while in the intuitionistic setting we only have additive disjunction ⊕ which denotes an alternative between two resources (we defer from a precise formulation of linear logic for now).

The idea of propositions as mutable resources that evolve independently over the course of logical inference sparked interest in using linear logic as a logic of concurrent computation. This idea was first explored in the work of Abramsky et. al [11], which developed a computational interpretation of linear logic proofs, identifying them with programs in a linear λ-calculus with parallel composition. In his work, Abramsky gives a faithful proof term assignment to classical linear logic sequent calculus, identifying proof composition with parallel composition. This proofs-as-
processes interpretation was further refined by Bellin and Scott [2], which mapped classical linear logic proofs to processes in the synchronous \( \pi \)-calculus with prefix commutation. Similar efforts were developed more recently in [27], connecting polarised proof-nets with an (IO) typed \( \pi \)-calculus, and in [14] which develops ludics as a model for the finitary linear \( \pi \)-calculus. However, none of these interpretations provided a true Curry-Howard correspondence insofar as they either did not identify a type system for which linear logic propositions served as type constructors (lacking the propositions-as-types part of the correspondence); or develop a connection with only a very particular formulation of linear logic (such as polarised proof-nets).

Given the predominant role of session types as a typing system for the \( \pi \)-calculus and their inherent notion of evolving state, in hindsight, its connections with linear logic seem almost inescapable. However, it wasn’t until the recent work of Caires and Pfenning [5] that this connection was made precise. The work of [5] develops a Curry-Howard correspondence between session types and intuitionistic linear logic, identifying (typed) processes with proofs, session types with linear logic propositions and process reduction with proof reduction. I refrain from a full presentation of their work here, opting for a slightly different presentation of this interpretation for both stylistic purposes and to address a few shortcomings of the original interpretation given in [5].

First, I use linear forwarders as an explicit proof term for the identity rule, which is not addressed in [5] (such a construct was first proposed for the interpretation in [50]). Secondly, I use a different assignment for the exponentials of linear logic (in line with [12, 53]), which matches proof reductions with process reductions in a more faithful way. The final and perhaps most important distinction is that the rules I present are all syntax driven, not using the \( \pi \)-calculus proof term assignment outright. My proof term assignment forms a basis for a concurrent, session-typed language that can be consistently mapped to session-typed \( \pi \)-calculus processes in a way that is consistent with the interpretation of [5], but that does not require structural congruence for typing nor for the operational semantics, by using a technique called substructural operational semantics [39] (SSOS in the sequel). I refer to these proof terms as process expressions, in opposition to \( \pi \)-calculus terms which I refer to as processes. Thus, my assignment does not require the explicit representation of the \( \pi \)-calculus \( \nu \)-binder in the syntax, nor the explicit commutativity of parallel composition in the operational semantics, which are artefacts that permeate the foundational work of [5] and so require the extensive use of structural congruence of process terms for technical reasons. However, there are situations where using \( \pi \)-calculus terms and structural congruence turn out to be technically convenient, and abandoning the \( \pi \)-calculus outright would diminish the claims of providing a true logical foundation of concurrent computation.

In my presentation I develop the proof term assignment in tandem with the \( \pi \)-calculus term assignment but without placing added emphasis on one or the other, enabling reasoning with techniques from proof theory and process calculi (and as we will see, combinations of both) and further emphasizing the back-and-forth from language to logic that pervades the works exploring logical correspondences in the sense of Curry-Howard. The presentation also explores the idea of identifying a process by the channel along which it offers its session behavior. This concept is quite important from a language design perspective (since it defines precise boundaries on the notion of a process), but is harder to justify in the \( \pi \)-calculus assignment. The identification of processes by a single channel further justifies the use of intuitionistic linear logic over classical linear logic, for which a correspondence with session types may also be developed (viz. [8, 55]), but where the classical nature of the system disallows such an identification. Moreover, in the following sections I go beyond the scope of [5, 8], fully exploiting the correspondence to give a logically grounded account of several other relevant phenomena in concurrent computation in a way that is fully compatible with the basis presented below.

Substructural Operational Semantics As mentioned above, I define the operational semantics of process expressions in the form of a substructural operational semantics (SSOS) [39]. For those unfamiliar with this style of presentation, it consists of a compositional specification of the operational semantics by defining a predicate on the expressions of the language, through rules akin to those of multiset rewriting [10], where the pattern to the left of the \( \rightsquigarrow \) arrow describes a state which is consumed and transformed into the one to the right. The SSOS framework allows us to generate fresh names through existential quantification and doesn’t require an explicit formulation of structural congruence, as one would expect when defining the typical operational semantics for process calculi-like languages.

For now, we rely on the following two predicates (these will be extended further as we proceed in the presentation): the linear proposition \( \text{exec} \ P \) denotes the state of a linear process expression \( P \) and \( !\text{exec} \ P \) denotes the state of a persistent process (which must always be a replicating input).
3.1 The Interpretation

I begin by defining the basic judgments used throughout this presentation. The intuitionistic linear logic judgment is written as:

\[ \Gamma; \Delta \vdash A \]

where \( A \) is a linear logic proposition (whose syntax conveniently matches Fig. [2]), \( \Delta \) is the linear context region (assumptions not subject to contraction or weakening), and \( \Gamma \) the unrestricted or exponential context region (subject to both contraction and weakening). I write \( \Gamma; \Delta \vdash P :: z : A \) to denote that \( P \) is a proof term for \( z : A \), or in terminology of session types, process expression \( P \) offers along the distinguished channel \( z \) the session behavior \( A \). In the process expression assignment I consistently label assumptions in \( \Gamma \) and \( \Delta \) with channel names (pairwise disjoint). Finally, I use a different turnstyle \( \Gamma; \Delta \Rightarrow P :: z : A \) to distinguish between the process expression and \( \pi \)-calculus term assignments (we use \( P \) interchangeably for \( \pi \)-calculus processes and proof terms, distinguishing between the two either by syntax or by the particular judgment used). In this development I introduce the connectives using a sequent calculus, made up of so-called right rules which define how to prove (or offer) a connective (or session), and left rules which define how to use a connective.

I now go over the linear logic connectives and their corresponding interpretation. I start with the basic judgmental principles of logic, namely the cut and identity principles (additional judgmental principles are required to justify the exponential \( ! \), but we leave those for the interpretation of the exponential). In linear logic a cut consists of a principle of proof composition, where some resources are used to prove a proposition \( A \) (also known as the cut formula) and the remainder are used to prove some proposition \( C \) under the assumption that a proof of \( A \) exists:

\[ \Gamma; \Delta_1 \vdash A \quad \Gamma; \Delta_2 \vdash C \quad \Gamma; \Delta_1, \Delta_2 \vdash C \quad \text{(CUT)} \]

At the level of process expressions and \( \pi \)-calculus processes, this is interpreted as a form of composition:

\[ \Gamma; \Delta_1 \vdash P :: x : A \quad \Gamma; \Delta_2, x : A \vdash Q :: z : C \quad \Gamma; \Delta_1, \Delta_2 \vdash \text{new } x. (P_x ; Q_x) :: z : C \quad \text{(CUT)} \]

Note how \( x \) is bound in \( Q \), since it is the only term which can actually make use of the session behavior \( A \) offered by \( P \). Operationally, we make use of the following SSOS rule to capture the parallel composition nature of a cut:

\[ \text{(cut) } \text{exec}(\text{new } x. (P_x ; Q_x)) \rightarrow \{ \exists x'. \text{exec}(P_{x'}) \otimes \text{exec}(Q_{x'}) \} \]

The rule transforms the process expression assigned to a cut into the parallel execution of the two underlying process expressions, sharing a fresh name \( x' \) generated by the existential quantifier. In the \( \pi \)-calculus this corresponds to parallel composition plus name restriction:

\[ \Gamma; \Delta_1 \vdash P :: x : A \quad \Gamma; \Delta_2, x : A \vdash Q :: z : C \quad \Gamma; \Delta_1, \Delta_2 \vdash (\nu x)(P \mid Q) :: z : C \quad \text{(CUT)} \]

The \( \nu \)-binder for \( x \) is necessary to ensure that the channel \( x \) is shared only between \( P \) and \( Q \). To determine the correct operational behavior for our constructs we use as a guiding principle the computational content of the proof of cut elimination in linear logic. In particular, we consider the cut reduction steps when a right rule is cut with a left rule of a particular connective (also known as a principal cut).

The identity principle of linear logic states that we may use an assumption \( A \) to prove \( A \), provided it is our only remaining (linear) resource in order to ensure linearity:

\[ \Gamma; A \vdash A \quad \text{(ID)} \]

To develop the process expression assignment we must consider the rule above at the level sessions: we have access to an ambient session \( x \) that offers \( A \) and we wish to use it to offer \( A \) outright along channel \( z \). Thus, the natural
interpretation is to forward communication between the two channels \( x \) and \( z \):

\[ \text{Γ; } x:A \vdash \text{fwd } x \, z :: z:A \quad \text{(ID)} \]

or in the \( \pi \)-calculus syntax of Fig. 1:

\[ \text{Γ; } x:A \Rightarrow [x \leftrightarrow z] :: z:A \quad \text{(ID)} \]

If we consider what happens during cut elimination we can see that a cut against identity simply erases the identity (but in a setting where we have labels attached to terms and assumptions, this requires renaming):

\[ \frac{\text{Γ; } \Delta_1, \Delta_2 \vdash P \, x :: x:A \quad \text{Γ; } \Delta_2, x:A \vdash \text{fwd } x \, z :: z:A \quad \text{(ID)}}{\text{Γ; } \Delta_1, \Delta_2 \vdash \text{new } x. (P \, x; \text{fwd } x \, z) :: z:A \quad \text{(CUT)}} \]

\[ \Rightarrow \text{Γ; } \Delta_1, \Delta_2 \vdash P\{z/x\} :: z:A \]

Thus the SSOS rule for forwarding simply equates the two channel names:

\[ \text{(fwd)} \text{ exec (fwd } x \, z) \rightarrow \{x = z\} \]

The corresponding \( \pi \)-calculus reduction is:

\[ (\nu x) (P \mid [x \leftrightarrow z]) \rightarrow P\{z/x\} \]

**Multiplicatives**  
Multiplicative conjunction \( A \otimes B \) denotes the simultaneous availability of both \( A \) and \( B \), where both \( A \) and \( B \) are intended to be used. Thus, to offer \( A \otimes B \) we must split our linear resources in two parts, one used to provide \( A \) and the other to provide \( B \):

\[ \frac{\text{Γ; } \Delta_1 \vdash A \quad \text{Γ; } \Delta_2 \vdash B}{\text{Γ; } \Delta_1, \Delta_2 \vdash A \otimes B \quad (\otimes \text{R})} \]

As highlighted in Fig. 2, the session type \( A \otimes B \) denotes an output of a (fresh) session channel that offers \( A \) and a change of state to offer \( B \) along the original channel. The process expression assignment is:

\[ \frac{\text{Γ; } \Delta_1 \vdash P \, y :: y:A \quad \text{Γ; } \Delta_2 \vdash Q :: z:B}{\text{Γ; } \Delta_1, \Delta_2 \vdash \text{output } z (y.P \, y); Q :: z:A \otimes B \quad (\otimes \text{R})} \]

We thus have a process expression which can be identified by the channel \( z \), along which an output of a fresh session channel is offered. The intended session behavior \( A \) for this fresh channel is implemented by the process expression \( P \) (note that \( y \) occurs free in \( P \) but not in the output process expression nor in \( Q \), whereas the continuation \( Q \) offers \( z:B \), independently. Using \( A \otimes B \) warrants the use of both \( A \) and \( B \), as captured in the following left rule:

\[ \frac{\text{Γ; } \Delta, A, B \vdash C}{\text{Γ; } \Delta, A \otimes B \vdash C \quad (\otimes \text{L})} \]

At the process expression level, using a session that offers an output naturally consists of performing an input:

\[ \frac{\text{Γ; } \Delta, y:A, x:B \vdash R \, y :: z:C}{\text{Γ; } \Delta, x:A \otimes B \vdash \text{input } x; R \, y :: z:C \quad (\otimes \text{L})} \]

The process expression consists of an input along the ambient session channel \( x \), which is of type \( A \otimes B \) and therefore outputs a fresh channel that offers \( A \), which is bound to \( y \) in the continuation \( R \). Moreover, the state of \( x \) changes after
the input is performed, now offering $B$. The $\pi$-calculus process assignment consists of a bound output and an input for the right and left rules, respectively:

$$\Gamma; \Delta_1 \Rightarrow P :: y:A \quad \Gamma; \Delta_2 \Rightarrow Q :: z:B \tag{\otimes R}$$

$$\Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu y)z(y). (P \mid Q) :: z:A \otimes B$$

$$\Gamma; \Delta, y:A, x:B \Rightarrow R :: z:C$$

$$\Gamma; \Delta, x:A \otimes B \Rightarrow x(y), R :: z:C \tag{\otimes L}$$

To determine the correct operational behavior of our process expression assignment, we examine the principal cut reduction for $\otimes$, which justifies the intuitive semantics of input and output:

$$\Gamma; \Delta_1 \vdash P :: y:A \quad \Gamma; \Delta_2 \vdash Q :: z:B$$

$$\Gamma; \Delta_1, \Delta_2 \vdash \text{output } x(y,P_y); Q :: x:A \otimes B \tag{\otimes R}$$

$$\Gamma; \Delta_3, y:A, x:B \vdash R :: z:C$$

$$\Gamma; \Delta_3, x:A \otimes B \vdash y \leftarrow \text{input } x; R :: z:C \tag{\otimes L}$$

$$\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{new } x.((\text{output } x(y,P_y); Q); y \leftarrow \text{input } x; R) :: z:C \tag{CUT}$$

$$\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \text{new } x.(Q \otimes \text{new } y.(P; R)) :: z:C$$

The output $(\otimes R)$ synchronizes with the input $(\otimes L)$, after which we have a composition of three process expressions: $P$ offering session $A$ along the fresh channel, the continuation $Q$ offering session $B$ and the continuation $R$ that uses both to provide $C$, resulting in the following SSOS rule:

$$(\text{scm}) \text{ exec } (\text{output } x(y,P_y); Q) \otimes \text{exec } (y \leftarrow \text{input } x; R_y) \rightarrow (\exists y'. \text{exec } (P_y') \otimes \text{exec } (Q) \otimes \text{exec } (R_y'))$$

The rule above specifies that whenever an input and an output action of the form above are available on the same channel, the communication fires and, after generating a fresh channel $y'$, we transition to a state where the three appropriate continuations execute concurrently (with $P$ and $R$ sharing the fresh channel $y'$ for communication).

At the level of $\pi$-calculus processes, this results in the familiar synchronization rule of reductions semantics:

$$(\nu y)x(y). P \mid x(z). Q \rightarrow (\nu y)(P \mid R\{y/z\})$$

For the reader familiar with linear logic, it might seem somewhat odd that $\otimes$ which is a commutative operator is given a seemingly non-commutative interpretation. In the proof theory of linear logic, the commutative nature of $\otimes$ is made precise via a type isomorphism between $A \otimes B$ and $B \otimes A$. In the process assignment a similar argument can be developed by appealing to observational equivalence. A precise notion of observational equivalence is deferred to Section 9 where the necessary technical tools are developed. However, we highlight that it is the case that given a session of type $A \otimes B$ we can produce a realizer that offers $B \otimes A$ and vice-versa (see [37] for the full details), such that the composition of the two is equivalent to the identity.

The remaining multiplicative is linear implication $A \rightarrow B$, which can be seen as a resource transformer: given an $A$ it will consume it to produce a $B$. To offer $A \rightarrow B$ we simply need to be able to offer $B$ by using $A$. Dually, to use $A \rightarrow B$ we must provide a proof of $A$, which justifies the use of $B$.

$$\Gamma; \Delta, A \vdash B \tag{\rightarrow R}$$

$$\frac{\Gamma; \Delta_1 \vdash A \quad \Gamma; \Delta_2 \vdash C}{\Gamma; \Delta_1, \Delta_2, A \rightarrow B \vdash C \tag{\rightarrow L}}$$

The proof term assignment for linear implication is dual to that of multiplicative conjunction. To offer a session of type $x:A \rightarrow B$ we need to input a session channel of type $A$ along $x$, which then enables the continuation to offer a session of type $x:B$. Using such a session requires the dual action, consisting of an output of a fresh session channel that offers $A$, which then warrants using $x$ as a session of type $B$.

$$\frac{\Gamma; \Delta, x:A \vdash P_x :: z:B}{\Gamma; \Delta \vdash x \leftarrow \text{input } z; P_x :: z:A \rightarrow B \tag{\rightarrow R}}$$

11
\[
\frac{\Gamma; \Delta_1 \vdash Q :: y:A \quad \Gamma; \Delta_2, x:B \vdash R :: z:C}{\Gamma; \Delta_1, \Delta_2, x:A \multimap B \vdash \text{output } x \langle y, Q_y \rangle; R :: z:C} \quad (\otimes L)
\]

The π-calculus process assignment follows the expected lines:
\[
\frac{\Gamma; \Delta, x:A \Rightarrow P :: z:B}{\Gamma; \Delta \Rightarrow \nu x.P :: z:A \multimap B} \quad (\otimes R)
\]
\[
\frac{\Gamma; \Delta_1 \Rightarrow Q :: y:A \quad \Gamma; \Delta_2, x:B \Rightarrow R :: z:C}{\Gamma; \Delta_1, \Delta_2, x:A \multimap B \Rightarrow \nu y x \langle y, (Q \mid R) :: z:C \rangle} \quad (\otimes L)
\]

The proof reduction obtained via cut elimination matches that of \(\otimes\), and results in the same operational semantics.

Finally, we consider the multiplicative unit of linear logic, written 1. Offering 1 requires no linear resources and so can only be done when the linear context is empty. Using 1 also adds no extra linear resources, and so we obtain the following rules:
\[
\frac{}{\Gamma; \cdot \vdash 1} \quad (1R) \quad \frac{\Gamma; \Delta \vdash C}{\Gamma; \Delta, 1 \vdash C} \quad (1L)
\]

Thus, a session channel of type 1 denotes a terminated session channel, along which no further interactive behavior is possible. We explicit signal this termination by closing the communication channel through a specialized process expression written \(\text{close}\). Dually, since our language is synchronous, we must wait for ambient sessions of type 1 to close:
\[
\frac{}{\Gamma; \cdot \vdash \text{close } z :: \nu z 1} \quad (1R) \quad \frac{\Gamma; \Delta \vdash Q :: z:C}{\Gamma; \Delta, x:1 \vdash \text{wait } x; Q :: z:C} \quad (1L)
\]
Since no such primitives exist in the π-calculus, we can simply use a form of nullary communication for the multiplicative unit assignment:
\[
\frac{}{\Gamma; \cdot \vdash \nu z.0 :: z:1} \quad (1R) \quad \frac{\Gamma; \Delta \vdash Q :: z:C}{\Gamma; \Delta, x:1 \vdash \nu x.Q :: z:C} \quad (1L)
\]

As before, our guiding principle of appealing to principal cut reductions validates the informally expected behavior of the process expression constructs specified above (as well as for the synchronization step on π-calculus processes):
\[
\frac{\Gamma; \cdot \vdash \text{close } x :: x:1}{\Gamma; \Delta \vdash Q :: z:C} \quad (1R) \quad \frac{\Gamma; \Delta, x:1 \vdash \text{wait } x; Q :: z:C}{\Gamma; \Delta \vdash \text{new } x. \langle \text{close } x; (\text{wait } x; Q) \rangle :: z:C} \quad (1L) \quad \text{(CUT)} \quad \text{\implies } \frac{}{\Gamma; \Delta \vdash Q :: z:C}
\]

The SSOS rule that captures this behavior is:
\[
(\text{CLOSE}) \quad \text{exec } \langle \text{close } x \rangle \otimes \text{exec } \langle \text{wait } x; P \rangle \rightarrow \{\text{exec } \langle P \rangle\}
\]

Essentially, whenever a session closure operation meets a process waiting for the same session channel to close, the appropriate continuation is allowed to execute. The associated reduction rule on π-calculus terms is just the usual communication rule.

**Additives**  Additive conjunction in linear logic \(A \& B\) denotes the ability to offer either \(A\) or \(B\). That is, the available resources must be able to provide \(A\) and must also be able to provide \(B\), but not both simultaneously (as opposed to \(\otimes\) which requires both to be available). Therefore, we offer \(A \& B\) by being able to offer \(A\) and \(B\), without splitting the context:
\[
\frac{\Gamma; \Delta \vdash A \quad \Gamma; \Delta \vdash B}{\Gamma; \Delta \vdash A \& B} \quad (\&R)
\]
To use \( A \& B \), unlike \( A \otimes B \) where we are warranted in using both \( A \) and \( B \), we must chose which of the two resources will be used:

\[
\begin{align*}
\Gamma; \Delta, A \vdash C & \quad (\& \mathcal{L}_1) \\
\Gamma; \Delta, A \& B \vdash C & \quad (\& \mathcal{L}_2)
\end{align*}
\]

Thus, \( A \& B \) denotes a form of alternative session behavior where a choice between the two behaviors \( A \) and \( B \) is offered.

\[
\frac{\Gamma; \Delta \vdash P :: z: A \quad \Gamma; \Delta \vdash Q :: z: B}{\Gamma; \Delta \vdash \text{case}(P,Q) :: z: A \& B} \quad (\& \mathcal{R})
\]

The process expression above waits on session channel \( z \) for a choice between the left or right branches, which respectively offer \( z: A \) and \( z: B \). Using such a process expression is achieved by performing the appropriate selections:

\[
\begin{align*}
\Gamma; \Delta, x: A \vdash R :: z: C & \quad (\& \mathcal{L}_1) \\
\Gamma; \Delta, x: A \& B \vdash x.\text{inl}; R :: z: C & \quad (\& \mathcal{L}_2)
\end{align*}
\]

The \( \pi \)-calculus process assignment coincides precisely with the process expression assignment above.

The two principal cut reductions (one for each right-left rule pair) capture the expected behavior of choice and selection:

\[
\begin{align*}
\frac{\Gamma; \Delta_1 \vdash P :: x: A \quad \Gamma; \Delta_1 \vdash Q :: x: B}{\Gamma; \Delta_1 \vdash \text{case}(P,Q) :: x: A \& B} \quad (\& \mathcal{R})
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma; \Delta_1, x: A \vdash R :: z: C & \quad (\& \mathcal{L}_1) \\
\Gamma; \Delta_2, x: A \& B \vdash x.\text{inl}; R :: z: C & \quad (\& \mathcal{L}_2) \\
\Gamma; \Delta_1, \Delta_2 \vdash \text{new } x.(x.\text{case}(P,Q);(x.\text{inl}; R)) :: z: C \quad \text{(cut)}
\frac{\Gamma; \Delta_1 \vdash P :: x: A \quad \Gamma; \Delta_2, x: A \vdash R :: z: C}{\Gamma; \Delta_1, \Delta_2 \vdash \text{new } x.(P; R) :: z: C} \quad \text{(cut)}
\end{align*}
\]

Resulting in the following SSOS rule (omitting the obvious rule for the right-branch selection):

\[
(\text{choice}) \quad \text{exec}(x.\text{inl}; P) \otimes \text{exec}(x.\text{case}(Q, R)) \rightarrow \{ \text{exec}(P) \otimes \text{exec}(Q) \}
\]

The reduction rules induced on \( \pi \)-calculus terms are those expected for guarded binary choice:

\[
\begin{align*}
x.\text{case}(P, Q) \mid x.\text{inl}; R & \rightarrow P \mid R \\
x.\text{case}(P, Q) \mid x.\text{inr}; R & \rightarrow Q \mid R
\end{align*}
\]

Additive disjunction \( A \oplus B \) denotes an internal choice between \( A \) or \( B \), in that offering \( A \oplus B \) only requires being able to offer either \( A \) or \( B \), but not necessarily both unlike additive conjunction. In this sense, additive disjunction is dual to additive conjunction: the “choice” is made when offering \( A \oplus B \), and using \( A \oplus B \) requires being prepared for both possible outcomes:

\[
\begin{align*}
\Gamma; \Delta \vdash A \quad (\oplus \mathcal{R}_1) \\
\Gamma; \Delta \vdash A \oplus B & \quad (\oplus \mathcal{L}_1)
\end{align*}
\]

\[
\begin{align*}
\Gamma; \Delta \vdash B \quad (\oplus \mathcal{R}_2) \\
\Gamma; \Delta, A \vdash C & \quad \Gamma; \Delta, B \vdash C \quad (\oplus \mathcal{L}_2)
\end{align*}
\]

The duality between \( \oplus \) and \( \& \) is also made explicit by the process expression assignment (and the \( \pi \)-calculus process assignment):

\[
\begin{align*}
\frac{\Gamma; \Delta \vdash P :: z: A}{\Gamma; \Delta \vdash z.\text{inl}; P :: z: A \oplus B} \quad (\oplus \mathcal{R}_1) \\
\frac{\Gamma; \Delta \vdash P :: z: B}{\Gamma; \Delta \vdash z.\text{inr}; P :: z: A \oplus B} \quad (\oplus \mathcal{R}_2)
\end{align*}
\]

\[
\begin{align*}
\frac{\Gamma; \Delta, x: A \vdash Q :: z: C \quad \Gamma; \Delta, x: B \vdash R :: z: C}{\Gamma; \Delta, x: A \oplus B \vdash x.\text{case}(Q, R) :: z: C} \quad (\oplus \mathcal{L})
\end{align*}
\]

The proof reductions obtained in cut elimination are identical to the cases for additive conjunction and so are omitted for brevity.
Exponential To allow for controlled forms of weakening and contraction in linear logic Girard introduced the exponential \(! A\), denoting a proposition \(A\) which need not satisfy linearity and can be weakened and contracted in a proof, thus being able to be used in an unrestricted way. In the formulation of linear logic presented here, using distinct contexts for unrestricted and linear propositions, we require additional judgmental principles. To ensure cut elimination, we need an additional cut principle that given an unrestricted proposition, places it in the unrestricted context (we write \(\cdot\) for the empty context). Moreover, we need a copy rule that enables the use of unrestricted propositions:

\[
\frac{\Gamma; A \vdash \Delta; C}{\Gamma; \Delta \vdash C} \quad (\text{cut}'\,1) \quad \frac{\Gamma; A; \Delta, A \vdash C}{\Gamma; A; \Delta \vdash C} \quad (\text{copy})
\]

At the level of session types, persistent or unrestricted sessions consist of replicated services that may be used an arbitrary number of times. The process expression assignment to the unrestricted version of cut consists of a replicated input that guards the process expression \(P\) implementing the persistent session behavior (which therefore cannot use any linear sessions), composed with the process expression \(Q\) that can use the replicated session. Such uses are accomplished by triggering a copy of the replicated session, which binds a fresh session channel \(x\) that will be used for communication with the replicated instance.

\[
\frac{\Gamma; A \vdash \Delta; Q_u :: z_C}{\Gamma; \Delta \vdash \text{new} \! u. (x \leftarrow \text{input} \! u \ P_x; Q_u) :: z_C} \quad (\text{cut}')
\]

\[
\frac{\Gamma; u; A; \Delta; x : A \vdash R_x :: z_C}{\Gamma; u; A; \Delta \vdash x \leftarrow \text{copy} \! u; R_x :: z_C} \quad (\text{copy})
\]

For the \(\pi\)-calculus assignment, the unrestricted cut essentially consists of a combination of replicated input and parallel composition, whereas the cut rule consists of a (bound) output:

\[
\frac{\Gamma; A \vdash \Delta; Q_u :: z_C}{\Gamma; \Delta \vdash (\nu u)(\! u. P \mid Q) :: z_C} \quad (\text{cut}')
\]

\[
\frac{\Gamma; u; A; \Delta \vdash R :: z_C}{\Gamma; u; A; \Delta \vdash (\nu x)u(x). R :: z_C} \quad (\text{copy})
\]

During cut elimination, when a persistent cut meets an instance of the copy rule, we generate a (linear) cut. We show this below, using process expressions:

\[
\frac{\Gamma; A; \Delta; x : A \vdash Q_x :: z_C}{\Gamma; A; \Delta \vdash x \leftarrow \text{copy} \! u; Q_x :: z_C} \quad (\text{copy})
\]

\[
\frac{\Gamma; u; A; \Delta \vdash (\nu u)(\! u. x \leftarrow \text{input} \! u \ P_x; (x \leftarrow \text{copy} \! u; Q_x)) :: z_C}{\Gamma; u; A; \Delta \vdash (\nu u)(\! u. x \leftarrow \text{input} \! u \ P_x; \text{new} x. (P \mid Q)) :: z_C} \quad (\text{cut}')
\]

The reduction above formally captures the intuitive meaning of spawning a replica of the persistent session offered by \(P\), which is then composed accordingly with the continuation \(Q\).

\[
(\text{ucut}) \quad \text{exec} \left(\text{new} \! u. (x \leftarrow \text{input} \! u \ P_x; Q)\right) \rightarrow \{\exists u. \text{exec}(x \leftarrow \text{input} \! u \ P_x) \otimes \text{exec}(Q_u)\}
\]

\[
(\text{copy}) \quad \text{exec}(x \leftarrow \text{input} \! u; P_x) \otimes \text{exec}(x \leftarrow \text{copy} \! u; Q_x) \rightarrow \{\exists x'. \text{exec}(P_{x'}) \otimes \text{exec}(Q_{x'})\}
\]

The rule (ucut) takes a persistent cut and sets up the replicated input required to execute the persistent session implemented by \(P\). Note the usage of \(\text{exec}\) to indicate that the input is in fact replicated. The rule (copy) defines how to use such a replicated input, triggering a replica which executes in parallel with the appropriate continuation \(Q\), sharing a fresh session channel for subsequent communication. The persistent nature of the replicated input ensures that further replication of \(P\) is always possible.

For the \(\pi\)-calculus processes we obtain the following:

\[
\! u(x). P \mid (\nu x)u(x). Q \rightarrow \! u(x). P \mid (\nu x)(P \mid Q)
\]
Having justified the necessary judgmental principles, we now consider the exponential $!A$. Offering $!A$ means that we must be able to realize $A$ an arbitrary number of times and thus cannot use any linear resources to do so, since these must be used exactly once. To use $!A$, we simply need to move $A$ to the appropriate unrestricted context, allowing it to be copied an arbitrary number of times.

$$
\frac{\Gamma; \vdash A}{\Gamma; \vdash !A} \quad \text{(IR)}
\frac{\Gamma, A; \Delta \vdash C}{\Gamma; \Delta, !A \vdash C} \quad \text{(L)}
$$

The session interpretation of $!A$ is of a session offering the behavior $A$ in a persistent fashion. Thus, the process expression assignment for $!A$ is:

$$
\frac{\Gamma; \vdash P_x :: x:A}{\Gamma; \vdash \text{output } z \,(x,P_x) :: z:!A} \quad \text{(IR)}
\frac{\Gamma; \vdash \text{output } z :: z:C}{\Gamma, w:A; \Delta \vdash Q_{tu} :: z:C} \quad \text{(L)}
\frac{\Gamma, w:A; \Delta \vdash Q_{tu} :: z:C}{\Gamma; \Delta, x:!A \vdash \text{input } x; Q_{tu} :: z:C} \quad \text{(L)}
$$

The process expression for the $!R$ rule performs an output of a (fresh) persistent session channel along $z$, after which its continuation will be prepared to receive requests along this channel by spawning new copies of $P$ as required. Dually, the process expression for the $!L$ rule inputs the persistent session channel along which such requests will be performed by $Q$.

For $\pi$-calculus process assignment, we offer a session of type $!A$ by performing the same output as in the process expression assignment, but we must make the replicated input explicit in the rules:

$$
\frac{\Gamma; \cdot \vdash P_y :: y:A}{\Gamma; \cdot \vdash \nu u \,(z(u).u(x).P :: z:A)} \quad \text{(IR)}
\frac{\Gamma, w:A; \Delta \vdash Q_{tu} :: z:C}{\Gamma; \Delta, x:!A \Rightarrow x(u).Q :: z:C} \quad \text{(L)}
$$

The proof reduction we obtain from cut elimination is given below (using process expressions):

$$
\frac{\Gamma; \cdot \vdash P_y :: y:A}{\Gamma; \cdot \vdash \text{output } x \,(y,P) :: x:!A} \quad \frac{\Gamma, w:A; \Delta \vdash Q_{tu} :: z:C}{\Gamma; \Delta, x:!A \vdash \text{input } x; Q_{tu} :: z:C}
\quad \text{(cut)}
\frac{\Gamma; \cdot \vdash P_y :: y:A \quad \Gamma, w:A; \Delta \vdash Q_{tu} :: z:C}{\Gamma; \Delta \vdash \text{new } x \,(\text{output } x \,(y,P_y); (\text{input } x; Q_{tu})) :: z:C}
\quad \text{(cut')}
\Rightarrow \frac{}{\Gamma; \Delta \vdash \text{new } x \,(z(u).z(u).P_y; Q_{tu})}
$$

Essentially, the cut is transformed into an unrestricted cut which then allows for the reductions of instances of the copy rule as previously shown. The SSOS rule that captures this behavior is:

\[
\text{(repl)} \quad \text{exec} \,(\text{output } x \,(y,P_y)) \otimes \text{exec} \,(\text{input } x; Q_{tu}) \rightarrow \{ \text{exec} \,(\text{new } x \,(\text{input } x; P_y; Q_{tu})) \}
\]

The associated $\pi$-calculus reduction rule is just an input/output synchronization. It should be noted that the term assignment for the exponential and its associated judgmental principles differs from that of $\Sigma$. In $\Sigma$, the $!L$ rule was silent at the level of processes and so despite there being a proof reduction in the proof theory, no matching process reduction applied. In the assignment given above this is no longer the case and so we obtain a tighter logical correspondence by forcing reductions on terms to match the (principal) reductions from the proof theory.

**Properties** One of the most important reasons for developing a correspondence between proofs and programs is that several properties of interest follow naturally from logical soundness. First, we note that a simulation between the dynamics of proofs and our process expression assignment is relatively straightforward to establish (cf. $\delta$ for the $\pi$-calculus assignment) and we refrain from doing so here for the sake of conciseness. In the interpretation developed above, the two fundamental properties that follow from logical soundness are type preservation or *session fidelity*, and progress or *deadlock-freedom*. We can formulate the two properties formally using both the $\pi$-calculus and the process expression term assignment. For the former, we refer the reader to $\delta$ for the complete development. In the latter case, to state the type preservation theorem we must be able to talk about the types and channels during execution. An elegant way to accomplish this is to annotate each $\text{exec } P$ with the channel $c$ along which $P$ offers its
output and its type $A$. This exploits the observation that every process offers a service along exactly one channel, and for every channel there is exactly one process providing a service along it. This extended form is written $\text{exec } P \in A$

The rules given above can be updated in a straightforward fashion, and the original rules can be recovered by erasure. The annotations fix the role of every channel in a communication as either offered or used, and we can check if the whole process state $\Omega$ is well-typed according to a signature of (linear and shared) channels $\Sigma$.

We write $\vdash (\Sigma; \Omega) ::= c_0 : 1$ if process state $\Omega$ uses channels in $\Sigma$ accordingly and offers $1$ along an initial channel $c_0$ that is offered but not used anywhere. Initially, we have a closed process expression $P_0$ and $\vdash (\cdot \text{exec } P_0 c_0 1) ::= c_0 : 1$ (without loss of generality since using composition we can introduce well-typed processes offering an arbitrary type, regardless of the type of $c_0$). Overall, a pair consisting of the currently available channels and the process state evolves via multiset rewriting [10] to another pair, potentially containing new channels and the new process state.

**Theorem 1** (Type Preservation). If $\vdash (\Sigma; \Omega) ::= c_0 : 1$ and $(\Sigma; \Omega) \rightarrow^* (\Sigma'; \Omega')$ then $\vdash (\Sigma'; \Omega') ::= c_0 : 1$.

The proof requires us to relate typing derivations of process expressions to typings of the global executing process state. This turns out to be easy, since substructural operational semantics breaks down the global state into its local process expressions.

**Theorem 2** (Progress). For every well-typed process state $\vdash (\Sigma; \Omega) ::= c_0 : 1$, either $\Omega = (!\Omega', \text{exec } \text{close } c_0 c_0 1)$ where $!\Omega'$ consists of propositions of the form $\text{exec } P$, or $(\Sigma; \Omega) \rightarrow (\Sigma'; \Omega')$ for some $\Sigma'$ and $\Omega'$.

Progress is, as usual, in the process setting, slightly harder to prove. Once we account for the internal transitions of processes, we note that in a well-typed state $\Omega$, persistent processes (which always perform a replicating input) can never block. Due to linear well-typing of the state, we can therefore restrict attention to the remaining $k + 1$ processes that offer communication along $k + 1$ channels, but using only $k$ channels since $c_0$ does not have a match. Now we perform an induction on $k$. If $P_0$ is blocked on $c_0$, it must have the form stated in the theorem (by inversion on its typing) and we are done. If not, it must be blocked on some other channel, say, $c_1$. Now the process $P_0$ offering $c_1$ is either blocked on $c_1$, in which case it can communicate with $P_0$ and we can make a transition, or it must be blocked on some other $c_2$. We proceed in this way until we must come to $P_k$, which must be blocked on $c_k$ and can communicate with $P_{k+1}$ since no other linear channel $c_{k+1}$ remains on which it could be blocked.

## 4 The Power of Logic: Beyond Traditional Session Types

In this section we support the more general claim that linear logic can indeed be used as a suitable foundation for message-passing concurrent computation by extending the connection beyond the basic session typing discipline of Section 3. Specifically, we account for the idea of value-dependent session types, that is, session types that may depend on the values exchanged during communication, and parametric polymorphism (in the sense of Reynolds [41, 40]) at the level of session types. The former allows us to express richer properties within the type structure by considering first-order linear logic, describing not just the interactive behavior of systems but also constraints and properties of the values exchanged during communication. Moreover, our logically grounded framework enables us to express a high-level form of proof-carrying code in a concurrent setting, where processes may exchange proof certificates that attest for the validity of the properties asserted in types. As we will see, we achieve this by also incorporating proof irrelevance and modal affirmation into our term language.

Polymorphism is a concept that has been explored in the session-type community but usually viewed from the subtyping perspective [17] (while some systems do indeed have ideas close to parametric polymorphism, they are either not applied directly to session types [3] or they do not develop the same results we are able to, given our logical foundation). Here we explore polymorphism in the sense of System F by studying the concurrent interpretation of second-order linear logic.

We highlight these two particular extensions to the basic interpretation of [3], although others are possible, ranging from asynchronous process communication [12] to parallel evaluation strategies for $\lambda$-calculus terms [51]. The key point we seek to establish is that our framework is not only rich enough to explain a wide range of relevant concurrent phenomena, but also that it enables us to do so in a clean and elegant way, where the technical machinery is greatly simplified when compared to existing (non-logically based) approaches. Moreover, we can study these phenomena
in a *compositional* way insofar as we remain within the boundaries of logic, that is, we can develop these extensions independently from each other, provided we ensure that the underlying logical foundation is sound.

### 4.1 Value-Dependent Session Types

In the context of \(\lambda\)-calculus and the Curry-Howard correspondence, when we move from intuitionistic propositional logic to the first-order setting we obtain a *dependently typed* \(\lambda\)-calculus [25]. While the idea of dependent types seems superficially simple (types that can depend on terms of the language), the gain in expressive power is far from trivial, providing the ability to write sophisticated theorems as types, for which the proofs are the inhabitants of the type. In this regard, dependent type theories have been studied to great lengths, both for the more theoretical aspects regarding the foundations of mathematics but also for the more pragmatical aspects of correctness by typing that dependent types enable.

In the area of session typed communication, dependent types in the sense described above are virtually non-existent. While some notions of dependent session types exist, they do not carry with them the richness associated with dependent types in the sense of type theory, either because they are designed with a very specific goal in mind (such as [58], where dependencies appear as indices that parameterize session types w.r.t the number of messages exchanged or the number of principals involved in a protocol) or simply because the appropriate logical basis has eluded the research community for a long time.

We here consider a form of dependent session types that is closer to the notion of dependent types in functional type theories, where session types may depend on (or be indexed by) terms of a functional type theory such as LF [25] or the Calculus of Constructions [11]. We achieve this by considering the first-order quantifiers of linear logic where the domain of quantification is itself a (functional) dependent type theory. In this sense, it is not a full dependent type theory since we introduce a separation between the index language and the language of session types, but already it greatly enhances the expressive power of traditional session typed languages. We refrain from a complete development of this idea here, referring the interested reader to [50] for the details.

We first consider the sequent calculus rules for (first-order) universal and existential quantification in linear logic, where the domain is quantification is an unspecified typed \(\lambda\)-calculus:

\[
\frac{\Psi; x:\tau; \Gamma; \Delta \vdash A}{\Psi; \Gamma; \Delta \vdash \forall x:\tau.A} \quad \frac{\Psi; M : \tau}{\Psi; \Gamma; \Delta, A\{M/x\} \vdash C} \quad \frac{\Psi; \Gamma; \Delta, \forall x:\tau.A \vdash C}{\Psi; \Gamma; \Delta \vdash C} \quad (\forall L)
\]

To justify the quantification scheme above (besides extending the language of propositions to include the quantifiers), we extend our basic judgment to \(\Psi; \Gamma; \Delta \vdash A\), where \(\Psi\) is a context region that tracks the term variables introduced by quantification. We also need an additional judgment for the domain of quantification, written \(\Psi \vdash M : \tau\), which allows us to assert that \(M\) witnesses \(\tau\) for the purposes of providing a witness to quantifiers.

What should then be the concurrent interpretation of universal quantification? Let us consider the principal cut reduction:

\[
\frac{\Psi; x:\tau; \Gamma; \Delta_1 \vdash A}{\Psi; \Gamma; \Delta_1 \vdash \forall x:\tau.A} \quad \frac{\Psi; M : \tau}{\Psi; \Gamma; \Delta_2, A\{M/x\} \vdash C} \quad \frac{\Psi; \Gamma; \Delta, \forall x:\tau.A \vdash C}{\Psi; \Gamma; \Delta, \Delta_1, \Delta_2 \vdash C} \quad (\forall L)
\]

We require a substitution principle that combines the fact that \(\Psi \vdash M : \tau\) with the premise of the \((\forall R)\) rule to obtain the left premise of the reduced cut. From a concurrency perspective, this hints that the term assignment for the right rule should have an input flavor. Compatibility, the left rule must be an output:

\[
\frac{\Psi; x:\tau; \Gamma; \Delta \vdash P : z:A}{\Psi; \Gamma; \Delta \vdash x \leftarrow \text{input} \ z : P \vdash z:\forall x:\tau.A} \quad \frac{\Psi \vdash M : \tau}{\Psi; \Delta, y:A\{M/x\} \vdash Q : z:C} \quad \frac{\Psi; \Gamma; \Delta, y:\forall x:\tau.A \vdash \text{output} \ y \ M : Q : z:C}{(\forall L)}
\]
Thus, we identify quantification with communication of terms of a (functional) type theory. Existential quantification is dual. The right rule consists of an output, whereas the left rule is an input:

$$
\frac{\Psi \vdash M : \tau \quad \Gamma ; \Delta \vdash P :: z:A{\{M/x\}}}{\Psi;\Gamma;\Delta \vdash \text{output } y \ M;P :: z:\exists x:\tau.A} \quad (\exists R) \quad \frac{\Psi;\Gamma;\Delta, y:A \vdash Q :: z:C}{\Psi;\Gamma;\Delta, y:z:z.A \vdash x \leftarrow \text{input } y; P :: z:C} \quad (\exists L)
$$

One important idea is that we do not restrict a priori what the particular type theory which defines the domain of quantification should be, only requiring it to adhere to basic soundness properties of type safety, and that it remain separate from the linear portion of our framework. To give an SSOS to term passing we require the ability to evaluate functional terms. The predicate \(\text{eval} M V\) expresses that the functional term \(M\) evaluates to value \(V\) without using linear resources. The evaluation of functional terms is a usual call-by-value semantics. And so we obtain the following SSOS rule:

$$\text{(vcom)} \quad \text{exec (output } c \ M; P \text{) } \otimes \text{exec (input } x \leftarrow \text{input } c; Q_x \text{) } \otimes \text{eval } M V \rightarrow_o \{\text{exec (} P \text{) } \otimes \text{exec (} Q \text{)}\}$$

The \(\pi\)-calculus process assignment requires an extension of the calculus with the functional term language, whose terms are then exchanged between processes during communication:

$$\frac{\Psi, x:\tau; \Gamma; \Delta \Rightarrow P :: z:A}{\Psi;\Gamma;\Delta \Rightarrow z(x),P :: z:\forall x:\tau.A} \quad (\forall R) \quad \frac{\Psi \vdash M : \tau \quad \Psi;\Gamma;\Delta, y:A{\{M/x\}} \Rightarrow Q :: z:C}{\Psi;\Gamma;\Delta, y:z:z.A \Rightarrow y(M),Q :: z:C} \quad (\forall L)$$

$$\frac{\Psi \vdash M : \tau \quad \Psi;\Gamma;\Delta \Rightarrow P :: z:A{\{M/x\}}}{\Psi;\Gamma;\Delta \Rightarrow y(M),P :: z:\exists x:\tau.A} \quad (\exists R) \quad \frac{\Psi, x:z; \Gamma; \Delta, y:A \Rightarrow Q :: z:C}{\Psi;\Gamma;\Delta, y:z:z.A \Rightarrow y(x),P :: z:C} \quad (\exists L)$$

The reduction rule is the expected synchronization between term inputs and outputs:

$$x\langle M \rangle.P \ | \ x\langle y \rangle.Q \rightarrow P \ | \ Q\{M/y\}$$

This seemingly simple extension to our framework enables us to specify incredibly richer types than those found in traditional session type systems since we may now refer to the communicated values in the types, and moreover, if we use a dependent type theory as the term language, we can also effectively state properties of these values and exchange proof objects that witness these properties, establishing a high-level framework of concurrent proof-carrying code.

An Example Consider the following session type, commonly expressible in session typed languages (in our case, consider a straightforward extension of our framework without quantification with basic data types):

\[\text{Indexer} \triangleq \text{file} \rightarrow \text{file} \otimes 1\]

The type \text{Indexer} is intended to specify an indexing service. Clients of this service send it a document and expect to receive back an indexed version of the document. However, the type only specifies the communication behavior of receiving and sending back a file, the indexing portion of the service (which is its actual functionality) is not captured at all in the type. This effectively means that any service that receives and sends files adheres to the specification of \text{Indexer}, which is clearly intended to be more precise. This is especially problematic in a distributed setting, where such a general specification is clearly not enough.

If we use dependent session types, we can make the specification much more precise by not only specifying that the sent and received files have to be, for instance, PDF files, but we can also specify that the file sent by the index has to "agree" with the received one, in the sense that it is, in fact, its indexed version:

\[\text{Indexer}_{\forall} \triangleq \forall f:\text{file} \forall p:\text{pdf}(f) \exists g:\text{file} \exists q_1:\text{pdf}(g) \exists q_2:\text{agree}(f, g) \cdot 1\]

The revised version of the indexer type \text{Indexer}_{\forall} now specifies a service that will receive a file \(f\), a proof object \(p\) certifying that \(f\) is indeed a pdf, and will then send back a file \(g\), a proof object certifying \(g\) as a pdf file and also a proof object that certifies that \(g\) and \(f\) are in agreement. This much more precise specification provides very strong guarantees to the users of the indexing service, which are not feasibly expressible without (value) dependent session types.
**Proof Irrelevance and Affirmation** While dependent session types provide additional guarantees on the properties of data exchanged during communication, these guarantees come at the price of explicit exchanges of proof objects that certify these properties. For instance, in the type \texttt{Indexer_{PDF}} clients must now not only send the file but also a proof object that certifies that it is a valid PDF file. While in the abstract this may be desirable, in practice it can be the case that some proof objects need not be communicated, either because the properties are easily decidable or because of some trust relationship established between the communicating parties. These rather pragmatic considerations also fall under the scope of our framework, again by appealing to our solid logical underpinnings.

To account for the idea of potentially omitting certain proof objects, we require the type theory that makes up the quantification domain to include a proof irrelevance modality.

Proof irrelevance is a technique that allows us to selectively hide portions of a proof (and by the proofs-as-programs principle, portions of a program). The idea is that these “irrelevant” proof objects are required to exist for the purpose of type-checking, but they must have no bearing on the computational outcome of the program. This means that typing must ensure that these hidden proofs are never required to compute something that is not itself hidden. We internalize proof irrelevance in our functional language by requiring a modal type constructor, \([\tau]\) (read bracket \(\tau\)), meaning that there is a term of type \(\tau\), but the term is deemed irrelevant from a computational point of view. For the sake of conciseness we omit the technical details involved in such a type, which are detailed in [50][38].

Operationally, terms of type \([\tau]\) must be present during type-checking, but may consistently be erased at runtime and therefore their communication omitted, since by construction they have no computationally relevant content. For instance, a version of the indexing service that now uses proof irrelevance in order not to require the communication of proofs can be:

\[
\text{Indexer}_F \vdash \forall f.\text{file} \forall p:[\text{pdf}(f)].\exists q_f.\text{file} \exists q_1:[\text{pdf}(g)].\exists q_2:[\text{agree}(f, g)].\top
\]

A service of type \texttt{Indexer}_F is still required to satisfy the constraints imposed by \texttt{Indexer}_F, and so all the proof objects must exist for typechecking purposes, but they need not be communicated at runtime.

We now have a way of explicitly exchanging and omitting proof objects during execution, but it is often the case that what is required is something that lies somewhere in between the two extremes of this spectrum: we want to avoid the communication overhead of exchanging proof objects, but do not necessarily want to completely omit them outright, instead demanding some signed certificate for the existence of the proof, whose validity should in principle be easier to check. For example, when we download a large application we may be willing to trust its safety if it is digitally signed by a reputable vendor. On the other hand, if we are downloading a piece of Javascript code embedded in a web page, we may insist on some explicit proof that it is safe and adheres to our security policy. The key to making such trade-offs explicit in session types is a notion of affirmation (in the sense of [16]) of propositions and proofs by principals. Such affirmations can be realized through explicit digital signatures on proofs by principals, based on some underlying public key infrastructure.

The basic idea is to consider an additional modality in the term language that states that some principal \(K\) asserts (or affirms) the existence of a proof of some property \(\tau\), written \(\Diamond_K \tau\). Terms of this type would in practice be generated by performing a digital signature to the actual proof object that witnesses \(\tau\). Such an affirmation may seem redundant: after all, the certificate contains the term which can be type-checked. However, checking a digitally signed certificate may be faster than checking the validity of a proof, so we may speed up the system if we trust \(K\)’s signature. More importantly, if we have proof irrelevance, and some parts of the proof have been erased, then we have in general no way to reconstruct the original proofs. In this case we must trust the signing principal \(K\) to accept the \(\tau\) as true, because we cannot be sure if \(K\) played by the rules and did indeed have a proof. Therefore, in general, the affirmation of \(\tau\) by \(K\) is weaker than the truth of \(\tau\), for which we demand explicit evidence. Conversely, when \(\tau\) is true \(K\) can always sign it and be considered as “playing by the rules”.

We can now add these signed certificates to our running example:

\[
\text{Indexer}_F \vdash \forall f.\text{file} \forall p:[\text{pdf}(f)].\exists q_f.\text{file} \exists q_1:[\text{pdf}(g)].\exists q_2:[\text{agree}(f, g)].\top
\]

While we still do not require the communication of the proof objects that assert the files are actually PDFs, the type \texttt{Indexer}_F now specifies that a digitally signed (by principal \(I\)) certificate of the agreement of the sent and received files is communicated back to the client of the service. The combination of both proof irrelevance and affirmation is crucial: the type \(\Diamond_I [\text{agree}(f, g)]\) denotes a proof relevant object \(q_2\) which is an affirmation (and thus digitally signed)
by principal $I$ of a proof irrelevant object, asserting the agreement of $f$ and $g$. Thus, the object $q_2$ may not contain the actual proof object and is in principle substantially smaller and simpler to check.

This once again showcases the strengths of having a robust logical foundation, which not only allows us to move from session types to value dependent session types in a rather straightforward way, preserving the desirable properties of session fidelity (which in the dependent case also entails the validity of the properties expressed in the dependencies) and deadlock freedom, but also enables us to incorporate known logical concepts such as proof irrelevance and affirmation in our concurrent, session-typed setting. These extensions provide a clean account of high-level concepts such as proof-carrying code with various refined levels of trust: we may not trust the used services at all, and demand that proofs of all the properties be communicated; we may trust the services completely (or check that the properties hold ourselves) by omitting proofs at runtime; or we may appeal to some trusted third party to digitally sign objects that certify the existence of proof objects witnessing the desired properties.

### 4.2 Polymorphism

When considering concurrent message-passing systems, its rather straightforward to consider polymorphism at the data level, that is, systems that admit polymorphism on the data values exchanged in communication. It is less common to consider polymorphism at the behavioral level (in the sense of communication protocols), or behavioral genericity. This form of genericity is particularly important since it allows systems to be generic with respect to arbitrary communication protocols, which may be known and instantiated only at runtime. For instance, critical web applications often involve dynamic reconfiguration of communication interfaces/protocols (e.g. replacing a service provider), which should be transparent clients. To this end, such application should be conceived as generic with respect to such interfaces/protocols. Another common scenario is that of “cloud-based” services, which acquire computing resources when demand is high and release them as they are no longer needed. These scaling policies require the services to be generic with respect to their underlying coordination protocols, which may depend on the system’s architecture at a given time.

In the realm of concurrent processes, polymorphism was first studied by Turner [54], in the context of a simply-typed $\pi$-calculus. Berger et al. [3] were the first to study a $\pi$-calculus with parametric polymorphism based on universal and existential quantification over types. However, in the context of session types, polymorphism has mainly been developed as bounded polymorphism [17, 19], which is controlled via subtyping. This form of polymorphism is sufficient to capture data-level genericity but not behavioral genericity. True parametric polymorphism for session types in the sense of Reynolds was proposed by Wadler [55] in a way similar to our development, but without the associated reasoning techniques that arise by the study of parametricity which is done in Section 6.

By making use of our linear logic foundation for message-passing concurrency, we can give a rather clean account of polymorphism in the sense of behavioral genericity by considering a concurrent interpretation of second-order linear logic, exploring a correspondence along the lines of that between the polymorphic $\lambda$-calculus [21, 40] and second-order intuitionistic logic.

Our formulation of second-order linear logic uses a specialized context region for type variables introduced by second-order quantification (which we write $\Omega$) and an additional judgment $\Omega \vdash A$ type stating that $A$ is a well-formed type (or proposition) with free variables registered in $\Omega$. Thus the main judgment is now $\Omega; \Gamma; \Delta \vdash A$. The rules that define the well-formedness judgment are straightforward and thus omitted. The rules for (second-order) universal and existential quantification are:

$$\begin{align*}
\Omega, X; \Gamma; \Delta \vdash A & \quad \Omega \vdash B \text{ type} \\
\Omega; \Gamma; \Delta \vdash \forall X.A & \quad \Omega; \Gamma; \Delta, A\{B/X\} \vdash C \quad (\forall 2 R) \\
\Omega \vdash B \text{ type} & \quad \Omega, X; \Gamma; \Delta, A\{B/X\} \vdash C \\
\Omega; \Gamma; \Delta \vdash \exists X.A & \quad (\exists 2 R) \\
\Omega; \Gamma; \Delta, \exists X.A & \vdash C \quad (\exists 2 L)
\end{align*}$$

Proving a universally quantified proposition $\forall X.A$ simply requires us to prove the proposition $A$ without assuming any specifics on the type variable, whereas using a universally quantified proposition requires us to form a valid proposition $B$ which then warrants the use of $A\{B/X\}$. The existential is dual. We note that form of quantification implemented by these rules is inherently impredicative (i.e. the type variable $X$ in $\forall X.A$ may be instantiated with $\forall X.A$).
Much like in the concurrent interpretation of the first-order quantifiers of Section 4.1, the second-order quantifiers also have an input/output flavor. However, instead of exchanging data-values, we exchange session types. This essentially means that the process expression assignment (and \( \pi \)-calculus assignment) for second-order quantification enables an expressive form of abstract protocol communication.

\[
\frac{\Omega, X; \Gamma; \Delta \vdash P :: z:A}{\Omega; \Gamma; \Delta \vdash \text{input} z; P :: z:\forall X.A} \quad (\forall 2 R)
\]

\[
\frac{\Omega \vdash B \text{ type} \quad \Omega; \Gamma; \Delta \vdash P :: z:A{B/X}}{\Omega; \Gamma; \Delta \vdash \text{output} x B; :: z:\exists X.A} \quad (\exists 2 R)
\]

As before, the intuitive semantics of the assignment are justified by the following cut reduction:

\[
\frac{\Omega, X; \Gamma; \Delta \vdash P :: x:A}{\Omega; \Gamma; \Delta \vdash \text{input} z; P :: x:\forall X.A} \quad (\forall 2 L)
\]

\[
\frac{\Omega \vdash B \text{ type} \quad \Omega; \Gamma; \Delta, x:A{B/X} \vdash Q :: z:C}{\Omega; \Gamma; \Delta, x:\forall X.A \vdash \text{output} x B; Q :: z:C} \quad (\exists 2 L)
\]

\[
\frac{\Omega; \Gamma; \Delta \vdash P{B/X} :: x:A{B/X}}{\Omega; \Gamma; \Delta \vdash x:A{B/X} \vdash Q :: z:C} \quad (\text{CUT})
\]

Which results in the following SSOS rule:

\[
\text{(tcom) } \text{exec} (\text{output} c B; P) \otimes \text{exec} (\text{input} x; Q) \rightarrow \{\text{exec} (P) \otimes \text{exec} (Q_B)\}
\]

The \( \pi \)-calculus assignment mimics the process expression assignment, requiring an extension of the \( \pi \)-calculus with the ability to communicate session types (we omit the rules for the existential for conciseness, they are completely dual to the ones for the universal):

\[
\frac{\Omega, X; \Gamma; \Delta \Rightarrow P :: z:A}{\Omega; \Gamma; \Delta \Rightarrow \text{input} z(X); P :: z:\forall X.A} \quad (\forall 2 R)
\]

\[
\frac{\Omega \Rightarrow B \text{ type} \quad \Omega; \Gamma; \Delta, x:A{B/X} \Rightarrow Q :: z:C}{\Omega; \Gamma; \Delta, x:\forall X.A \Rightarrow x(B), Q :: z:C} \quad (\forall 2 L)
\]

The full details of polymorphic session types as they arise form second-order linear logic can be found in [7, 6]. Here we just remark that the same soundness properties of before still follow naturally from the fact that we are working within the confines of logic.

So what kind of expressiveness do we gain with behavioral polymorphism in this sense? Consider the following session type:

\[
\text{CloudServer} \triangleq \forall X.!(\text{api} \rightarrow X) \rightarrow !X
\]

The CloudServer type represents a simple interface for a cloud-based application server. In our theory, this is the session type of a system which first inputs an arbitrary type (say GMaps); then inputs a shared service of type api \( \rightarrow \) GMaps. Each instance of this service yields a session that when provided with the implementation of an API will provide a behavior of type GMaps; finally becoming a persistent (shared) server of type GMaps. Our application server is meant to interact with developers who, by building upon the services it offers, implement their own applications. In our framework, the dependency between the cloud server and applications may be expressed by the typing judgment

\[
\vdash x: \text{CloudServer} \vdash \text{DripBox} :: z:dbox
\]

Intuitively, the judgment above says that to offer behavior dbox on z, the file hosting service represented by process DripBox relies on a linear behavior described by type CloudServer provided on x (no shared behaviors are required). The crucial role of behavioral genericity should be clear from the following observation: to support interaction with developers such as DripBox—which implement all kinds of behaviors, such as dbox above—any process realizing type CloudServer should necessarily be generic on such expected behaviors, which is precisely what we accomplish here through our logical foundation.

A natural question to ask is whether or not a theory of parametricity as that for the polymorphic \( \lambda \)-calculus exists in our setting, given our development of impredicative polymorphism in a way similar to that present in the work of Reynolds [41]. We answer positively to this question in Section 6, developing a theory of parametricity for polymorphic session types as well as the natural notion of equivalence that arises from parametricity, showing it coincides with the familiar process calculus notion of (typed) barbed congruence.
5 Towards a Concurrent Programming Language

My development up to this point focused on providing logically motivated accounts of relevant concurrent phenomena. Here I take a different approach to the logical interpretation and exploit its ability to express concurrent computation to develop a simple yet powerful session-based concurrent programming language based on the process expression assignment developed in the previous sections, encapsulating process expressions in a contextual monad embedded in a λ-calculus.

Clearly this is not the first concurrent programming language with session types. Many such languages have been proposed over the years [18][28][24][1]. The key aspect that is lacking from such languages is a true logical foundation, with the notable exception of Wadler’s GV [55]. This makes the metatheory of such languages substantially more intricate, and often the properties obtained “for free” due to logic cannot easily be replicated in their setting. The key difference between the language developed here and that of [55] is that we combine functions and concurrency through monadic isolation, whereas GV provides no such separation. This entails that the entirety of GV is itself linear, whereas the language developed here is not. Another significant difference between the two approaches is that the underlying type theory of GV is classical, whereas ours is intuitionistic. Monads are intuitionistic in their logical form [15], which therefore makes the intuitionistic form of linear logic a particularly good candidate for a monadic integration of functional and concurrent computation based on a Curry-Howard correspondence. We believe our natural examples demonstrate this clearly.

We note that our language is mostly concerned with the concurrent nature of computation, rather the issues that arise when considering distribution of actual executing code. We believe it should be possible to give a precise account of both concurrency and explicit distribution of running processes by exploring ideas of hybrid or modal logic in the context of our linear framework, similar to that of [34], where worlds in the sense of modal logic are considered as sites where computation may take place. One of the key aspects of our development that seems to align itself with this idea is the identification of running processes with the channel along which they offer their intended behavior, which provide a delimited running object that may in principle be located in one of these sites.

5.1 A Contextual Monadic Integration

The process expression assignment developed throughout this document enables us to write expressive concurrent behavior, but raises some issues when seen as a proper programming language. In particular, it is not immediately obvious how to fully and uniformly incorporate the system into a complete functional calculus to support higher-order, message-passing concurrent computation. Moreover, its not clear how significantly our session-based (typed) communication restricts the kinds of programs we are allowed to write, or how easy it is to fully combine functional and concurrent computation while preserving the ability to reason about programs in the two paradigms.

To address this challenge we use a contextual monad to encapsulate open concurrent computations, which can be passed in functional computation but also communicated between processes in the style of higher-order processes, providing a uniform integration of both higher-order functions and concurrent computation. For expressiveness and practicality, we allow the construction of recursive processes, which is a common motif in applications.

The key idea that motivates the use of our contextual monad is that we can identify an open process expression as a sort of “black box”: if we plug it in with process expressions implementing the sessions required by the open process expression, it will provide a session channel along which it implements its specified behavior. If we try to simply integrate process expressions and functional terms outright, it is unclear how to treat session channels combined with ordinary functional variables. One potential solution is to map channels to ordinary functional variables, forcing the entire language to be linear, which is a significant departure from typical approaches. Moreover, even if linear variables are supported, it is unclear how to restrict their occurrences so they are properly localized with respect to the structure of running processes. Thus, our solution consists of isolating process expressions from functional terms in such a way that each process expression is bundled with all channels (both linear and shared) it uses and the one that it offers.

The type structure of the language is given in Fig. 3, separating the types from the functional language, written τ, σ from session types, written A, B, C. The most significant type is the contextual monadic type \(\{\alpha_i : A_i \vdash a : A\}\), which is the type of monadic values that encapsulate session-based concurrency in the λ-calculus. The idea is that a value of type \(\{\alpha_i : A_i \vdash a : A\}\) denotes a process expression that offers the session \(A\) along channel \(a\), when provided with sessions \(a_1 : A_1\) through \(a_n : A_n\) (we write \(\{a : A\}\) when the context regions are empty). The language of session types
\[
\begin{align*}
\tau, \sigma & ::= \tau \to \sigma \mid \ldots \mid \forall t. \tau \mid \mu t. \tau \mid t \quad \text{(ordinary functional types)} \\
& | \{a_i : A_i \vdash a : A\} \quad \text{process offering } A \text{ along channel } a, \\
& \quad \text{using channels } a_i \text{ offering } A_i \\
A, B, C & ::= \tau \supset A \quad \text{input value of type } \tau \text{ and continue as } A \\
& | \tau \land A \quad \text{output value of type } \tau \text{ and continue as } A \\
& | A \rightarrow B \quad \text{input channel of type } A \text{ and continue as } B \\
& | A \otimes B \quad \text{output fresh channel of type } A \text{ and continue as } B \\
& | 1 \quad \text{terminate} \\
& | \&\{l_j : A_j\} \quad \text{offer choice between } l_j \text{ and continue as } A_j \\
& | \oplus\{l_j : A_j\} \quad \text{provide one of the } l_j \text{ and continue as } A_j \\
& | !A \quad \text{provide replicable service } A \\
& | \mu X. A \mid X \quad \text{recursive session type}
\end{align*}
\]

Figure 3: The Syntax of Types

is basically the one presented in the previous sections of this document, with the ability to send and receive values of functional type (essentially a non-dependent version of \(\forall\) and \(\exists\) from Section 4.1), with \(n\)-ary labelled versions of the choice and selection types and with recursive types. One subtle point is that since monadic terms are seen as opaque functional values, they can be sent and received by process expressions, resulting in a setting that is reminiscent of higher-order processes in the sense of \([44]\).

Our language appeals to two typing judgments, one for functional terms which we write \(\Psi \vdash M : \tau\), where \(M\) is a functional term and \(\Psi\) a context of functional variables; and another for process expressions \(\Psi; \Gamma; \Delta \vdash P :: z : A\), as before. For convenience of notation, we use \(\Delta = \text{lin}(\alpha_i; A_i)\) to denote that the context \(\Delta\) consists of the linear channels \(c_i : A_i\) in \(\alpha_i; A_i\); and \(\Gamma = \text{shd}(\alpha_i; A_i)\) to state that \(\Gamma\) consists of the unrestricted channels \(!u_i; A_i\) in \(\alpha_i; A_i\).

To construct a value of monadic type, we introduce a monadic constructor, typed as follows:

\[
\frac{\Delta = \text{lin}(\alpha_i; A_i) \quad \Gamma = \text{shd}(\alpha_i; A_i) \quad \Psi; \Gamma; \Delta \vdash P :: c : A}{\Psi \vdash c \leftarrow \{P_{\vec{c}, \vec{m}}\} \leftarrow \overline{\alpha_i; A_i} : \{\alpha_i; A_i \vdash c : A\}} (\{I\})
\]

We thus embed process expressions within the functional language. A term of monadic type consists of a process expression, specifying a channel name along which it offers and those it requires to be able to offer said session. Typing a monadic value requires that the underlying process expression be well-typed according to its specification.

To use a value of monadic type, we extend the process expression language with a form of contextual monadic composition (in Haskell terminology, with a monadic bind):

\[
\frac{\Delta = \text{lin}(\alpha_i; A_i) \quad \text{shd}(\alpha_i; A_i) \subseteq \Gamma \quad \Psi \vdash M : \{c; A \leftarrow \alpha_i; A_i\} \quad \Psi; \Gamma; \Delta' \vdash c : A \vdash Q_c :: d : D}{\Psi; \Gamma; \Delta, \Delta' \vdash c \leftarrow \overline{\alpha_i; Q_c} :: d : D} (\{E\})
\]

Thus, only process expressions may actually “unwrap” monadic values, which are opaque values from the perspective of the functional language. In a monadic composition, the monadic term \(M\) is provided with the channels \(\overline{\sigma}\) required to run the underlying process expression. When the evaluation of \(M\) produces the actual monadic value, it is composed in parallel with the process expression \(Q\), sharing a fresh session channel along which the two processes may interact.

In order to precisely define the SSOS for monadic composition we extend our semantics with a predicate for evaluating functional terms \(\text{eval} M V\) which expresses that the functional term \(M\) evaluates to value \(V\) without using linear resources. The SSOS rule for monadic composition is:

\[
\text{exec } (c \leftarrow \overline{\pi_c}; Q_{c'}) \otimes \text{eval } M (c \leftarrow \{P_{c, \overline{m}}\} \leftarrow \overline{\pi} \rightarrow \exists c'. \text{exec } (P_{c', \overline{m}}) \otimes \text{exec } (Q_{c'}))
\]

Executing a monadic composition evaluates \(M\) to a value of the appropriate form, which must contain a process expression \(P\). We then create a fresh channel \(c'\) and execute of \(P_{c', \overline{m}}\) itself, in parallel with \(Q_{c'}\). In the value of \(M\), the channels \(c\) and \(\overline{\pi}\) are all bound names, so we rename them implicitly to match the interface of \(M\) in the monadic composition.
One interesting aspect of monadic composition is that it subsumes the process composition constructs presented in the previous sections (proof theoretically this is justified by the fact that monadic composition effectively reduces to a cut). Given this fact, in our language we omit the previous process composition constructs and enable composition only through the monad.

The remainder of the constructs of the language that pertain to the concurrent layer are those of our process expression language, with a few minor conveniences. Value input and output uses the same constructs for the first-order quantifiers of Section 4.1, but without type dependencies. The choice and selection constructs are generalized to their \( n \)-ary labelled version, and so we have:

\[
\frac{\Psi; \Gamma; \Delta \vdash P_1 :: c_1 : A_1 \ldots \Psi; \Gamma; \Delta \vdash P_k :: c_k : A_k}{\Psi; \Gamma; \Delta \vdash \text{case } c \text{ of } \tilde{l}_j \Rightarrow P_j :: c : \& \{ \tilde{l}_j : A_j \} } \quad (\&R)
\]

\[
\frac{\Psi; \Gamma; \Delta, c : \& \{ \tilde{l}_j : A_j \} \vdash P :: d : D}{\Psi; \Gamma; \Delta \vdash \text{case } c \text{ of } \tilde{l}_j \Rightarrow c.l_j; P :: d : D } \quad (\&L)
\]

The rules for \( \oplus \) are dual, using the same syntax. The reader may then wonder how are recursive session types inhabited if we do not extend the process expression language with a recursor or fold and unfold constructs. Indeed, our process expression language does not have a way to internally define recursive behavior. Instead, we enable the definition of recursive process expressions through recursion in the functional language combined with the monad, expressed with a general fixpoint construct.

To clarify the point on recursion, consider the following session type that encodes an infinite stream of integers (for convenience, we assume a type definition mechanism in the style of ML or Haskell, distinguishing between functional and session type definitions with \texttt{type} and \texttt{stype}, respectively):

\[
\text{stype intStream} = \text{int} /\!\!\!\backslash \text{intStream}
\]

In order to produce such a stream, we write a recursive function producing a process expression:

\[
nats : \text{int} \to \{ c : \text{stream} \} \\
c \leftarrow nats \ x = \\
\{ \text{output c x} \} \\
c' \leftarrow nats \ (x+1) \\
fwd \ c \ c'
\]

This an example of a (recursive) function definition. We take some liberties with the syntax of these definitions for readability. In particular, we list interface channels on the left-hand side of the definition and omit the explicit fixpoint construct. We also omit the semicolon that separates actions from their process expression continuation, using a new line for readability. Since the recursive call is made via monadic composition, it starts a new process offering along a fresh channel \( c' \). Both for conciseness of notation and efficiency we provide a short-hand: if a tail-call of the recursive function provides a new channel which is then forwarded to the original offering channel, we can reuse the name directly, making the last line of the function above simply \( c \leftarrow nats \ (x+1) \). Superficially, it looks as if, for example, calling \( nats \ 0 \) might get into an infinite loop. However, communication in our language is synchronous, so the output will block until a matching consumer inputs the numbers. We can now construct a stream transducer. As an example, we write a filter that takes a stream of integers and produces a stream of integers, retaining only those satisfying a given predicate \( q : \text{int} \to \text{bool} \):

\[
\text{filter} : (\text{int} \to \text{bool}) \to \{ c : \text{stream} \mid d : \text{stream} \} \\
d \leftarrow \text{filter} \ q \leftarrow c = \\
\{ \ x \leftarrow \text{input c} \} \\
\text{case q x of true} \Rightarrow \text{output d x} \\
\mid \text{false} \Rightarrow d \leftarrow \text{filter} \ q \leftarrow c 
\]

The \texttt{filter} function is recursive, but not a valid coinductive definition unless we can show that filter will be true for infinitely many elements of the stream. To review, we summarize the syntax of our language in Fig 4.

An important aspect of our monadic integration is that it enables process expressions to be communicated as values by other process expressions. We showcase this feature in the following example, which also aims to clarify
the distinctions between recursion and replication by specifying an “app store” service. Further examples, as well as a study of the metatheory of the monadic language, ensuring it adheres to the desired type preservation and progress properties can be found in [53].

5.1.1 Example - An AppStore

The store offers a variety of different applications, themselves concurrent programs that rely on a remote API. Each store instance sends its client the application of choice as a monadic object and then terminates. The client can then run the application (in this scenario, all applications rely on a remote API, but the weather application also relies on a GPS module). The session types for the app. store and the weather app. are as follows (we use =>, Choice and Or as the concrete syntax for ⊃, & and ⊕, respectively):

\[
\text{stype Weather} = \text{Or}\{\text{sun:Weather, rain:Weather, cloudy:Weather}\}
\]

\[
\text{stype AppStore} = !\text{Choice}\{\text{weather: payInfo => \{d:API, g:GPS | - c:Weather\} /\ 1, travel: payInfo => \{d:API | - c:Travel\} /\ 1, game: payInfo => \{d:API | - c:Game\} /\ 1}\}
\]

The App Store service is represented as a replicated session, since each Client-Store interaction is a separate, independent session with a finite behavior, even though the store itself is a persistent service that can be accessed by an arbitrary number of clients. On the other hand, the weather application offers a recursive session that can continuously inform the client of the current weather conditions.

Below we show the code for a client that downloads and runs the weather application, by appealing to some library function to enable its GPS module and then plugging all the components together (the GPS module interface along channel g, the API on channel d and the application itself, bound to w) and executing the received application through monadic composition.

```ml
ActivateGPS : unit -> {g:GPS}
c <- WeatherClient () <- !u, d = { s <- copy !u
   s.weather
```
5.2 Connection to Higher Order-\(\pi\) Calculus

As the example of Section 5.1.1 shows, our language can easily encode process passing. Thus, the connection of our language with a process calculus can only be achieved via a \(\pi\)-calculus with higher-order features. The higher-order features of the calculus rely on the ability to communicate and execute processes as monadic values, using a dedicated construct which we dub spawn, similar to run in [33] or \(\ast\) in the higher-order calculi of [44].

The spawn primitive allows for the execution of monadic values and is written as \(\text{spawn}(M; a_1, \ldots, a_n; a.P)\), where \(a_1, \ldots, a_n\) denote the ambient channels required for the execution of the process expression underlying \(M\), and \(a\) the channel that will be generated for the interactions with \(P\), allowing for the execution of higher-order code. The operational semantics of spawn consists of evaluating \(M\) to a monadic value of the form \(a \leftarrow \{Q\} \leftarrow d_1, \ldots, d_n\) and then composing \(Q\) in parallel with \(P\), sharing a fresh name \(a\), where \(d_1\) through \(d_n\) in \(Q\) are instantiated with the channels \(a_1\) through \(a_n\). The typing rule is virtually identical to that of monadic composition. The typing rule for the spawn construct is:

\[
\Delta = \text{lin}(a_i:A_i) \quad \text{shd}(a_i:A_i) \subseteq \Gamma \quad \Psi \vdash M : \{a_i:A_i \vdash a; A\} \quad \Psi; \Gamma; \Delta, a: A \Rightarrow Q : z; C \quad \text{(SPAWN)}
\]

Operationally, the reduction rule for spawn matches precisely the SSOS rule for monadic composition, considering a big-step semantics for \(\lambda\)-terms since that is the strategy encoded in SSOS for convenience. We recall that \(\lambda\)-terms are evaluated using a call-by-value strategy and that monadic terms are values of the language.

\[
\text{spawn}(c \leftarrow \{P\} \leftarrow b; c.Q) \quad \rightarrow \quad (\nu c)(P[b/c] | Q)
\]

5.3 Reconciling Programming with Logic

The language developed in Section 5 departs from the full connection with linear logic through the inclusion of potentially divergent behavior (i.e. non-terminating internal, unobservable computations) in programs, which is inherently inconsistent with logical soundness. A practical consequence of this disconnect is that plugging together subsystems, e.g. as a result of dynamic channel passing or of linking downloaded (higher-order) code to local services as in the example of Section 5.1.1 may result in a system actually unable to offer its intended services due to divergent behavior, even if the component subsystems are well-typed and divergence-free, that is, the language no longer has the property of compositional non-divergence.

To recover the connections with logic and restore compositional non-divergence, while maintaining a reasonable degree of expressiveness, we restrict general recursive session types to their logically sound version of coinductive session types by eliminating general recursion and replacing it with corecursion (i.e. recursion that ensures productivity of the coinductive definition). We achieve this by putting in place certain syntactic constraints on the recursive definitions.

The nature of the syntactic constraints we impose on general recursion is in its essence similar to those used in dependently-typed programming languages such as Agda [35] or the language of the Coq proof assistant [49], however, the concurrent nature of process expressions introduces additional challenges. For instance, consider the following recursive definition:

\[
P : \text{int} \to \{c:\text{intStream}\}
c \leftarrow P n = \{ \text{output } c n
c' \leftarrow P (n+1)
x \leftarrow \text{input } c'
f\text{wd } c \leftarrow c' \}
\]
Here we have the definition of an integer stream that given an integer $n$ will output it along channel $c$. Afterwards, a recursive call is made that is bound to a local channel $c'$. We then input from $c'$ and conclude by forwarding between the two channels. It turns out that this seemingly simple recursive definition is not productive. A process interacting with $P$, after receiving the first integer would trigger an infinite sequence of internal actions in $P$. This argues that even seemingly harmless recursive definitions can introduce divergence in rather non-obvious ways.

For simplicity, we restrict recursive definitions to be functions with a monadic target type, essentially enforcing that (co)recursive definitions are coinductive process expressions with functional parameters, as one would expect when writing a recursive process expression in practice.

To fully attack the problem, we must first consider the meaning of productivity in our concurrent setting. The intuitive idea of productivity is that there must always be a finite sequence of internal actions between each observable action. In our setting, the observable actions are those on the session channel that is being offered, while internal actions are those generated through interactions with ambient sessions. Given our definition of productivity, a natural restriction is to require an action on the channel being offered before performing the recursive call (i.e. the recursive call must be guarded by a visible action). Thus, the definition,

$$
c <- S n = \{ \text{output } c \ n \ 
c' <- S (n+1) \}
$$

satisfies the guardedness condition and is in fact a valid coinductive definition of an increasing stream of integers. This is identical to the guardedness conditions imposed in Coq [20]. However, guardedness alone is not sufficient to ensure productivity. Consider our initial process definition:

$$
c <- P n = \{ \text{output } c \ n 
\c' <- P (n+1) 
\x <- \text{input } c' 
\text{fwd } c \ c' \}
$$

While guardedness is satisfied, the definition is not productive. A process interacting with $P$ will be able to observe the first output action, but no other actions are visible since they are consumed within $P$ by the input action that follows the recursive call. In essence, $P$ destroys its own productivity by interacting with the recursive call.

The problematic definition above leads us to consider restrictions on what may happen after a recursive call, which turn out to be rather subtle. For instance, we may impose that recursion be either terminal (i.e. the continuation must be a forwarder) or that after it an action on the observable channel must be performed, before interactions with the recursive call are allowed. While such a restriction indeed excludes the definition of $P$, it is insufficient:

$$
c <- P' n = \{ \text{output } c \ n 
\c' <- P' (n+1) 
\text{output } c \ n 
\x <- \text{input } c' 
\y <- \text{input } c' 
\text{fwd } c \ c' \}
$$

The definition above diverges after the two outputs, essentially for the same reason $P$ diverges. One could explore even more intricate restrictions, for instance imposing that the continuation be itself a process expression satisfying guardedness and moreover offering an action before interacting with its ambient context. Not only would this be a rather ad-hoc attempt, it is also not sufficient to ensure productivity.

Thus, our syntactic restrictions that ensure productivity are the combination of both guardedness and a condition we dub coregular recursion, that is, we require that after a recursive call the only valid continuation is a forwarder (between the channel of the recursive call and the one being offered). These two conditions combined are sufficient to ensure productivity and thus non-divergence, re-establishing the connections with logic. To formally obtain this result we must develop reasoning techniques that go beyond those used to establish type preservation and progress. More precisely, we develop a theory of linear logical relations that enables us to show non-divergence of well-typed programs, as well as setting up a framework to reason about program (and process expression) equivalence. We give an account of this development in Section 6.

However, one case that has yet to be made is what is the cost of these restrictions in terms of the kinds of programs that we are able to write that adhere to guardedness and coregular recursion. While we are obviously restricting...
the space of valid programs, we argue that the expressiveness of the resulting programming language is not overly
constrained by an extended example, consisting of an alternative implementation of the bit counter example of [53]
that adheres to the necessary restrictions to ensure non-divergence.

5.3.1 Example - Little Endian Bit Counter

Our goal is to implement a coinductive counter protocol offering three operations: we can poll the counter for its
current integer value; we can increment the counter value and we can terminate the counter. The protocol is encoded
via the following session type,

\[
\text{Counter} = \text{Choice}(\text{val}:\text{int} /\\text{Counter, inc:Counter, halt:1})
\]

Our implementation of \text{Counter} consists of a network of communicating processes, each process node encoding
a single bit of the binary representation of the counter value in little endian form (we can think of the leftmost process
in the network as representing the least significative bit). These network nodes communicate with their two adjacent bit
representations, whereas the top level coordinator for the counter communicates with the most significant bit process
(and with the client), spawning new bits as needed. The protocol for the nodes is thus given by the following session
type:

\[
\text{CImpl} = \text{Choice}\{\text{val}:\text{int} \Rightarrow \text{int} /\\text{CImpl},
\text{inc:Or}\{\text{carry:CImpl, done:CImpl}\}, \text{halt:1}\}
\]

The protocol is as follows: to compute the integer value of the bit network, each node expects to input the integer
value of the node to its right (i.e. the more significant bit), update this value with its own contribution and propagate it
down the bit network. When the last bit is reached, the final value is sent back through the network to the client. To
increment the value of the counter, given the little endian representation, we must propagate the increment command
down the network to the least significant bit, which will flip its value and either send a \text{carry} or a \text{done} message up
the chain, flipping the bits as needed (a 1 bit receiving \text{carry} will flip and propagate \text{carry}, whereas a 0 bit will flip
and send \text{done}, signalling no more bits need to be flipped). If the top level coordinator receives a \text{carry} message
we must generate a new bit node. The type and code for the node process is given below. Notice how \text{Node} is a valid
coinductive definition: recursive calls are guarded by an action on \(n\) and coregular.

\[
\text{Node} : \text{int} \rightarrow \{x:\text{CImpl} \mid n:\text{CImpl}\}
\]

\[
n <- \text{Node} \ b <- x = \{ \text{case} \ n \ of
\begin{align*}
\text{val} & \Rightarrow x.\text{val} \\
 & \text{m} <- \text{input} \ n \\
 & \text{output} x \ (2*\text{m}+b) \\
 & \text{m} <- \text{input} \ x \\
 & \text{output} \ n \ \text{m} \\
 & \text{n} <- \text{Node} \ b <- x
\end{align*}
\]

\[
\text{inc} \Rightarrow x.\text{inc}
\]

\[
\begin{align*}
\text{case} \ x \ of
\text{carry} \Rightarrow \text{if} \ (b=1) \ \text{then} \\
& \text{n.carry} \\
& \text{n} <- \text{Node} \ 0 <- x \\
\text{else}
& \text{n.done} \\
& \text{n} <- \text{Node} \ 1 <- x \\
\text{done} \Rightarrow \text{n} <- \text{Node} \ b <- x
\end{align*}
\]

\[
\text{halt} \Rightarrow x.\text{halt}
\]

\[
\text{wait} \ x
\]

\[
\text{close} \ n
\]

The coordinator process interfaces between the clients and the bit network, generating new bit nodes as needed.
Again, we note that \text{Coord} meets our criteria of a valid coinductive definition.

\[
\text{Coord} : \text{unit} \rightarrow \{x:\text{CImpl} \mid z:\text{Counter}\}
\]
The only remaining component is the representation of the empty bit string \( \epsilon \), which will be a closed process expression of type \( x : \text{impl} \), implementing the “base cases” of the protocol. For polling the value, it just ping-pongs the received value. For incrementing, it just signals the \( \text{carry} \) message. We can then put the \( \text{Counter} \) system together by composing \( \epsilon \) and \( \text{Coord} \).

\[
\epsilon : \text{unit} \to \{x : \text{CImpl}\}
\]

\[
x \leftarrow \epsilon () = \{ \text{case } x \text{ of } \]
\[
\quad \text{val} \Rightarrow n \leftarrow \text{input } x \quad \text{output } x n \\
\quad \text{inc} \Rightarrow x.\text{carry} \\
\quad \text{halt} \Rightarrow \text{close } x\}
\]

\[
z \leftarrow \text{Counter} = \{ x \leftarrow \epsilon () \\
\quad z \leftarrow \text{Coord} () \leftarrow x\}
\]

This example exhibits a more general pattern, where we have an interface protocol (in this case \( \text{Counter} \)) that is then implemented by a lower level protocol through some network of processes (\( \text{CImpl} \)), similar to abstract data types and implementation types, but here from the perspective of a concurrent communication protocol.

## 6 Reasoning Techniques

In the previous section I have mostly been concerned with connecting logic and concurrent computation by identifying concurrent phenomena that can be justified using our linear logical foundation, or dually, logical aspects that can be given a concurrent interpretation.

As I have argued from the start, a true logical foundation must also be able to provide with techniques for reasoning about concurrent computation, not just be able to give them a logical meaning. This is particularly important because reasoning about concurrent computation is notoriously hard. For instance, even establishing a seemingly simple property such as progress in traditional session types \( [13] \) already requires complex technical machinery, whereas in this logical interpretation we obtain this property “for free” \( [8, 50, 37] \). More sophisticated and important properties such as termination \( [56] \) and confluence pose even harder challenges (no such results even exist for session type systems). One of the existing main issues is the lack of a uniform proof technique that is robust enough to handle all these properties of interest.

Beyond the actual formal machinery required to prove such properties, if we consider the techniques commonly used to reason about concurrent behavior we typically rely on behavioral equivalence techniques such as bisimulations \( [57, 45] \). In a typed setting, it turns out to be quite challenging to develop bisimulations that are consistent with the canonical notion of behavioral (or observational) equivalence of barbed congruence. This is even more so the case in the higher-order setting \( [42, 46] \). Again, as in the case for proof techniques, these bisimulations are very specifically tuned to the particular language being considered and while common patterns arise, there is no unified framework that guides the development of these bisimulations.
We address these issues by developing a theory of linear logical relations. We apply the ideas of logical relations to our concurrent interpretation of linear logic, developing a rich framework which enables reasoning about concurrent programs in a uniform way, allowing us to establish sophisticated properties such as termination and confluence in a very natural and uniform way. The logical foundation also allows us to extend the linear logical relations beyond simple session types, accounting also for polymorphism and coinductive session types. Moreover, beyond serving as a general proof technique for session-typed programs, our linear logical relations, when extended to the binary case, naturally arise as typed bisimulations which we can show to coincide with barbed congruence in the polymorphic and simply-typed settings. These results establish linear logical relations as a general reasoning technique for concurrent session-typed computation, providing the remaining missing feature that supports our claim of linear logic as a foundation of message passing computation.

In the remainder of this section we develop the basic idea of linear logical relations for the session typed $\pi$-calculus language that coincides with our logical interpretation and show how the technique can be applied to show termination in the polymorphic and coinductive session-typed setting. We also develop the notion of process equivalence naturally induced by the binary version of the logical relations technique in the polymorphic setting, which entails parametricity and coincides with the canonical notion of observational equivalence in process calculi of (typed) barbed congruence. Finally, we discuss the idea of equivalence in our coinductive session-typed programming language and how it can serve as a form of extensional equality in a full dependent session type theory.

6.1 Linear Logical Relations

We develop our theory of linear logical relations by building on the well-known reducibility candidates technique of Girard [23]. The general outline of the development consists of defining a notion of reducibility candidate at a given type, which is a predicate on well-typed terms satisfying some crucial closure conditions. Given this definition, we can then define a logical predicate on well-typed terms by induction on the type (more precisely, first on the size of the typing contexts and then on the right-hand-side typing).

We will not produce the full development of the linear logical relations here, deferring the interested reader to [37] and [6] for the details on the propositional and polymorphic settings, respectively. We highlight that our linear logical relations are defined on (typed) $\pi$-calculus processes. The reasoning behind this is twofold: first, it more directly enables us to explore the connections of our logical relations to usual process calculi techniques of bisimulations and barbed congruence, providing a more familiar setting in which to explore these ideas; secondly, it allows us to more deeply explore the $\pi$-calculus term assignment and take advantage of certain features that turn out to be technically convenient such as labelled transitions. The reason why labelled transitions are particularly convenient is because they allow us to reason about the observable behavior of a process as it interacts along its distinguished session channel, allowing for a systematic treatment of closed processes (while such a transition system could be defined for our SSOS, it already exists for the $\pi$-calculus assignment and so there’s no reason to avoid it). For reference, the particular labelled transition system we consider is given in Fig. 5. It is the so-called early labelled transition system for $\pi$-calculus processes, extended with the appropriate labels for binary choice.

One interesting aspect of our development is how cleanly it matches the kind of logical relations arguments typically defined for $\lambda$-calculi. In particular, the conditions imposed by our definition of reducibility candidate are fundamentally the same.

We now outline the logical relations argument for showing termination by defining, as expected, a logical predicate on typed processes. Our particular case study is for the setting of impredicative polymorphism since it aptly showcases the main technical aspects of the development, which in the general case consists of a logical relation that is equipped with some form of additional mappings that treat type variables as appropriate. Below we state that a process $P$ terminates (written $P \upharpoonright \bot$) if there is no infinite reduction sequence starting with $P$. We write $\equiv_1$ (resp. $\equiv_\alpha \Rightarrow$) for the reflexive transitive closure of reduction (resp. labelled transitions) on $\pi$-calculus processes.

The logical predicate uses the following extension to structural congruence with the so-called sharpened replication axioms [44].

**Definition 1.** We write $\equiv_1$ for the least congruence relation on processes which results from extending structural congruence $\equiv$ with the following axioms:

1. $(\nu u)(!u(z).P \mid (\nu y)(Q \mid R)) \equiv_1 (\nu y)((\nu u)(!u(z).P \mid Q) \mid (\nu u)(!u(z).P \mid R))$
Definition 2 (Reducibility Candidate). Given a type \( A \) and a name \( z \), a reducibility candidate at \( z : A \), written \( R[z : A] \), is a predicate on all processes \( P \) such that \( \vdash P : z : A \) and satisfy the following:

1. If \( P \in R[z : A] \) then \( P \downarrow \).
2. If \( P \in R[z : A] \) and \( P \Rightarrow P' \) then \( P' \in R[z : A] \).
3. If for all \( P_i \) such that \( P \Rightarrow P_i \) we have \( P_i \in R[z : A] \) then \( P \in R[z : A] \).

As in the functional case, the properties required for our reducibility candidates are termination (1), closure under reduction (2), and closure under backward reduction (or head expansion) (3).

Given the definition of reducibility candidate we then define our logical predicate as an inductive definition. Intuitively, the logical predicate captures the terminating behavior of processes as induced by typing. This way, e.g., the meaning of a terminating process of type \( z : \forall X . A \) is that after inputing an arbitrary type \( B \), a terminating process of type \( z : A \{ B / X \} \) is obtained. As we consider impredicative polymorphism, the main technical issue is that \( A \{ B / X \} \) may be larger than \( \forall X . A \), for any measure of size. This issue is solved in the same way as in the strong normalization proof for System F, by parameterizing the predicate with mappings from type variables to types and from type variables to reducibility candidates at the appropriate types. This idea of a parameterized predicate is quite useful in the general case, not only for the polymorphism setting but also, as we will see, for the coinductive setting.

It is instructive to compare the key differences between our development and the notion of logical relation for functional languages with impredicative polymorphism, such as System F (or type variables in general). In that context, types are assigned to terms and thus one maintains a mapping from type variables to reducibility candidates at the appropriate types. In our setting, since types are assigned to channel names, we need the ability to refer to reducibility candidates at a given type at channel names which are \( \textit{yet to be determined} \). Therefore, when we quantify over types and reducibility candidates at that type, intuitively, we need to “delay” the choice of the actual name along which the candidate must offer the session type. A reducibility candidate at type \( A \) which is “delayed” in this sense is denoted as \( R[\_ : A] \), where \( \_ \) stands for a name to be instantiated later on.

For the particular setting of polymorphism, the logical predicate is parameterized by two mappings, \( \omega \) and \( \eta \). Given a context of type variables \( \Omega \), we write \( \omega : \Omega \) to denote that \( \omega \) is an assignment of closed types to variables in \( \Omega \). We
write $\omega[X \mapsto A]$ to denote the extension of $\omega$ with a new mapping of $X$ to $A$. We use a similar notation for extensions of $\eta$. We write $\hat{\omega}(P)$ (resp. $\hat{\omega}(A)$) to denote the application of the mapping $\omega$ to free type-variables in $P$ (resp. in $A$). We write $\eta : \omega$ to denote that $\eta$ is an assignment of functions taking names to reducibility candidates, to type variables in $\Omega$ (at the types in $\omega$).

The logical predicate itself is defined as a sequent-indexed family of process predicates: a set of processes $T^\omega_{\eta}[\Gamma; \Delta \vdash T]$ satisfying some conditions is assigned to any sequent of the form $\Omega; \Gamma; \Delta \vdash T$, provided both $\omega:\Omega$ and $\eta:\omega$. The predicate is defined inductively on the structure of the sequents: the base case considers sequents with an empty left-hand side typing (abbreviated $T^\omega_{\eta}[T]$), whereas the inductive case considers arbitrary typing contexts and relies on principles for process composition.

**Definition 3** (Logical Predicate - Inductive Case). For any sequent $\Omega; \Gamma; \Delta \vdash T$ with a non-empty left hand side environment, we define $T^\omega_{\eta}[\Gamma; \Delta \vdash T]$ (with $\omega : \Omega$ and $\eta:\omega$) as the set of processes inductively defined as follows:

$$P \in T^\omega_{\eta}[\Gamma; \Delta \vdash T] \text{ iff } \forall R \in T^\omega_{\eta}[y:A], (\nu y)(\hat{\omega}(R) | \hat{\omega}(P)) \in T^\omega_{\eta}[\Gamma; \Delta \vdash T]$$

$$P \in T^\omega_{\eta}[u:A, \Gamma; \Delta \vdash T] \text{ iff } \forall R \in T^\omega_{\eta}[y:A], (\nu u)(\hat{\omega}(R) | \hat{\omega}(P)) \in T^\omega_{\eta}[\Gamma; \Delta \vdash T]$$

Definition 3 above enables us to focus on closed processes by inductively “closing out” open processes according to the proper composition forms. While the definition above is specific to the polymorphic case due to the $\eta$ mappings, it exhibits a general pattern in our framework where we take an open well-typed process and “close it” by composition with closed processes in the predicate at the appropriate type, allowing us to focus on the behavior of closed processes, which takes place on the channel along which they offer a session.

The definition of the predicate for closed processes captures the behavior of a well-typed process, as specified by the session it is offering. We do this by appealing to the appropriate labelled transitions in the $\pi$-calculus. For instance, to define the logical predicate for processes offering a session of type $z : A \rightarrow B$ we specify that such a process must be able to input a channel $y$ along $z$ such that the resulting process $P'$, when composed with a process $Q$ that offers $A$ along $y$ (and is in the predicate), is in the predicate at the type $z : B$, which is precisely the meaning of a session of type $A \rightarrow B$. Similarly, the predicate at type $z : 1$ specifies that the process must be able to internally transition to a process that is structurally equivalent (under the extended notion of structural equivalence which “garbage collects” replicated processes that are no longer used) to the inactive process. Type variables are accounted for in the predicate by appealing to the appropriate mappings. In the case for polymorphism this simply means looking up the $\eta$ mapping for the particular type variable, instantiating the channel name accordingly. To define the predicate at type $z : \forall X.A$ we quantify over all types and over all candidates at the given type, such that a process offering $z : \forall X.A$ must be able to input any type $B$ and have its continuation be in the predicate at $z : A$, with the mappings extended accordingly. The case for the existential is dual. This quantification over all types and candidates matches precisely the logical relations argument for impredicative polymorphism in the functional setting. The formal definition of the base case for the logical predicate is given in Definition 4.

**Definition 4** (Logical Predicate - Base Case). For any session type $A$ and channel $z$, the logical predicate $T^\omega_{\eta}[z:A]$ is inductively defined by the set of all processes $P$ such that $\vdash \hat{\omega}(P) :: z\hat{\omega}(A)$ and satisfy the conditions in Figure 6.

As usual in proofs using logical relation arguments, the burden of proof lies in showing that all well-typed terms (in this case, processes) are in the logical predicate at the appropriate type, from which termination follows as a corollary. The details of the proof can be found in [6].

**Theorem 3** (Fundamental Theorem). If $\Omega; \Gamma; \Delta \vdash P :: T$ then, for all $\omega : \Omega$ and $\eta : \omega$, we have that $\hat{\omega}(P) \in T^\omega_{\eta}[\Gamma; \Delta \vdash T]$.

### 6.1.1 Coinductive Session Types

One of the key advantages of the framework that we have presented in the previous section is its uniformity in accounting for several forms of session types. We make this explicit by giving a succinct account of how to develop the termination argument for coinductive session types in the sense of Section 5.3 using linear logical relations (a full account of this formulation is given in [52]). To distinguish between the two definitions, here we use the notation $L^\omega[x:A]$. 

32
\( P \in T^\omega[z:X] \) \iff \( P \in \eta(X)(z) \)
\( P \in T^\omega[z:1] \) \iff \( \forall P'. (P \implies P' \land P' \not\rightarrow \implies P' \equiv \top) \)
\( P \in T^\omega[z:A \rightarrow B] \) \iff \( \forall P'. (P \rightarrow P') \implies \forall Q \in T^\omega[y:A].(\nu y) (P' \mid Q) \in T^\omega[z:B] \)
\( P \in T^\omega[z:A \otimes B] \) \iff \( \forall P'. (P \rightarrow P') \implies \exists P_1. P_2. (P' \equiv_1 P_1 \land P_2 \in T^\omega[y:A] \land P_2 \in T^\omega[z:B]) \)
\( P \in T^\omega[z!:A] \) \iff \( \forall P'. (P \equiv_1 P') \implies \exists P_1. (P' \equiv_1 !z(y).P_1 \land P_1 \in T^\omega[y:A]) \)
\( P \in T^\omega[z:A \& B] \) \iff \( \forall P'. (P \rightarrow P') \implies P' \in T^\omega[z:A] \land \forall P'. (P \rightarrow P') \implies P' \in T^\omega[z:B] \)
\( P \in T^\omega[z:X.A] \) \iff \( \forall B, P', R[\rightarrow :B]. (B \land P \equiv_{\pi(B)} P') \implies P' \in T^\omega[X \rightarrow R[\rightarrow :B]][z:A] \)
\( P \in T^\omega[z:\exists X.A] \) \iff \( \exists B, R[\rightarrow :B]. (B \land P \equiv_{\pi(B)} P') \implies P' \in T^\omega[X \rightarrow R[\rightarrow :B]][z:A] \)

Figure 6: Logical predicate (base case).

Since the language includes \( \lambda \)-terms, we must also define the usual notion of candidate and logical predicate for functional terms, including for terms of monadic type. This makes the definition of the logical predicate mutually inductive, given that we need to simultaneously define the predicate for processes and \( \lambda \)-terms (since we appeal to the functional predicate in the cases of value communication and to the process predicate in the case for the contextual monadic type).

The particulars of the logical predicate in the coinductive setting essentially consist of the way in which we handle type variables and, naturally, the definition of the logical predicate for coinductive session types. Unlike in the polymorphism setting, where a type variable could be instantiated with any type, a type variable in a coinductive session type always stands for the coinductive session type itself. Thus, we no longer need to track which particular type we are instantiating a type variable with and only need to extend the predicate with a single mapping \( \omega \), which maps type variables to reducibility candidates (at the appropriate type), encoding the unfolding of the coinductive type.

The definition of candidate for \( \lambda \)-terms is the expected one, essentially consisting of a functional analogue of Definition 2 (which is unchanged for processes). The logical predicate, as usual, is defined inductively on types. The only new clause in the functional case is the one for the contextual monadic type, which simply requires terms to extend the predicate with a single mapping \( \omega \), and naturally, the definition of the logical predicate for coinductive session types. Unlike in the polymorphic setting, where we need to simultaneously define the predicate for processes and \( \lambda \)-terms, we appeal to the functional predicate in the cases of value communication and to the process predicate in the case for the contextual monadic type.

\[
\mathcal{L}^\omega[\{a:A \mid c:A\}] \triangleq \{ M \mid M \implies a \leftarrow \{ P \leftarrow \bar{\sigma} \text{ and } P \in \mathcal{L}^\omega[\text{shd}(a:A_i); \text{lin}(a:A_i) \implies a:A]\} \}
\]

For processes, the significant cases are those that pertain to type variables and coinductive session types. We define the logical interpretation of a coinductive session type \( \nu X.A \) as the union of all reducibility candidates \( \Psi \) of the appropriate coinductive type that are in the predicate for type \( A \), when \( X \) is mapped to \( \Psi \) itself. The technical definition is somewhat intricate, but the key observation is that for open types, we may view our logical predicate as a mapping between sets of reducibility candidates, of which the interpretation for coinductive session type turns out to be a greatest fixpoint:

\[
\mathcal{L}^\omega[z:\nu X.A] \triangleq \bigcup \{ \Psi \in \mathcal{R}[\nu X.A] \mid \Psi \subseteq \mathcal{L}^\omega[\text{shd}[a:A_i]; \text{lin}(a:A_i) \implies a:A]\}
\]

In order to show the fundamental theorem of logical relations for coinductive session types, we must first establish the fixpoint property mentioned above.

**Definition 5.** Let \( \nu X.A \) be a strictly positive type. We define: \( \phi_A(s) \triangleq \mathcal{L}^\omega[\nu X.A][z:A] \)

**Theorem 4 (Greatest Fixpoint).** \( \mathcal{L}^\omega[z:\nu X.A] \) is a greatest fixpoint of \( \phi_A \).
Theorem 5 (Fundamental Lemma). If \( \Psi \models M : \tau \) and \( \Psi; \Gamma; \Delta \Rightarrow P : z : A \) then for any mapping \( \psi \) s.t. \( x : \tau_0 \in \Psi \) iff \( \psi(x) \subseteq L[\tau_0] \) we have that \( \psi(M) \subseteq L[\tau] \) and \( P \subseteq L[\Psi; \Gamma; \Delta \Rightarrow z : A] \).

The proof proceeds by mutual induction on the given typing derivation. There are some technical challenges involved that we omit here for the sake of presentation. The key case is the one for corecursive definitions, of which we outline the main points of the proof. The proof consists of producing a set of processes \( C_P \), containing \( P \), show that \( C_P \) is a reducibility candidate at \( \nu X.A \) and that \( C_P \subseteq L^{\omega}[X \mapsto C_P][z : A] \). This is a sufficient condition since we established that \( L[\nu\nu X.A] \) is the largest such set. Showing that \( C_P \) satisfies this condition relies crucially on coregular recursion and guardedness, which intuitively enforces that the unfolding of the coinductive type is offered by the recursive call, followed by a forwarder, reducing to a process in \( C_P \) (roughly, \( P \) itself).

6.2 Equivalence

As we argue in the beginning of Section 6, one of the advantages of the framework of linear logical relations is that not only can it be used as a proof technique to show rather non-trivial properties such as termination (the case for which we established above) but it also enables us to reason about program (or process) equivalence by considering the binary case of the logical relation, that is, lifting from a predicate on processes to a binary relation.

In the polymorphism setting, this entails parametricity in the sense of Reynolds’ abstraction theorem \(^{[41]} \), a concept which was previously unexplored in session typed settings. Moreover, the equivalence relation that arises naturally from the move to the binary setting turns out to coincide with the canonical notion of (typed) observational equivalence on \( \pi \)-calculus processes, commonly known as barbed congruence.

For the sake of conciseness we omit the full technical development of these results, which is carried out in \(^{[6]} \). Instead, we highlight the key insights of the development.

First, we introduce the notion of barbed congruence from process calculi. The basic idea is to identify a canonical notion of observational equivalence that is not tied directly to a labelled transition system, since many such systems can be defined for a given calculus, but rather to the reduction semantics which is a more satisfactory operational semantics insofar as there is little discussion over what is the correct definition of reduction for process calculi. Thus, barbed congruence consists of a binary relation on processes that relies on the notion of a barb. A barb is the most basic observable on the behavior of processes, consisting simply of taking an action on a channel. For instance, the process \( x(M).P \) has an (output) barb on \( x \). Typically we distinguish between different kinds of barbs for convenience (e.g. output, input and selection).

Given the notion of a barb, there is a natural definition barbed equivalence: two processes are barbed equivalent if they have the same barbs. Since we are interested in the so-called weak variants of equivalence (those that ignore differences at the level of internal actions), we can define weak barbed equivalence as an extension of barbed equivalence that is closed under reduction, that is, if \( P \) is weak barbed equivalent to \( Q \) then not only do they have the same barbs, but also every reduction of \( P \) is matched by zero or more reduction in \( Q \) (and vice-versa). Finally, we want our definition of observational equivalence to capture the intuition that two processes deemed observationally equivalent cannot be distinguished from one another. This is made precise by defining barbed congruence (written \( \cong \)) as the closure of weak barbed equivalence under any process context, that is, given a process with a “hole” which can be instantiated (i.e. a context), two processes are deemed barbed congruent if they are weak barbed equivalent when placed in any context. In a typed setting, we restrict our attention to processes of the same type, and to well-formed (according to typing) process contexts.

Given our intuitive notion of barbed congruence, it is easy to see why it is not straightforward how to determine if two processes are indeed barbed congruent (and so, observationally equivalent), since we are quantifying over all possible process contexts. So, the typical approach is to devise a sound and complete proof technique for barbed congruence that does not rely on such quantifications. This is usually achieved by defining some bisimulation (appealing to the labelled transition semantics) on processes and then showing it to be a congruence, which is not a straightforward task in general.

So how can we use our linear logical relations framework to address the problem of identifying a proof technique for observational equivalence? In the functional setting, we can define a notion of observational equivalence that is analogous to the idea of barbed congruence and then develop a notion of logical equivalence, obtained by extending the unary logical predicate to the binary case, that characterizes observational equivalence. Moreover, in the polymorphic
setting this logical equivalence also captures the idea of *parametricity*. It turns out that a similar development can be achieved in our session typed setting.

To move to a binary or *relational* version of our logical predicate, we first need the notion of an *equivalence candidate* which is the binary analogue of a reducibility candidate, but instead of enforcing termination, it consists of a binary relation on typed processes that is *closed* under barbed congruence.

**Definition 6 (Equivalence Candidate).** Let $A, B$ be types. An equivalence candidate $\mathcal{R}$ at $z: A$ and $z: B$, noted $\mathcal{R} :: z: A \leftrightarrow B$, is a binary relation on processes such that, for every $(P, Q) \in \mathcal{R} :: z: A \leftrightarrow B$ both $\cdot \vdash P :: z: A$ and $\cdot \vdash Q :: z: B$ hold, together with the following conditions:

1. If $(P, Q) \in \mathcal{R} :: z: A \leftrightarrow B$, then $\cdot \vdash P \equiv P' :: z: A$, and $\cdot \vdash Q \equiv Q' :: z: B$ then $(P', Q') \in \mathcal{R} :: z: A \leftrightarrow B$.

We often write $(P, Q) \in \mathcal{R} :: z: A \leftrightarrow B$ as $P \mathcal{R} Q :: z: A \leftrightarrow B$.

Given this notion of candidate we then define logical equivalence by considering the binary version of the logical predicate of Section 6.1 written $\Gamma; \Delta \vdash P \approx L Q :: T[\eta : \omega \leftrightarrow \omega']$, where both $P$ and $Q$ are typed in the same contexts and right-hand side typing, $\omega$ and $\omega'$ as are before, mappings from type variables to types (the former applied to $P$ and the latter to $Q$) and $\eta$ is now a mapping from type variables to equivalence candidates that respects $\omega$ and $\omega'$ (written $\eta : \omega \leftrightarrow \omega'$). Just as before, we follow the same ideas of inductively closing out processes and then focusing on closed processes.

**Definition 7 (Logical Equivalence - Inductive Case).** Let $\Gamma, \Delta$ be non empty typing environments. Given the sequent $\Omega; \Gamma; \Delta \vdash T$, the binary relation on processes $\Gamma; \Delta \vdash P \approx L Q :: T[\eta : \omega \leftrightarrow \omega']$ (with $\omega, \omega' : \Omega$ and $\eta : \omega \leftrightarrow \omega'$) is inductively defined as:

$$
\begin{align*}
\Gamma; \Delta, y : A \vdash P \approx L Q :: T[\eta : \omega \leftrightarrow \omega'] & \quad \text{iff} \quad \forall R_1, R_2, \text{s.t.} \ R_1 \approx L R_2 :: y : A[\eta : \omega \leftrightarrow \omega'], \\
\Gamma; \Delta \vdash (vy)(\bar{\omega}(P)) \approx L (vy)(\bar{\omega}'(Q)) & \quad \text{iff} \quad \forall R_1, R_2, \text{s.t.} \ R_1 \approx L R_2 :: y : A[\eta : \omega \leftrightarrow \omega'], \\
\Gamma, u : A; \Delta \vdash P \approx L Q :: T[\eta : \omega \leftrightarrow \omega'] & \quad \text{iff} \quad \forall R_1, R_2, \text{s.t.} \ R_1 \approx L R_2 :: y : A[\eta : \omega \leftrightarrow \omega'], \\
\Gamma; \Delta \vdash (vy)(\bar{\omega}(P)) \circ !u(y).\bar{\omega}(R_1)) \approx L (vy)(\bar{\omega}'(Q)) & \quad \text{iff} \quad \forall R_1, R_2, \text{s.t.} \ R_1 \approx L R_2 :: y : A[\eta : \omega \leftrightarrow \omega'].
\end{align*}
$$

The formalism is quite verbose, but the underlying ideas are really just straightforward generalizations of the logical predicate we previously introduced. For instance, in the logical predicate of Section 6.1, the case for $z: A \to B$ was defined as being able to observe an input action $z(x)$ on a process, for which the resulting continuation, when composed with any process in the predicate at type $x: A$ would be in the predicate at type $z: B$. The definition of logical equivalence follows exactly this pattern: two processes $P$ and $Q$ are logically equivalent at type $z: A \to B$ when we are able to identify a (weak) input action $z(x)$ on both processes, such that for any logically equivalent processes $R_1$ and $R_2$ at type $x: A$, the respective continuations, when composed respectively with $R_1$ and $R_2$ will be logically equivalent at type $z: B$. The remaining cases follow a similar generalization pattern. The precise definition (omiting the additives for conciseness) is given below (Definition 8).

**Definition 8 (Logical Equivalence - Base Case).** Given a type $A$ and mappings $\omega, \omega', \eta$, we define logical equivalence, noted $P \approx L Q :: z: A[\eta : \omega \leftrightarrow \omega']$, as the largest binary relation containing all pairs of processes $(P, Q)$ such that (i) $\cdot \vdash \bar{\omega}(P) :: z: \bar{\omega}(A)$; (ii) $\cdot \vdash \bar{\omega}'(Q) :: z: \bar{\omega}'(A)$; and (iii) satisfies the conditions in Figure 2.

Equipped with our notion of logical equivalence, we can then show the fundamental result of *relational parametricity* and with it, that logical equivalence is a sound and complete proof technique for (typed) barbed congruence.

**Theorem 6 (Relational Parametricity).** If $\Omega; \Gamma; \Delta \vdash P :: z: A$ then, for all $\omega, \omega' : \Omega$ and $\eta : \omega \leftrightarrow \omega'$, we have $\Gamma; \Delta \vdash \bar{\omega}(P) \approx L \omega'(P) :: z: A[\eta : \omega \leftrightarrow \omega']$.

Parametricity states that a process is always logically equivalent to itself, for any possible (consistent) instantiation of the type variables present in its type. It is quite remarkable how much the development of parametricity in our concurrent, session-typed setting mirrors that of polymorphic functional programming languages. Where in the latter parametricity captures the intuitive idea that a polymorphic function must behave uniformly, regardless of the particular
\[ P \approx_L Q :: z:X[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad (P, Q) \in \eta(X)(z) \]
\[ P \approx_L Q :: z:1[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad \forall P', Q'. (P \implies P' \land P' \not\rightarrow \land Q \implies Q' \land Q' \not\rightarrow) \Rightarrow \]
\[ (P' \equiv_0 0 \land Q' \equiv_0 0) \]
\[ P \approx_L Q :: z:A \rightarrow B[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad \forall P', y. (P \xrightarrow{(y)} P') \Rightarrow \exists Q'. Q \xrightarrow{(y)} Q' \text{ s.t.} \]
\[ \forall R_1, R_2. \quad R_1 \approx_L R_2 :: y:A[\eta : \omega \leftrightarrow \omega'] \]
\[ \nu y)(P' | R_1) \approx_L (\nu y)(Q' | R_2) :: z:B[\eta : \omega \leftrightarrow \omega'] \]
\[ P \approx_L Q :: z:A \odot B[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad \forall P', y. (P \xrightarrow{(y)} P') \Rightarrow \exists Q'. Q \xrightarrow{(y)} Q' \land \quad \text{ s.t.} \]
\[ \forall R_1, R_2, n. \quad y:A \vdash R_1 \approx_L R_2 :: n:1[\eta : \omega \leftrightarrow \omega'] \]
\[ (\nu y)(P' | R_1) \approx_L (\nu y)(Q' | R_2) :: z:B[\eta : \omega \leftrightarrow \omega'] \]
\[ P \approx_L Q :: z:A[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad \forall P'. (P \xrightarrow{z(y)} P') \Rightarrow \exists Q'. Q \xrightarrow{z(y)} Q' \land \quad \text{ s.t.} \]
\[ \forall R_1, R_2, n. \quad y:A \vdash R_1 \approx_L R_2 :: n:1[\eta : \omega \leftrightarrow \omega'] \]
\[ (\nu y)(P' | R_1) \approx_L (\nu y)(Q' | R_2) :: z:A[\eta : \omega \leftrightarrow \omega'] \]
\[ P \approx_L Q :: z:X.A[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad \exists B_1, B_2, P', R :: \vdash B_1 \iff B_2. \quad (P \xrightarrow{(B_1)} P') \Rightarrow \]
\[ \exists Q'. Q \xrightarrow{(B_2)} Q', P' \approx_L Q' :: z:A[\eta[X \rightarrow R]] : \omega[X \rightarrow B_1] \leftrightarrow \omega'[X \rightarrow B_2] \]
\[ P \approx_L Q :: z:X.A[\eta : \omega \leftrightarrow \omega'] \quad \text{iff} \quad \exists B_1, B_2, R :: \vdash B_1 \iff B_2. \quad (P \xrightarrow{(B)} P') \Rightarrow \]
\[ \exists Q'. Q \xrightarrow{(B)} Q', P' \approx_L Q' :: z:A[\eta[X \rightarrow R]] : \omega[X \rightarrow B_1] \leftrightarrow \omega'[X \rightarrow B_2] \]

Figure 7: Logical equivalence (base case) – Abridged.

instantiations of the polymorphic type variable; here, parametricity captures a notion of communication protocol independence, or genericity, where a process implementing a polymorphic session type must do so in a way that is independent of the particular session protocol that will instantiate the type variable.

**Theorem 7** (Logical Equivalence and Barbed Congruence coincide). **Relations \( \approx_L \) and \( \equiv \) coincide for well-typed processes. More precisely:**

1. If \( \Gamma; \Delta \vdash P \approx_L Q :: z:A[\eta : \omega \leftrightarrow \omega'] \) holds for any \( \omega, \omega' : \Omega \) and \( \eta : \omega \leftrightarrow \omega' \), then \( \Omega; \Gamma; \Delta \vdash P \equiv Q :: z:A \)
2. If \( \Omega; \Gamma; \Delta \vdash P \cong Q :: z:A \) then \( \Gamma; \Delta \vdash P \approx_L Q :: z:A[\eta : \omega \leftrightarrow \omega'] \) for some \( \omega, \omega' : \Omega \) and \( \eta : \omega \leftrightarrow \omega' \).

Theorem 7 establishes a definitive connection between the usual barb-based notion of observational equivalence from concurrency theory, and the logical equivalence induced by our logical relational semantics. Once again, this supports our claim that our linear logical relations provide a robust, uniform framework for reasoning about session-typed programs and their behavior.

### 6.2.1 Coinductive and Dependent Session Types

Just as moving from the unary to the binary setting in the polymorphic setting gives rise to a logical characterization of observational equivalence, it is expectable that doing so in the coinductive session typed setting will produce similar results.

This is particularly interesting for two main reasons: our language of coinductive session types also includes higher-order features and, from the perspective of concurrency theory, the behavioral theory of higher-order processes is particularly intricate [46] and so a logically grounded approach to this problem could provide elegant solutions to this problem.

Secondly, the monadic nature of our language with coinductive session types seems to lend itself naturally to a full dependent session type theory, where we can have not only session types that depend on functional terms (such as that of Section 4.1) but also on process expressions through the contextual monad, and functional types that depend
on monadic values. The motivation for the development of a full dependent session type theory would be to obtain a uniform language in which to express and formally reason about session-typed concurrent computation.

As usual in dependent type theories, most of the burden is placed on the type conversion rule, which appeals to equality of types. Ultimately, equality of types reduces to equality of terms and therefore for such a dependent type theory to exist, we must crucially develop a suitable notion of equality of process expressions. For typechecking to be decidable, this equality should be of an intensional flavor and therefore sound wrt the extensional equivalence induced by our linear logical relations.

The combination of dependent and coinductive session types enables us to express particularly rich properties of processes. For instance, by defining session type families indexed by monadic objects we can specify bisimilarity arguments within the language, for which the proof objects are process expressions themselves. To make this slightly more concrete, consider two alternative implementations of the stream of all natural numbers starting from a given natural:

\[
nats : \text{nat} \to \{c: \text{natStream}\}
\]

\[
nats' : \text{nat} \to \{c: \text{natStream}\}
\]

With dependent types in the sense discussed above, we could encode a coinductive proof that the two implementations are indeed the same by defining a type family indexed by the two streams (we write \(eq\) for the propositional equality type):

\[
equiv : \{c: \text{natStream}\} \to \{c: \text{natStream}\} \to \text{stype}.
\]

\[
\begin{align*}
stype \; \text{bisim} \; P \; Q & = \\
& \exists n: \text{nat} \exists m: \text{nat} \exists p: n=m. \\
& \exists P': \{c: \text{natstream}\}. \\
& \exists Q': \{c: \text{natstream}\}. \\
& \exists _: \text{eq} \; P \{\text{output} \; c \; n ; \; P'\} \\
& \exists _: \text{eq} \; Q \{\text{output} \; c \; m ; \; Q'\} \\
& \equiv \; P' \; Q'.
\end{align*}
\]

\[
bisim\_proof : \Pi n: \text{nat}. \{c: \text{bisim} \; (nats \; n) \; (nats' \; n)\}
\]

\[
c \leftarrow \text{bisim\_proof} \; n = \\
\{ \\
\text{output} \; c \; n \\
\text{output} \; c \; n \\
\text{output} \; c \; \text{refl}(n) \\
\text{output} \; c \; (\text{nats} \; (n+1)) \\
\text{output} \; c \; (\text{nats'} \; (n+1)) \\
\text{output} \; c \; \text{refl}(\text{nats} \; n) \\
\text{output} \; c \; \text{refl}(\text{nats'} \; n) \\
c \leftarrow \text{bisim\_proof} \; n+1 \\
\}
\]

Naturally, the kinds of examples of Section 4.1 could also be combined with coinductive types to produce rich specifications where we impose constraints on the values produced by processes that implement coinductive session types (e.g. specify the values of a stream in the type, assert by typing that a certain process network computes some predetermined function, etc.).

While we have yet to fully flesh out these ideas, we believe it should be possible to develop the necessary equalities that make this form of dependent type theory work, for which our linear logical relations serve as a basis for an extensional equality.

7 Conclusion

In this proposal, I have presented a formulation of intuitionistic linear logic as a logical basis for session-based, message passing concurrency. To support this claim I have developed the basic interpretation of propositional linear logic
as a session-typed concurrent language and expanded this interpretation to both the first and second-order settings, providing logically based accounts of value dependent and polymorphic session types, the former of which I have also combined with notions of proof irrelevance and affirmation to give a high-level account of proof-carrying code in a concurrent, session based setting. To further support the concept of intuitionistic linear logic as a logical foundation for session-based concurrency, I used the interpretation as the basis for a programming language combining session-based concurrency and functional programming using a contextual monad, resulting in an expressive language that supports higher-order concurrency (viz. higher-order process calculi) and general recursion, with session fidelity and deadlock freedom by typing. I have also studied the necessary restrictions to general recursion that re-establish the connections with logic by eliminating divergence from the language.

Finally, I develop a theory of linear logical relations for session typed \( \pi \)-calculus processes that can be used as a uniform proof technique to establish sophisticated properties such as termination and confluence, as well as provide reasoning techniques by providing a natural notion of logical equivalence on process terms, shown to coincide with the canonical notion of (typed) barbed congruence in the polymorphic setting. I also sketch how this notion of logical equivalence can be extended to the monadic/higher-order setting, providing an extensional equality on process expressions that may be used as a soundness criteria for intensional equality in a full dependent session type theory.

In this section I will return to a brief discussion of related work, outline the structure of the thesis I am proposing and discuss my general work plan.

7.1 Related Work

While Girard himself foreshadowed the connections of linear logic and concurrency, the first work to flesh out the technical details of this connection is that of Abramsky [1], relating (classical) linear logic to a \( \lambda \)-calculus with parallel composition. This interpretation was later refined by Belin and Scott [2] to a \( \pi \)-calculus with some non-standard prefix commutations necessary to fully capture all the commuting conversions of linear logic. The explicit connection to session types was first developed by Caires and Pfenning [5], proposing a full correspondence in the sense of Curry-Howard but not identifying all proof conversions of linear logic as process reductions (and thus not requiring the prefix commutations of the work of Belin and Scott). This correspondence was developed further in [5,7], accounting for value dependent types and parametric polymorphism in the sense discussed in this proposal, in [51], developing concurrent evaluation strategies for \( \lambda \)-calculus as they arise from the mapping of the translation of linear to intuitionistic logic to the process setting, and also in [12], accounting for asynchronous communication and also a slightly different, more logically faithful term assignment. It is the synchronous version of this assignment that is developed in this proposal. The work of Wadler [55] also develops a correspondence of linear logic and session types, in the classical setting, which results in a language quite different from our own.

From the point of view of dependently typed languages with concurrency, perhaps the most significantly related is the work on F star, a distributed functional programming language [48]. F star does not directly employ session types, but they can be encoded in the language. In the non-dependently typed setting, Wadler’s GV [55] (based on a language by Gay and Vasconcelos [18]) is quite close to our integration of functional and concurrent programming. However, GV is itself linear since it does not separate the functional and the concurrent components of the language through a monad as we do.

The usage of logical relations applied to \( \pi \)-calculus processes has been studied in [3], but not in a session typed setting which results in a substantially different theory.

7.2 Thesis Outline

Here I sketch the basic structure of my thesis. While the specifics of its structure depends obviously on the results of the research I have yet to do, the general outline of the dissertation should be as follows:

**Intuitionistic Linear Logic as Session Types.** I will present the basic interpretation of intuitionistic propositional linear logic as a session typed language, along the lines of Section 3 of this proposal, detailing more precisely the subject reduction and progress properties and their proofs.
**Beyond Propositional Logic.** I will develop fully the theory of value-dependent and polymorphic session types, as they arise from first and second-order intuitionistic linear logic (based on Section 4 of this document), proving subject reduction and progress for these extensions.

**A Concurrent Session-Based Programming Language.** I will develop a concurrent programming language that combines functional and concurrent computation using a contextual monad, including full recursion for the sake of expressiveness which will be showcased by example. I prove type preservation and progress for the language, as well as formally re-establish the connections with logic by restricting general recursion, eliminating divergent computations.

**Reasoning using a Logical Foundation.** I fully develop the theory of linear logical relations as it applies to the propositional, polymorphic and coinductive session typed setting, proving termination of the respective languages. I also develop the relational version of the framework for the polymorphic and coinductive setting, obtaining a notion of process (or program) equivalence that can be shown to coincide with barbed congruence in the polymorphic setting. I will explore the properties of this equivalence in the coinductive setting and how it relates to the work on the behavioral theory of higher-order process calculi.

**Dependent Types.** By using the extensional equality that arises from logical equivalence in the coinductive and monadic setting as a soundness criteria, I will develop an intensional dependent session type theory. This requires developing a sound and decidable intensional equality of process expressions that is rich enough to enable reasoning about process behavior. Developing this equality, I will show how to use this type theory to prove interesting properties of processes, potentially using processes as proof witnesses for the properties themselves (in the case of coinductive proofs).

### 7.3 Goals and Plans

My main initial focus is the development of the relational version of the linear logical relations for the monadic / coinductive setting and exploring its connections to the behavioral theory of higher-order process calculi. Given this notion of logical equivalence, or equality, it should be possible to define a syntactic (decidable) equality on process expressions that is sound wrt logical equivalence that can then be used as a definitional equality in a full dependently typed version of our monadic language, where both functional and concurrent layers may be dependently typed. Provided I am able to develop such an equality, I need to explore the expressiveness of the type theory through examples, in order to determine the kinds of properties one can prove within the theory and how feasible it is to reason about concurrent computation using the type theory. The remaining topics I have discussed in the previous sections have already been developed during my PhD work.

I do not propose to produce an implementation of either the concurrent programming language or its hypothetical dependently typed version since that would be overly ambitious given the time constraints. I do note that such work is not to be seen as irrelevant, and is indeed already being researched by others.

While at this point I strongly believe that it should be possible to develop the intensional equality necessary to produce a full dependent session type theory, in the end it may turn out to be the case that I am unable to develop an equality that is both sound wrt our notion of extensional equality and at the same time expressive enough for our needs. Much of the tentative work proposed here relies crucially on the expressiveness of this hypothetical equality, and such an equality will be investigated further, even if beyond the time constraints of my thesis. However, I believe the remainder of the work discussed in this proposal is already a significant research effort in establishing linear logic as a de facto logical foundation for message-passing concurrency, even if in the end of my dissertation I am unable to produce a full dependent type theory based on these ideas.

### References


