# Corecursion in Session-Typed Processes

Bernardo Toninho*†, Luís Caires* and Frank Pfenning†
*CITI and FCT - Universidade Nova de Lisboa
†Carnegie Mellon University

***Abstract*—Session types are widely accepted as a useful expressive discipline for structuring communications in concurrent and distributed systems. In order to express infinitely unbounded sessions, as required to model the behaviour of realistic distributed systems, session typed languages often introduce general recursion operators at both the program and the type level. Unfortunately, general recursion, in particular when combined with name passing and mobility, may easily introduce undesirable divergence, e.g., infinite unobservable reduction sequences.**

**In this paper we address, by means of typing, the challenge of ensuring non-divergence in a session-typed $\pi$-calculus with general (co)recursion, while still allowing interesting infinite behaviours to be definable, as necessary to model realistic open process networks and distributed systems. Our approach builds on a Curry-Howard correspondence between our type system and a standard system of linear logic extended with co-inductive types, for which our non-divergence property implies consistency.**

**We prove type safety for our framework, implying protocol compliance and global progress of well-typed processes. Remarkably, we also establish - through a logical relations argument - that well-typed processes are compositionally non-divergent, in the sense that no well-typed composition of processes, including compositions dynamically assembled through name passing, can result in divergent behaviour, a property of practical relevance.**

## I. Introduction

This work addresses, by means of a logically motivated typing discipline, the problem of ensuring compositional non-divergence in session-based process systems with infinite behaviour defined using (co)recursive types.

We live in an age of concurrent and distributed software, meant to run not only as local applications but as cooperating parts of larger, distributed and mobile services, meant to run indefinitely with multiple independent clients, which are hard to build and ensure correct. Process models combined with techniques for precisely characterising and analysing system behavior have been exploited to verify properties such as deadlock freedom, protocol compliance, and availability of distributed and service based systems. Among type-based approaches to verification, a rather successful technique has been that of *session types* [1], [2], [3].

Session types structure message-based concurrency around the notion of a session, which is a precise description of the interaction patterns of two (or more) communicating agents with an intrinsic notion of protocol state (e.g. input a string and then output an integer) and duality. Thus, session types are a form of protocol descriptions that can be statically checked for compliance. The recent discovery of a correspondence between (all linear and shared features of) session types and pure Linear Logic in the style of Curry-Howard [4], linking proofs with typed processes and proof reduction as communication, has sparked a renewed interest in session types and their foundations [5], and in the idea of exploiting logically motivated approaches for providing powerful reasoning techniques about concurrent and distributed systems. This line of work has addressed concepts such as value-dependent types [6], proof-carrying code [7] behavioural polymorphism [8], and higher-order computation [9], approaching a general type theory for session-based concurrency, with strong guarantees by virtue of typing such as deadlock freedom and session fidelity.

Although in the untyped case infinite behavior is potentially encodable using replication in the $\pi$-calculus, to express infinitely unbounded session exchanges, as required to model the behaviour of realistic distributed systems, session typed languages often introduce general recursion operators at both the program and the type level. In a typed setting, replication can only capture *finite session behavior* that can be replicated (shared) arbitrarily often. It is not rich enough to model *infinite session behavior*, nor repeating behavioral patterns that depend on evolving state.

Unfortunately, existing session type systems for (channel-passing, higher-order) systems equipped with general recursion operators do not avoid validating systems exhibiting undesirable divergence, e.g., infinite sequences of unobservable internal reduction steps. While this issue already arises at the level of individual systems, it but becomes more serious when one needs to consider realistic dynamically linked distributed systems. For example, plugging together subsystems, e.g. as a result of dynamic channel passing or of linking downloaded (higher-order) code to local services, may undesirably 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, and this may be caused by simple programming error or by malicious intentions (e.g. a denial of service attack).

In this work, we tackle the challenge of reconciling general recursion, thus enabling potentially infinite behaviour to be expressed, with local termination and strong normalization within a session typed framework. As a toy example, consider a web service from Twitter offering a replicated service $trends$ which is intended to produce a stream of current trends, according to some custom metrics. To that end, the service is parametrized by a filter process that given a stream of tweets produces a stream of trends. The code implementing the service is:

$$\mathsf{Serv} \triangleq !trends(x).x(f).(\boldsymbol{\nu} y) f\langle y\rangle.(\mathsf{TweetSrc}_y \mid [f \leftrightarrow x])$$

When invoked, the Serv replicated service creates a new

session on fresh channel $x$, inputs on session $x$ the appropriate filter function $f$, links it to the internal tweet source $\mathsf{TweetSrc}_y$ and then repeatedly forward the results back on session $x$. The session types involved are:

$$\mathsf{TrendService} \triangleq !((\mathsf{Tweets} \multimap \mathsf{Trends}) \multimap \mathsf{Trends})$$
$$\mathsf{Tweets} \triangleq \nu X.(\mathsf{tweet} \wedge X) \quad \mathsf{Trends} \triangleq \nu Y.(\mathsf{trend} \wedge Y)$$

A possible client for the service Serv is

$$\mathsf{Joe} \triangleq (\boldsymbol{\nu} x)\mathit{trends}\langle x\rangle.(\boldsymbol{\nu} k)x\langle k\rangle.(\mathsf{F}_k|(\mathsf{corec}\ Z.x(y).p\langle y\rangle.Z))$$

This process invokes the shared server, resulting in a fresh session in channel $x$, sends (access to) a custom analytics package $\mathsf{F}_k$, and then sits in a loop printing out (on session $p$) each trend received from the server in session $x$. In our type system, we may derive the typing judgment

$$\mathit{trends}{:}\mathsf{TrendService} \vdash \mathsf{Joe} :: p{:}\mathsf{Trends}$$

Notice that we build on the intuitionistic formulation of linear logic based session types [4], where typing judgments have the form $\Gamma;\Delta \vdash P :: x{:}U$. This states of process $P$ that it provides a session of type $U$ at $x$, when composed with services / sessions as specified by $\Gamma;\Delta$. Using the cut rule, we may compose the server with a client in a system $\mathsf{Sys} \triangleq (\boldsymbol{\nu}\mathit{trends})(\mathsf{Serv} \mid \mathsf{Joe})$. System Sys will (unboundedly) print out trends on channel $p$, actually, system Sys is typed as $\vdash \mathsf{Sys} :: p{:}\mathsf{Trends}$. Being typed in our system, and although it generates an infinite stream of trends at $p$, involves higher-order name passing, dynamic linking, and occurrences of recursive calls, Sys will never get into internal divergence. It is challenging to obtain an expressive typing discipline ensuring the compositional non-divergence property. For instance, consider the very similar looking processes

$$\mathsf{Loop} \triangleq \quad \mathsf{corec}\ L(c).c(x).(\boldsymbol{\nu} d)(L(d) \mid d\langle n\rangle.[d \leftrightarrow c])$$
$$\mathsf{Good} \triangleq \quad \mathsf{corec}\ G(c).c\langle x\rangle.(\boldsymbol{\nu} d)(G(d) \mid c\langle n\rangle.[d \leftrightarrow c])$$

These two processes do not autonomously eventually diverge. It is possible to type Good, thus ensuring that it will never diverge, even when composed with arbitrary (well-typed) processes. However, this is not the case for process Loop, which gets into an infinite internal reduction sequence after the first communication on $c$, and (of course) is not typeable in our type system.

Our work has important practical and foundational consequences. From a foundational perspective, we proceed by developing a theory of coinductive session types, fully preserving the connections with linear logic in the sense of [4]. In this context, non-divergence implies termination of cut-elimination, and therefore we establish logical consistency of our framework. From a practical perspective, we provide a typing discipline for session types supporting rich infinite interactive behaviour, while rejecting systems that may engage into autistic internal divergence. More precisely, our typing discipline ensures, together with global progress (actually, lock-freedom) and protocol fidelity, the *compositional non-divergence* of infinite behaviors (i.e. that there is no well-typed process context under which a well-typed process will evolve to a divergent behavior), an important property out of the scope of existing session type systems with recursive types.

We summarise the contributions of our work:

- We introduce a session type system for our process calculus based on linear logic with coinductive types, associating corecursive process definitions with coinductive session types which encode potentially infinite session behavior.
- We show that well-typed processes enjoy very strong safety properties such as type preservation (or session fidelity) and progress, even in the presence of corecursion.
- We prove that well-typed processes are *compositionally non-divergent* by employing a logical relations argument, extended to coinductive session types, which combined with type safety ensures that any well-typed, distributed service implemented in our calculus will not only never "get stuck", but will also never become unavailable through divergent behavior.

## II. Process Model

In this section we introduce our process calculus, essentially consisting of a (synchronous) $\pi$-calculus with basic data types for convenience, input-guarded replication, labelled choice and selection and corecursion. The syntax of processes is given below:

$$\begin{array}{lcl} M, N & ::= & \ldots \quad \text{(basic data constructors)} \\ P, Q & ::= & x\langle M\rangle.P \mid x(y).P \mid x\langle y\rangle.P \mid (\boldsymbol{\nu} y)P \\ & \mid & !x(y).P \mid P \mid Q \mid x.\mathsf{case}(\overline{l_j \Rightarrow P_j}) \mid x.l_i; P \\ & \mid & (\mathsf{corec}\ X(\overline{y}).P)\ \overline{c} \mid X(\overline{c}) \mid [x \leftrightarrow y] \mid \mathbf{0} \end{array}$$

We range over basic data with $M, N$ and processes with $P, Q$. We write $\overline{y}$ and $\overline{c}$ for a list of variables and channels, respectively. We write $fn(P)$ for the free names of process $P$. Basic data type constructors include the typical constructs for manipulating data such as numbers, strings and lists. The process language is a synchronous $\pi$-calculus with term input $x(y).P$ and output $x\langle M\rangle.P$, channel output $c\langle y\rangle.P$, input-guarded replication $!x(y).P$, $n$-ary labelled choice $x.\mathsf{case}(\overline{l_j \Rightarrow P_j})$ and selection $x.l_i; P$, channel forwarding $[x \leftrightarrow y]$ and, crucially, a parametrized *corecursion* operator $(\mathsf{corec}\ X(\overline{y}).P)\ \overline{c}$, enabling corecursive process definitions (the variables $\overline{y}$ are bound in $P$). The parameters are used to instantiate channels (and values) in recursive calls accordingly.

The operational behavior of processes is given in terms of reduction and labelled transitions, both defined modulo structural congruence (written $\equiv$) which captures basic structural identities of processes (Fig. 1). Reduction $P \longrightarrow Q$ is defined by the rules in Fig. 2. Term communication is only done in value form, meaning that all terms are reduced to values before communication takes place (we omit the reduction rules for terms and the compatible closure rules for conciseness). We note the standard unfolding semantics for corecursive definitions. Channel communication is always fresh. In the epilogue, we write $\Rightarrow$ for the reflexive transitive closure of $\longrightarrow$ and $P \Downarrow$ iff $P$ is non-divergent (i.e. there is no infinite reduction sequence starting with $P$).

$x \notin fn(P) \Rightarrow P \,|\, (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \,|\, Q)$ $\quad P \equiv_\alpha Q \Rightarrow P \equiv Q$
$P \mid (Q \,|\, R) \equiv (P \,|\, Q) \mid R$ $\quad P \,|\, Q \equiv Q \,|\, P$
$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$ $\quad [y \leftrightarrow x] \equiv [x \leftrightarrow y]$
$P \,|\, \mathbf{0} \equiv P$ $\quad (\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}$

Fig. 1. Structural Congruence

$c\langle V\rangle.P \mid c(x).Q \longrightarrow P \mid Q\{V/x\}$
$c\langle y\rangle.P \mid c(x).Q \longrightarrow P \mid Q\{y/x\}$
$c\langle y\rangle.P \mid !c(x).Q \longrightarrow P \mid Q\{y/x\} \mid !c(x).Q$
$c.\mathsf{case}(\overline{l_i \Rightarrow P_i}) \mid c.l_i; Q \longrightarrow P_i \mid Q$
$(\nu x)(P_y \mid [y \leftrightarrow x]) \longrightarrow P\{y/x\} \;\; (x \neq y)$
$(\mathsf{corec}\ X(\overline{y}).P)\ \overline{c} \longrightarrow P\{\overline{c}/\overline{y}\}\{(\mathsf{corec}\ X(\overline{y}).P)/X\}$
If $Q \longrightarrow Q'$ then $P \mid Q \longrightarrow P \mid Q'$
If $P \longrightarrow Q$ then $(\boldsymbol{\nu} y)P \longrightarrow (\boldsymbol{\nu} y)Q$
If $P \equiv P'$ and $P' \longrightarrow Q'$ and $Q' \equiv Q$ then $P \longrightarrow Q$

Fig. 2. Reduction

The labelled transition system is defined by the rules of Fig. 3, consisting of a typical early transition system with internal actions $\tau$, output $\overline{x\langle y\rangle}$ and its dual input $x(z)$ action, bound output $\overline{(\boldsymbol{\nu} y)x\langle y\rangle}$, label selection $\overline{x.l}$ and choice $x.l$.

## III. Type System

In this section we motivate and present our type system based on intuitionistic linear logic. The syntax of types is given in Fig. 4. We distinguish the types of basic data $\tau, \sigma$ from session types $A, B, C$.

The language of session types covers the standard session constructs: data input and output ($\tau \supset A$ and $\tau \wedge A$), (fresh) session input and output ($A \multimap B$ and $A \otimes B$), termination ($\mathbf{1}$), labelled choice and selection ($\&\{\overline{l_j{:}A_j}\}$ and $\oplus\{\overline{l_j{:}A_j}\}$), replication ($!A$) and coinductive session behavior ($\nu X.A$).

### A. From General Recursion to Corecursion

While both recursive and coinductive session types denote (potentially) infinite session behavior, the fundamental distinction is that using general recursion a process might generate an infinite sequence of *internal* actions, whereas a valid coinductive definition of a session typed process is guaranteed to always have a finite sequence of internal actions before offering some observable behavior. It is this *external*, observable behavior that may be infinite in a coinductively defined, session typed process. This is of paramount importance since it ensures that a well-typed process offering a coinductive session type will always be able to truly offer its specified service in a divergence-free way, even if the type specifies an infinite sequence of observable actions. Moreover, this non-divergence result is *compositional*: well-typed coinductive sessions may be safely composed, ensuring that the resulting system is itself non-divergent.

To rule out divergence we must impose a discipline on the occurrence of the recursion variable in processes, in line with the work on coinduction in implementations of dependent type theories such as Coq [10] or Agda [11], but here mapped to a concurrent process setting. Essentially, we must ensure that a corecursive process definition is *productive* – there is always a finite sequence of internal actions between *observable* actions.

In our setting, observable actions are those that take place on the session channel that is being *offered* by a given (well-typed) process, whereas 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 offered channel before allowing recursive calls to take place (i.e. recursive calls must be guarded by an observable action). For instance, the process $\mathsf{corec}\, P.c\langle 0\rangle.P$, is guarded if we assume that $c$ is the session channel that is being offered. However, as known from type theory, guardedness alone is not sufficient to ensure productivity. Process Loop from Section I is guarded but produces divergent behavior when composed with a process that provides it with an output. The problem with the definition of Loop is that after the input along $c$, we have an occurrence of the recursion variable in parallel with a process that *interacts* with the recursive occurrence locally, essentially destroying the productivity of the original definition.

Thus, ensuring non-divergence in a compositional way requires not only ensuring guardedness but also disallowing interactions with corecursive calls within corecursive definitions, a property we call *co-regular recursion*. In functional type theories, such restrictions typically consist of disallowing pattern matching over the result of recursive calls, or passing such results as arguments to other functions. In our process setting, interactions are generated by communication and so we must impose that processes that are placed in parallel with the corecursive call may not communicate with it, although they may themselves perform other actions. As we discuss at the end of this section, our type system ensures this form of non-interference by not exposing the communication interface of corecursive calls internally, which ensures that processes composed with corecursive calls may perform communication, but *not* with the corecursive call itself.

### B. Typing Coinductive Sessions

Having made the informal case for the kinds of restrictions on general recursion that are needed in order to eliminate divergent computation, we now present the type system for our language, essentially made up of the rules of linear logic plus the rules that pertain to corecursive process definitions, coinductive session types and the corecursion variable.

We define the typing judgment: $\Psi; \Gamma; \Delta \vdash_\eta P :: z{:}A$ denoting that process $P$ offers the session behavior typed with $A$ along channel $z$, when composed with the (linear) session behaviors specified in $\Delta$, with the (unrestricted, or shared) session behaviors specified in $\Gamma$ and where $\eta$ is a mapping from (corecursive) type variables to typing contexts (we detail this further below). We note that the names in $\Gamma$, $\Delta$ and $z$ are all pairwise distinct. We assume typing to be defined modulo structural congruence by definition. The context $\Psi$ tracks the free variables in $P$ that pertain to basic data values that are to be sent and received. We make use of a second judgment

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}\ par \qquad \frac{P \xrightarrow{\overline{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ com \qquad \frac{P \xrightarrow{\alpha} Q}{(\boldsymbol{\nu} y)P \xrightarrow{\alpha} (\boldsymbol{\nu} y)Q}\ res \qquad \frac{P \xrightarrow{\overline{x\langle y\rangle}} Q}{(\boldsymbol{\nu} y)P \xrightarrow{\overline{(\boldsymbol{\nu} y)x\langle y\rangle}} Q}\ open$$

$$\frac{P \xrightarrow{\overline{(\boldsymbol{\nu} y)x\langle y\rangle}} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} y)(P' \mid Q')}\ close \qquad !u(x).P \xrightarrow{u(z)} P\{z/x\} \mid !u(x).P \qquad x\langle y\rangle.P \xrightarrow{\overline{x\langle y\rangle}} P \qquad x(y).P \xrightarrow{x(z)} P\{z/y\}$$

$$(\nu x)([x \leftrightarrow y] \mid P) \xrightarrow{\tau} P\{y/x\} \qquad x.l; P \xrightarrow{\overline{x.l}} P \qquad x.\mathsf{case}(\overline{l_i \Rightarrow P_i}) \xrightarrow{x.l_i} P_i \qquad (\mathsf{corec}\ X(\overline{y}).P)\ \overline{c} \xrightarrow{\tau} P\{\overline{c}/\overline{y}\}\{(\mathsf{corec}\ X(\overline{y}).P)/X\}$$

Fig. 3. Labelled Transition System

| | | | |
|---|---|---|---|
| $\tau, \sigma$ | ::= | $\mathtt{nat} \mid \mathtt{string} \mid \ldots$ | (basic data types) |
| $A, B, C$ | ::= | $\tau \supset A$ | input value of type $\tau$ and continue as $A$ |
| | | $\tau \wedge A$ | output value of type $\tau$ and continue as $A$ |
| | | $A \multimap B$ | input channel of type $A$ and continue as $B$ |
| | | $A \otimes B$ | output fresh channel of type $A$ and continue as $B$ |
| | | $\mathbf{1}$ | terminate |
| | | $\&\{\overline{l_j{:}A_j}\}$ | offer choice between $l_j$ and continue as $A_j$ |
| | | $\oplus\{\overline{l_j{:}A_j}\}$ | provide one of the $l_j$ and continue as $A_j$ |
| | | $!A$ | provide replicable service $A$ |
| | | $\nu X.A \mid X$ | coinductive session type |

Fig. 4. The Syntax of Types

$\Psi \vdash M{:}\tau$ to denote that term $M$, denoting a value that is to be communicated, is well typed under the assumptions in $\Psi$.

The rules that define our type system are given in Fig. 5, consisting essentially of those of [4] with the identity rule, value input and output (present in [9]) and coinductive types, which are associated with corecursion in the process calculus. We note that coinductive types have strictly positive occurrences of the type variable, also excluding coinductive types that have no associated session behavior before the type variable occurrence (such as $\nu X.X$). Moreover, we require coinductive types to mention the type variable.

We refrain from a detailed presentation of every rule for the sake of conciseness, highlighting instead the rules pertaining to corecursive processes and coinductive session types. We begin with the right rule for coinductive sessions, which types (parameterized) corecursive process definitions:

$$\frac{\Psi; \Gamma; \Delta \vdash_{\eta'} P :: c{:}A \quad \eta' = \eta[X(\overline{y}) \mapsto \Psi; \Gamma; \Delta \vdash c{:}Y]}{\Psi; \Gamma; \Delta \vdash_{\eta} (\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z} :: c{:}\nu Y.A}\ (\nu\mathsf{R})$$

In the rule above, the process $P$ may use the recursion variable $X$ and refer to the parameter list $\overline{y}$, which is instantiated with the list of (distinct) names $\overline{z}$ which may occur in $\Psi$, $\Delta$, $\Gamma$ or $c$. Moreover, we keep track of the contexts $\Psi$, $\Gamma$ and $\Delta$ in which the corecursive definition is made, as well as the channel name along which the coinductive behavior is offered and the type variable associated with the corecursive behavior, by extending the mapping $\eta$ with a binding for $X$ with the appropriate information. This is necessary because, intuitively, each occurrence of the corecursion variable stands for $P$ itself (modulo the parameter instantiations) and therefore we must check that the necessary ambient session behaviors are available for $P$ to execute in a type correct way, respecting linearity. $P$ itself simply offers along channel $c$ the session behavior $A$ (which is an open type). To type the corecursion variable we use the following rule:

$$\frac{\eta(X(\overline{y})) = \Psi; \Gamma; \Delta \vdash d{:}Y \quad \rho = \{\overline{z}/\overline{y}\}}{\rho(\Psi); \rho(\Gamma); \rho(\Delta) \vdash_{\eta} X(\overline{z}) :: \rho(d){:}Y}\ (\text{VAR})$$

We type a process corecursion variable $X$ by looking up in $\eta$ the binding for $X$, which references the typing environments $\Psi$, $\Gamma$ and $\Delta$ under which the corecursive definition is well defined, the coinductive type variable $Y$ associated with the corecursive behavior and the channel name $d$ along which the behavior is offered. The corecursion variable $X$ is typed with the type variable $Y$ if the parameter instantiation is able to satisfy the typing signature (by renaming available linear and exponential resources or term variables). We also allow for the offered session channel to be a parameter of the corecursion. Finally, the left rule for coinductive session types simply unfolds the type accordingly:

$$\frac{\Psi; \Gamma; \Delta, c{:}A\{\nu X.A/X\} \vdash_{\eta} Q :: d{:}D}{\Psi; \Gamma; \Delta, c{:}\nu X.A \vdash_{\eta} Q :: d{:}D}\ (\nu\mathsf{L})$$

While the rules look fairly straightforward, they turn out to introduce quite subtle restrictions on what constitutes a well-formed (i.e. well-typed) corecursive process definition. First, observe that the introduction form for corecursive definitions does not directly unfold the coinductive type in its premise and thus references an *open* type, which means that the (corecursive) definition of $P$ provides the behavior specified in $A$ up to occurrences of the coinductive type variable $Y$. On the other hand, using a coinductive session type (which is

$$(\wedge\mathsf{R})\ \frac{\Psi \vdash M{:}\tau \quad \Psi;\Gamma;\Delta \vdash_\eta P :: c{:}A}{\Psi;\Gamma;\Delta \vdash_\eta c\langle M\rangle.P :: c{:}\tau \wedge A} \qquad (\wedge\mathsf{L})\ \frac{\Psi, x{:}\tau\ ;\Gamma;\Delta, c{:}A \vdash_\eta Q :: d{:}D}{\Psi\ ;\Gamma;\Delta, c{:}\tau \wedge A \vdash_\eta c(x).Q :: d{:}D} \qquad (\supset\mathsf{R})\ \frac{\Psi, x{:}\tau\ ;\Gamma;\Delta \vdash_\eta P :: c{:}A}{\Psi;\Gamma;\Delta \vdash_\eta c(x).P :: c{:}\tau \supset A}$$

$$(\supset\mathsf{L})\ \frac{\Psi \vdash M{:}\tau \quad \Psi;\Gamma;\Delta, c{:}A \vdash_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\tau \supset A \vdash_\eta c\langle M\rangle.Q :: d{:}D} \qquad (\text{ID})\ \frac{}{\Psi;\Gamma;d{:}A \vdash_\eta [d \leftrightarrow c] :: c{:}A} \qquad (\mathbf{1}\mathsf{R})\ \frac{}{\Psi;\Gamma;\cdot \vdash_\eta \mathbf{0} :: c{:}\mathbf{1}} \qquad (\mathbf{1}\mathsf{L})\ \frac{\Psi;\Gamma;\Delta \vdash_\eta P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\mathbf{1} \vdash_\eta P :: d{:}D}$$

$$(!\mathsf{R})\ \frac{\Psi;\Gamma;\cdot \vdash_\eta P :: x{:}A}{\Psi;\Gamma;\cdot \vdash_\eta !c(x).P :: c{:}!A} \qquad (!\mathsf{L})\ \frac{\Psi;\Gamma, u{:}A;\Delta \vdash_\eta P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}!A \vdash_\eta P\{c/u\} :: d{:}D} \qquad (\text{COPY})\ \frac{\Psi;\Gamma, u{:}A;\Delta, x{:}A \vdash_\eta P :: d{:}D}{\Psi;\Gamma, u{:}A;\Delta \vdash_\eta (\boldsymbol{\nu} x)u\langle x\rangle.P :: d{:}D}$$

$$(\otimes\mathsf{R})\ \frac{\Psi;\Gamma;\Delta_1 \vdash_\eta P_1 :: x{:}A \quad \Psi;\Gamma;\Delta \vdash_\eta P_2 :: c{:}B}{\Psi;\Gamma;\Delta_1, \Delta_2 \vdash_\eta (\boldsymbol{\nu} x)c\langle x\rangle.(P_1 \mid P_2) :: c{:}A \otimes B} \qquad (\otimes\mathsf{L})\ \frac{\Psi;\Gamma;\Delta, x{:}A, c{:}B \vdash_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}A \otimes B \vdash_\eta c(x).Q :: d{:}D}$$

$$(\multimap\mathsf{R})\ \frac{\Psi;\Gamma;\Delta, x{:}A \vdash_\eta P :: c{:}B}{\Psi;\Gamma;\Delta \vdash_\eta c(x).P :: c{:}A \multimap B} \qquad (\multimap\mathsf{L})\ \frac{\Psi;\Gamma;\Delta_1 \vdash_\eta Q_1 :: x{:}A \quad \Psi;\Gamma;\Delta_2, c{:}B \vdash_\eta Q_2 :: d{:}D}{\Psi;\Gamma;\Delta_1, \Delta_2, c{:}A \multimap B \vdash_\eta (\boldsymbol{\nu} x)c\langle x\rangle.(Q_1 \mid Q_2) :: d{:}D}$$

$$(\&\mathsf{R})\ \frac{\Psi;\Gamma;\Delta \vdash_\eta P_1 :: c{:}A_1 \ \ldots\ \Psi;\Gamma;\Delta \vdash_\eta P_n :: c{:}A_n}{\Psi;\Gamma;\Delta \vdash_\eta c.\mathsf{case}(\overline{l_j \Rightarrow P_j}) :: c{:}\,\&\,\{\overline{l_j{:}A_j}\}} \qquad (\&\mathsf{L})\ \frac{\Psi;\Gamma;\Delta, c{:}A_i \vdash_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\,\&\,\{\overline{l_j{:}A_j}\} \vdash_\eta c.l_i; Q :: d{:}D}$$

$$(\oplus\mathsf{R})\ \frac{\Psi;\Gamma;\Delta \vdash_\eta P :: c{:}A_i}{\Psi;\Gamma;\Delta \vdash_\eta c.l_i; P :: c{:}\,\oplus\,\{\overline{l_j{:}A_j}\}} \qquad (\oplus\mathsf{L})\ \frac{\Psi;\Gamma;\Delta, c{:}A_1 \vdash_\eta Q_1 :: d{:}D \ \ldots\ \Psi;\Gamma;\Delta, c{:}A_n \vdash_\eta Q_n :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\,\oplus\,\{\overline{l_j{:}A_j}\} \vdash_\eta c.\mathsf{case}(\overline{l_j \Rightarrow Q_j}) :: d{:}D}$$

$$(\nu\mathsf{L})\ \frac{\Psi;\Gamma;\Delta, c{:}A\{\nu X.A/X\} \vdash_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\nu X.A \vdash_\eta Q :: d{:}D} \qquad (\nu\mathsf{R})\ \frac{\Psi;\Gamma;\Delta \vdash_{\eta'} P :: c{:}A \quad \eta' = \eta[X(\overline{y}) \mapsto \Psi;\Gamma;\Delta \vdash c{:}Y]}{\Psi;\Gamma;\Delta \vdash_\eta (\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z} :: c{:}\nu Y.A} \qquad (\text{VAR})\ \frac{\eta(X(\overline{y})) = \Psi;\Gamma;\Delta \vdash d{:}Y \quad \rho = \{\overline{z}/\overline{y}\}}{\rho(\Psi);\rho(\Gamma);\rho(\Delta) \vdash_\eta X(\overline{z}) :: \rho(d){:}Y}$$

$$(\text{CUT})\ \frac{\Psi;\Gamma;\Delta_1 \vdash_\eta P :: x{:}A \quad \Psi;\Gamma;\Delta_2, x{:}A \vdash_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta_1, \Delta_2 \vdash_\eta (\boldsymbol{\nu} x)(P \mid Q) :: d{:}D} \qquad (\text{CUT}^!)\ \frac{\Psi;\Gamma;\cdot \vdash_\eta P :: x{:}A \quad \Psi;\Gamma, u{:}A;\Delta \vdash Q :: d{:}D}{\Psi;\Gamma;\Delta \vdash_\eta (\boldsymbol{\nu} u)(!u(x).P \mid Q) :: d{:}D}$$

Fig. 5. Typing Rules

achieved by the left rule $\nu\mathsf{L}$) entails unfolding the coinductive type as expected, and so a user of a coinductive behavior may use as many unfoldings of $\nu Y.A$ as required.

Up to this point, we have yet to discuss how our type system enforces the necessary restrictions to ensure non-divergence. It turns out that these restrictions are imposed by the variable rule in quite subtle ways, due to its interaction with the $\nu\mathsf{R}$ rule. First, observe that we enable multiple occurrences of the recursion variable in a well-typed process, for instance in different branches of a case construct, through cuts or the $\otimes\mathsf{R}$ rule (e.g. Fig 6). Moreover, we do not syntactically exclude processes without terminal recursion (for instance, process Good from Section I).

However, a well-typed corecursive definition *cannot* interact with its corecursive calls (which could potentially destroy productivity). To see why this is the case, consider how such an interaction might be allowed in our type system: in order for a corecursive definition to interact with its own corecursive call in a well-typed manner, the system would require some form of parallel composition where we obtain a handle to the corecursive call. This would only be possible through a cut, given that the right rule for $\otimes$ ensures that the two parallel processes cannot interact. However, since the coinductive type is never unfolded when *offering* a coinductive definition, the session interface of the corecursive occurrence is not visible internally. Thus, a cut of the corecursion variable with some other process $Q$ will generate a fresh channel $c$ that offers the coinductive type variable, but not the *coinductive type*, meaning that no left rules that interact with the unfolding of the recursive definition can be applied. In fact, the only rule that can use $c{:}X$ is the identity rule, which will forward the corecursively defined session. This does not prohibit $Q$ from having additional behavior besides forwarding, it simply excludes (potentially) problematic internal interactions with corecursive calls.

## IV. Examples

In this section we discuss three examples: the Loop process and the hypothetical Twitter web service, combining replication and recursion, from Section I, and a coordinated process network implementing a binary numeral.

### A. Excluding Unobservable Divergence

Elaborating on the process Loop given in Section I, we show how it can result in divergent behavior when interacting with a

$$\dfrac{\dfrac{\dfrac{}{\vdash X(y) :: y{:}Y}\,(var) \quad \dfrac{}{\vdash X(z) :: z{:}Y}\,(var)}{\vdash (\boldsymbol{\nu} y)z\langle y\rangle.(X(y) \mid X(z)) :: z{:}Y \otimes Y}\,(\otimes\mathsf{R}) \quad \vdash 1 : \mathsf{nat}}{\vdash z\langle 1\rangle.(\boldsymbol{\nu} y)z\langle y\rangle.(X(y) \mid X(z)) :: z{:}\mathsf{nat} \wedge (Y \otimes Y)}\,(\wedge\mathsf{R})}{\vdash (\mathsf{corec}\ X(z).z\langle 1\rangle.(\boldsymbol{\nu} y)z\langle y\rangle.(X(y) \mid X(z)))\ z :: z{:}\nu Y.\mathsf{nat} \wedge (Y \otimes Y)}\,(\nu\mathsf{R})$$

Fig. 6. Example Typing Derivation

client. We also show why Loop is not typeable in our system. Recall the definition of Loop:

$$\mathsf{Loop} \triangleq \quad \mathsf{corec}\ L(c).c(x).(\boldsymbol{\nu} d)(L(d) \mid d\langle n\rangle.[d \leftrightarrow c])$$

The intended behavior of Loop is to continuously input along the channel $c$, which is represented by the session type $\nu X.\mathsf{nat} \supset X$. Consider a process $P$ that provides an output along $c$, composed with Loop. We have the following reduction(s):

$$(\boldsymbol{\nu} c)(\mathsf{Loop} \mid P) \Rightarrow (\boldsymbol{\nu} c)(\boldsymbol{\nu} d)(\mathsf{Loop}\{c/d\} \mid d\langle n\rangle.[d \leftrightarrow c] \mid P')$$

In the process to the right of the arrow, there is an internal synchronization between the output along $d$ and the input along the unfolding of Loop, resulting in the following process:

$$(\boldsymbol{\nu} c)(\boldsymbol{\nu} d)((\boldsymbol{\nu} d')(\mathsf{Loop}\{d'/c\} \mid d'\langle n\rangle.[d' \leftrightarrow d]) \mid [d \leftrightarrow c] \mid P')$$

The infinite internal reduction is now made clear, where regardless of the behavior of $P'$ we always have available an internal reduction that produces an additional unfolding of Loop and may repeat this process an infinite number of times.

It is easy to see that Loop is not well-typed in our system: if we try to type the process bottom-up, we first apply the $\nu$R rule, followed by the $\supset$R rule. We are then left with trying to type the following cut:

$$\dfrac{\vdash L(d) :: d{:}X \quad d{:}X \vdash d\langle n\rangle.[d \leftrightarrow c] :: c{:}X}{\vdash (\boldsymbol{\nu} d)(L(d) \mid d\langle n\rangle.[d \leftrightarrow c]) :: c{:}X}$$

It is then obvious that the right premise of the cut cannot be typed, since all that is known about session $d$ is that it has type $X$, so no communication along $d$ can be well-typed. A type-correct version of the behavior $c{:}\nu X.\mathsf{nat} \supset X$ is:

$$\mathsf{Ins} \triangleq \mathsf{corec}\ L(c).c(x).L(c)$$

Note however that a process that is meant to interact with Ins by just outputting continuously along $c$ is *not* well-typed in our system. To use the behavior of Ins in this way, we must necessarily define a corecursive process, and thus a process that offers a coinductive session type. However, the continuous user of Ins that just outputs along $c$ offers no observable behavior and thus no valid coinductive session type can be assigned to it (recall that valid coinductive types need to be strictly positive and non-trivial). We exclude such processes from our system since the composition of such a continuous user process with Ins would result in a process with no observable actions, but with infinite internal actions:

$$(\boldsymbol{\nu} c)(\mathsf{Ins} \mid \mathsf{corec}\ S.c\langle 0\rangle.S) \Rightarrow (\boldsymbol{\nu} c)(\mathsf{Ins} \mid \mathsf{corec}\ S.c\langle 0\rangle.S) \Rightarrow \ldots$$

Morally, such a user process must offer some observable behavior to be useful (even a divergent process should be receptive to a "kill" signal). For instance, the process $P \equiv \mathsf{corec}\ S'.c\langle 0\rangle.d\langle 0\rangle.S'$ is a well-typed process offering behavior $d{:}\nu X.\mathsf{nat} \wedge X$ in a context where Ins is available, and so the composition of Ins and $P$ is well-typed:

$$\vdash (\boldsymbol{\nu} c)(\mathsf{Ins} \mid P) :: d{:}\nu X.\mathsf{nat} \wedge X$$

### *B. Replication and Corecursion*

A web service such as Twitter offers a persistent (and thus replicated) service *trends* which is intended to produce a stream of current trends, according to some custom metrics. To do this, the service is parametrized by a filter process (of type Tweets $\multimap$ Trends) that given a stream of tweets produces a stream of trends. Such a protocol is captured via the following type(s):

$$\begin{array}{lcl} \mathsf{TrendService} & \triangleq & !((\mathsf{Tweets} \multimap \mathsf{Trends}) \multimap \mathsf{Trends}) \\ \mathsf{Tweets} & \triangleq & \nu X.(\mathsf{tweet} \wedge X) \\ \mathsf{Trends} & \triangleq & \nu Y.(\mathsf{trend} \wedge Y) \end{array}$$

The use of the exponential ! in TrendService ensures that each invocation of the service provides access to a fresh, independent session. Any such session expects to receive a session of type Tweets $\multimap$ Trends able to transduce a stream of tweets to a stream of trends. This is the usual way to represent higher-order communication (e.g., sending an "object" of type Tweets $\multimap$ Trends) in a communication) in the $\pi$-calculus. The service will use this session to finally provide the stream of trends, after connecting it to a (private) source of tweets $\mathsf{TweetSrc}_y$. The code implementing the service is:

$$\mathsf{Serv} \triangleq !trends(x).x(f).(\boldsymbol{\nu} y)f\langle y\rangle.(\mathsf{TweetSrc}_y \mid [f \leftrightarrow x])$$

Given that $\vdash \mathsf{TweetSrc}_y :: y{:}\mathsf{Tweets}$, we can derive the typing $\vdash \mathsf{Serv} :: trends{:}\mathsf{TrendService}$.

Possible client code for such a service is:

$$\mathsf{Cl} \quad \triangleq \quad (\boldsymbol{\nu} x)trends\langle x\rangle.(\boldsymbol{\nu} k)x\langle k\rangle.(\mathsf{Analytics}_k \mid [x \leftrightarrow tr])$$

This process interacts by first invoking the (replicated) trends service, and afterwards interacting with the freshly generated replica (at $x$). It first provides access to the appropriate analytics process, and afterwards forwards the resulting trend stream (from $k$) on channel $tr$. Assuming $\vdash \mathsf{Analytics}_k :: k{:}\mathsf{Tweets} \multimap \mathsf{Trends}$ we derive

$$trends : \mathsf{TrendService} \vdash \mathsf{Cl} :: tr{:}\mathsf{Trends}$$

### C. Little Endian Bit Counter

We illustrate how our framework can express fairly general process networks with nodes interacting according to structured session protocols. We consider the implementation of a binary counter, where each bit of a (arbitrary length) binary numeral is implemented by a process node which can only communicate with its neighbouring processes in the network (in [9] we discussed a similar example, but expressed using non co-regular recursion, and not typeable in our system. We present here a version using co-regular recursion, and requiring more sophisticated handshaking). Overall, the network implements a protocol offering three operations: poll the counter for its current integer value; increment the counter value, or terminate the counter. The corresponding session type is expressed coinductively, as we do not want to limit a priori the counter size:

$$\mathsf{Counter} \triangleq \nu X.\ \&\{\mathsf{val}{:}\mathsf{int} \wedge X, \mathsf{inc}{:}X, \mathsf{halt}{:}1\}$$

Our implementation of Counter is based on a coordinator process that keeps a (linear) network of communicating processes, representing the counter value in little endian form (the tail process in the network holds the least significant bit). Each network node communicates with its two adjacent bit representations, whereas the coordinator communicates with the most significant bit process (and with the counter's external client), spawning new bits as needed. Overall, the coordinator works as follows. To halt the counter, the coordinator halts every node before halting itself. To provide the value of the counter, the coordinator propagates a val message along the network, which will compute the value as an integer. To increment the counter, the coordinator injects an inc message into the network, which will be propagated to the least significant bit process, and, in a second phase, propagated back as a carry message travelling back in the opposite direction, incrementing each bit (modulo 2) as needed. If the carry message, instead of a done message, reaches the coordinator, a new bit process will be dynamically spawned.

The behavior of each node is as follows. When a node receives a val message it receives the integer value computed by all the nodes encoding more significant bits, updates it with its own contribution and propagates the val message to the less significant bit nodes. When it gets a inc message it forwards it to its less significant bit neighbour, and waits for it to send back either a carry or done message. In the latter case, it will just forward the done message network along the most significant bit node up to the coordinator, signalling that nothing more needs to be done. In the former case, it will flip its value and either send a carry or a done message to its most significant bit neighbour. When a carry message reaches the coordinator, it generates a new bit, as mentioned above.

So, the type for each node $\mathsf{Node}(b,x,n)$ will be given by the judgment $x : \mathsf{CImpl} \vdash \mathsf{Node}(b,x,n) :: n{:}\mathsf{CImpl}$ where

$$\begin{array}{rcl}\mathsf{CImpl} & \triangleq & \nu X.\ \&\{\mathsf{val}{:}\mathsf{int} \supset \mathsf{int} \wedge X,\\ & & \quad \mathsf{inc}{:} \oplus\{\mathsf{carry}{:}X, \mathsf{done}{:}X\}, \mathsf{halt}{:}\mathbf{1}\}\end{array}$$

In $\mathsf{Node}(b,x,n)$, $b$ holds the bit value, $x$ is the session channel connecting to the less significant node (or to the terminal node) and $n$ is the channel connecting to the most significant node (or to the coordinator). Code for $\mathsf{Node}(b,x,n)$ is as follows:

$$\begin{array}{rcl}\mathsf{Node}(b,x,n) & \triangleq & \mathsf{corec}\ X(b,x,n).n.\mathsf{case}(\\ & & \quad \mathsf{val} \Rightarrow x.\mathsf{val}; n(m).x\langle(2*m+b)\rangle.\\ & & \quad\quad X(b,x,n),\\ & & \quad \mathsf{inc} \Rightarrow x.\mathsf{inc};\\ & & \quad x.\mathsf{case}(\mathsf{carry} \Rightarrow\\ & & \quad \mathsf{if}\ (b=1)\ \mathsf{then}\ n.\mathsf{carry}; X(0,x,n)\\ & & \quad\quad \mathsf{else}\ n.\mathsf{done}; X(1,x,n),\\ & & \quad \mathsf{done} \Rightarrow X(1,x,n),\\ & & \quad \mathsf{halt} \Rightarrow x.\mathsf{halt}; \mathbf{0})\ (b,x,n)\end{array}$$

The coordinator process code interfaces with clients and wraps the bit process network, generating new bit nodes as needed.

$$\begin{array}{rcl}\mathsf{Coord}(x,z) & \triangleq & \mathsf{corec}\ X(x,z).\\ & & z.\mathsf{case}(\mathsf{val} \Rightarrow x.\mathsf{val}; x\langle 0\rangle.\\ & & \quad x(n).z\langle n\rangle.X(x,z),\\ & & \mathsf{inc} \Rightarrow x.\mathsf{inc};\\ & & \quad x.\mathsf{case}(\mathsf{carry} \Rightarrow (\boldsymbol{\nu} n')(\mathsf{Node}(1,x,n') \mid\\ & & \quad\quad \mathsf{Coord}(n',z)),\\ & & \quad\quad \mathsf{done} \Rightarrow X(x,z))\\ & & \mathsf{halt} \Rightarrow x.\mathsf{halt}; \mathbf{0})\ (x,z)\end{array}$$

To complete the system we provide the implementation of the empty bit string epsilon, which will be a closed process of type CImpl, expressing the "base case" of the protocol. For reading the value, it ping pongs the received value. For incrementing, it emits the (first) carry message. The Counter system is then produced by composing epsilon and Coord.

$$\begin{array}{rcl}\mathsf{epsilon}(x) & \triangleq & \mathsf{corec}\ X(x).x.\mathsf{case}(\mathsf{val} \Rightarrow x(n).x\langle n\rangle.X(x),\\ & & \quad\quad \mathsf{inc} \Rightarrow x.\mathsf{carry}; X(x),\\ & & \quad\quad \mathsf{halt} \Rightarrow \mathbf{0})\ x\\ \mathsf{Counter}(c) & \triangleq & (\boldsymbol{\nu} e)(\mathsf{epsilon}(e) \mid \mathsf{Coord}(e,c))\end{array}$$

The several relevant typings are

$$\begin{array}{l} e : \mathsf{CImpl} \vdash \mathsf{Coord}(e,c) :: c{:}\mathsf{Counter}\\ \vdash \mathsf{epsilon}(e) :: e{:}\mathsf{CImpl}\\ \vdash \mathsf{Counter}(c) :: c{:}\mathsf{Counter}\end{array}$$

The two examples above showcase the interaction between corecursive session types, which denote potentially infinite interactions (such as streams) on one session channel; and replication, which denotes the potential for an arbitrary number of copies of a given session, but on different channels.

## V. Main Results

In this section, we establish type safety for our calculus, entailing session fidelity and deadlock freedom. We also develop our main result of compositional non-divergence by extending the linear logical relations of [8], [12] to the coinductive setting, fully restoring the connection of our framework with the logical interpretation.

### A. Type Safety

Following [4], our proof of type preservation relies on a simulation between reductions in the session-typed $\pi$-calculus and proof reductions from logic.

*Theorem 5.1 (Type Preservation):* If $\Psi;\Gamma;\Delta \vdash_\eta P :: z{:}A$ and $P \to Q$ then $\Psi;\Gamma;\Delta \vdash_\eta Q :: z{:}A$.

The proof of progress also follows the lines of [4], but with some additional caveats due to the presence of corecursive definitions. The key technical aspects of the proof are a series of inversion lemmas and a notion of a *live* process, consisting of a process that has not yet fully carried out its ascribed session behavior, and thus is a parallel composition of processes where at least one is a non-replicated process, guarded by some action. We define $live(P)$ if and only if $P \equiv (\boldsymbol{\nu}\tilde{n})(\pi.Q \mid R)$, for some process $R$, sequence of names $\tilde{n}$ and a non-replicated guarded process $\pi.Q$.

*Theorem 5.2 (Progress):* If $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ and $live(P)$ then there is $Q$ with $P \to Q$ or one of the following holds:

(a) $\Delta = \Delta', y{:}B$, for some $\Delta'$ and $y{:}B$. There exists $\Gamma;\Delta'' \vdash R :: y{:}B$ s.t. $(\boldsymbol{\nu}y)(R \mid P) \to Q$.
(b) Exists $\Gamma; z{:}A, \Delta' \vdash R :: w{:}C$ s.t. $(\boldsymbol{\nu}z)(P \mid R) \to Q$.
(c) $\Gamma = \Gamma', u{:}B$, for some $\Gamma'$ and $u{:}B$. There exists $\Gamma;\cdot \vdash R :: x{:}B$ s.t. $(\boldsymbol{\nu}u)(!u(x).R \mid P) \to Q$.

Our notion of progress states that well-typed processes never get stuck even in the presence of infinite session behavior. Either the process progresses outright via internal computation, awaits on an interaction with its environment ($(a)$ or $(c)$) or is waiting for an interaction along its offered channel ($(b)$).

### B. Compositional Non-Divergence

To prove the key property of compositional non-divergence, we develop a (linear) logical relations argument for coinductive session types. As in prior work for languages without recursion [12], [8], we build on Girard's technique of reducibility candidates: a set of non-divergent, well-typed terms, closed under reduction and expansion (backward reduction), adapted to the setting of our process calculus.

*Definition 5.3 (Reducibility Candidates):* Given a type $A$ and a name $z$, a reducibility candidate at $z{:}A$, written $\mathcal{R}[z{:}A]$ is a set of processes satisfying the following properties:

1) If $P \in \mathcal{R}[z{:}A]$ then $\cdot;\cdot;\cdot \vdash_\eta P :: z{:}A$.
2) If $P \in \mathcal{R}[z{:}A]$ then $P \Downarrow$.
3) If $P \in \mathcal{R}[z{:}A]$ and $P \Rightarrow P'$ then $P' \in \mathcal{R}[z{:}A]$
4) If for all $P_i$ such that $P \Rightarrow P_i$ we have $P_i \in \mathcal{R}[z{:}A]$ then $P \in \mathcal{R}[z{:}A]$.

We refer to $\mathbb{R}[-{:}A]$ as the collection of all sets of reducibility candidates at (closed) type $A$. Our definition of the logical predicate identifies processes up to the compatible extension of structural congruence with the well-known *sharpened replication axioms* [13], written $\equiv_!$. The replication axioms express strong behavioral equivalences in our typed setting [12].

*Definition 5.4:* We write $\equiv_!$ for the least congruence relation on process expressions resulting from extending structural congruence $\equiv$ with the following axioms:

1) $(\nu u)(!u(z).P \mid (\nu y)(Q \mid R)) \equiv_!$
$(\nu y)((\nu u)(!u(z).P \mid Q) \mid (\nu u)(!u(z).P \mid R))$
2) $(\nu u)(!u(y).P \mid (\nu v)(!v(z).Q \mid R))$
$\equiv_!$ $(\nu v)((!v(z).(\nu u)(!u(y).P|Q)) \mid (\nu u)(!u(y).P|R))$
3) $(\nu u)(!u(y).Q \mid P) \equiv_!$ $P$ if $u \notin fn(P)$

Intuitively, axioms (1) and (2) represent the distribution of shared servers among "client" processes, and (3) garbage collects shared servers which can no longer be invoked.

*1) Logical Predicate:* We define a logical predicate on processes by induction on types and the size of typing contexts in the case of processes. The predicate captures the computational behavior of non-divergent processes, as defined by their typing. In the development below, we make use of $\mathcal{L}[\tau]$, which denotes the logical interpretation of the values exchanged in communication (i.e. sets for well-typed values of the appropriate type). We omit this definition due to its simplicity and for the sake of conciseness.

*Definition 5.5 (Logical Predicate - Open Process Expressions):* Given $\Psi;\Gamma;\Delta \vdash_\eta T$ with a non-empty left hand side environment, we define $\mathcal{L}^\omega[\Psi;\Gamma;\Delta \vdash_\eta T]$, where $\omega$ is a mapping from type variables to reducibility candidates, as the set of processes inductively defined by the rules of Fig. 7.

The definition of the logical interpretation for open processes inductively composes the process with the appropriate witnesses in the logical interpretation at the types specified in the three contexts, following [12].

The key part of our development is the definition of the logical predicate for closed processes. Similar to the treatment of type variables in logical relations for polymorphic languages (c.f. [8]), we employ a mapping from type variables to candidates at a given type. More precisely, we map type variables to candidates at the appropriate coinductive type, representing the unfolding of coinductive types.

We interpret a coinductive session type $\nu X.A$ as the union of all reducibility candidates $\psi$ of the appropriate coinductive type that are in the interpretation of the *open* type $A$, when $X$ is mapped to $\psi$ itself, enabling a principle of proof by coinduction.

*Definition 5.6 (Logical Predicate - Closed Process Expressions):* For any type $T = z{:}A$ we define $\mathcal{L}^\omega[T]$ as the set of all processes $P$ such that $P\Downarrow$ and $\cdot;\cdot;\cdot \vdash_\eta P :: T$ satisfying the rules of Fig. 8.

We elide several technical aspects and focus mainly on the intuitions behind the development. 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.

*Definition 5.7:* Let $\nu X.A$ be a strictly positive type. We define: $\phi_A(s) \triangleq \mathcal{L}^{\omega[X\mapsto s]}[z{:}A]$

*Theorem 5.8 (Greatest Fixpoint):* $\mathcal{L}^\omega[z{:}\nu X.A]$ is a greatest fixpoint of $\phi_A$.

Using Theorem 5.8 we can show an unfolding lemma for coinductive session types, relating type unfolding with the interpretation of open types.

*Lemma 5.9 (Unfolding):* $P \in \mathcal{L}^\omega[z{:}A\{\nu X.A/X\}]$ iff $P \in \mathcal{L}^{\omega[X\mapsto\mathcal{L}^\omega[-:\nu X.A]]}[z{:}A]$

$$
\begin{array}{lll}
P \in \mathcal{L}^{\omega}[\Psi, x{:}\tau; \Gamma; \Delta \vdash_{\eta} T] & \text{iff} & \forall M \in \mathcal{L}[\tau].P\{M/x\} \in \mathcal{L}^{\omega}[\Psi; \Gamma; \Delta \vdash_{\eta} T] \\
P \in \mathcal{L}^{\omega}[\Gamma; \Delta, y{:}A \vdash_{\eta} T] & \text{iff} & \forall R \in \mathcal{L}^{\omega}[y{:}A].(\nu y)(R \mid P) \in \mathcal{L}^{\omega}[\Gamma; \Delta \vdash_{\eta} T] \\
P \in \mathcal{L}^{\omega}[\Gamma, u{:}A; \Delta \vdash_{\eta} T] & \text{iff} & \forall R \in \mathcal{L}^{\omega}[y{:}A].(\nu u)(!u(y).R \mid P) \in \mathcal{L}^{\omega}[\Gamma; \Delta \vdash_{\eta} T]
\end{array}
$$

Fig. 7. Logical Predicate - Open Processes

$$
\begin{array}{ll}
\mathcal{L}^{\omega}[z{:}\nu X.A] & \triangleq \bigcup\{\Psi \in \mathbb{R}[-{:}\nu X.A] \mid \Psi \subseteq \mathcal{L}^{\omega[X \mapsto \Psi]}[z{:}A]\} \\
\mathcal{L}^{\omega}[z{:}X] & \triangleq \omega(z)(X) \\
\mathcal{L}^{\omega}[z{:}\mathbf{1}] & \triangleq \{P \mid \forall P'.(P \Longrightarrow P' \wedge P' \not\longrightarrow) \Rightarrow P' \equiv_! \mathbf{0}\} \\
\mathcal{L}^{\omega}[z{:}A \multimap B] & \triangleq \{P \mid \forall P' y.(P \overset{z(y)}{\Longrightarrow} P') \Rightarrow \forall Q \in \mathcal{L}^{\omega}[y{:}A].(\nu y)(P' \mid Q) \in \mathcal{L}^{\omega}[z{:}B]\} \\
\mathcal{L}^{\omega}[z{:}A \otimes B] & \triangleq \{P \mid \forall P' y.(P \overset{\overline{(\nu y) z\langle y\rangle}}{\Longrightarrow} P') \Rightarrow \\
 & \quad \exists P_1, P_2.(P' \equiv_! P_1 \mid P_2 \wedge P_1 \in \mathcal{L}^{\omega}[y{:}A] \wedge P_2 \in \mathcal{L}^{\omega}[z{:}B])\} \\
\mathcal{L}^{\omega}[z{:}!A] & \triangleq \{P \mid \forall P'.(P \Longrightarrow P') \Rightarrow \exists P_1.(P' \equiv_! !z(y).P_1 \wedge P_1 \in \mathcal{L}^{\omega}[y{:}A])\} \\
\mathcal{L}^{\omega}[z{:}\,\&\,\{\overline{l_i \Rightarrow A_i}\}] & \triangleq \{P \mid \bigwedge_i\ (\forall P'.(P \overset{z.l_i}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A_i])\} \\
\mathcal{L}^{\omega}[z{:}\oplus\{\overline{l_i \Rightarrow A_i}\}] & \triangleq \{P \mid \bigwedge_i\ (\forall P'.(P \overset{\overline{z.l_i}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A_i])\} \\
\mathcal{L}^{\omega}[z{:}\tau \wedge A] & \triangleq \{P \mid \forall P'.(P \overset{z\langle M\rangle}{\Longrightarrow} P') \Rightarrow (M \in \mathcal{L}[\tau] \wedge P' \in \mathcal{L}^{\omega}[z{:}A])\} \\
\mathcal{L}^{\omega}[z{:}\tau \supset A] & \triangleq \{P \mid \forall P', M.(M \in \mathcal{L}[\tau] \wedge P \overset{z(M)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A]\}
\end{array}
$$

Fig. 8. Logical Predicate - Closed Processes

The combination of these results enables us to show the fundamental lemma: all well-typed processes are in the logical predicate, from which follows that all well-typed processes are compositionally non-divergent.

*Theorem 5.10 (Fundamental Lemma):* If $\Psi; \Gamma; \Delta \vdash_{\eta} P :: z{:}A$ then $P \subseteq \mathcal{L}[\Psi; \Gamma; \Delta \vdash_{\eta} z{:}A]$.

The proof proceeds by induction on typing. The case for rule $\nu\mathsf{L}$ follows from Lemma 5.9. The most interesting case is the one for the $\nu\mathsf{R}$ rule. Since the interpretation of coinductive types $\nu X.A$ is a greatest fixpoint (Theorem 5.8), we proceed by coinduction: we produce a set of processes $\mathcal{C}_{P'}$, containing the body of the corecursive definition $P'$ closed under composition with representatives for the linear and exponential contexts, where the corecursion variable has been instantiated with an unfolding of the corecursive definition (let us refer to this process a $P''$). $\mathcal{C}_{P'}$ is also closed under reduction and expansion. We show that $\mathcal{C}_{P'}$ is a reducibility candidate at $\nu X.A$ and that $\mathcal{C}_{P'} \subseteq \mathcal{L}^{\omega[X \mapsto \mathcal{C}_{P'}]}[z{:}A]$. This is a sufficient condition since $\mathcal{L}[c{:}\nu X.A]$ is the largest such set. We need essentially focus on $P''$.

Showing that $\mathcal{C}_{P'}$ satisfies this condition relies crucially on type preservation, progress and the fact that type variables are ultimately offered by the unfolding of the corecursive definition. The crucial points are the occurrences of the corecursion variable in $P'$, which in $P''$ are instantiated with $P'$ itself: if its a terminal occurrence the property is immediate, since the corecursion variable is typed with the type variable $X$, which is mapped to $\mathcal{C}_{P'}$, containing $P'$. If the corecursion variable occurs in the left branch of a cut, we know that the only possible use of the fresh channel (typed with the type variable) by the right branch of the cut is to eventually forward it, potentially after a number of internal reductions or observable actions as specified by the session $A$. When the forwarder is triggered, we must necessarily be at the type variable $X$ and we can conclude since the resulting process is $P'$, in $\mathcal{C}_{P'}$. We remark that if the corecursion variable is guarded in the left branch of a cut, these guards must be consumed by the right branch of the cut (this follows by the well-typedness and progress) through internal reductions.

*Corollary 5.11 (Non-Divergence):* If $\Psi; \Gamma; \Delta \vdash_{\eta} P :: z{:}A$ then $P \Downarrow$

Combining type safety (Theorem 5.1 and Theorem 5.2) and non-divergence (Corollary 5.11) we conclude that typing enables strong guarantees on programs written in our calculus. A well-typed program will always be able to fulfil its protocol: no divergence can take place, nor can any deadlocks occur. Moreover, these properties are *compositional*, ensuring that any (well-typed) composition of services produces a deadlock-free, non-divergent service.

## VI. Related Work and Concluding Remarks

Forms of general recursion have been often introduced in session typed languages, but without much concern on how to conciliate unbounded behaviour with local termination (e.g, [2], [14]). From the perspective of more traditional work on session typed languages, [15] establishes a strong normalization result for action types, which are similar to session types, but without addressing recursive types. Still within the logic based approach to session typed programming, our work is related to [5], which develops a logical interpretation of session types using second-order classical linear logic and a polymorphic session-typed functional language similar to that of [16], but does not consider coinductive sessions. In prior work, we have studied strong normalisation for session types based on linear logic, including extensions to behavioral polimorphism [12], [8], however, this work is the first to address general (co)recursion in the framework.

Expressiveness relationships between replication and recursion in the context of $\pi$-calculi have been investigated (e.g, see [17]); in general such constructs are not reducible to each other, in particular in the presence of name scoping constructs.

In our context, they are fundamentally separated by the typing discipline, which reflect different behavioral usages.

The particular issues related to coinductive session types have to the best of our knowledge been overlooked in the literature, which typically deals with general recursive types and definitions [2], [3], [16] and so is inherently non-divergent. In this regard our work is closer to that on termination for $\lambda$-calculi with inductive and coinductive types [18]. For instance, [19] develops a strong normalization result for the second-order $\lambda$-calculus with explicit inductive and coinductive types using logical relations. The restrictions imposed by our type system to ensure non-divergence are related to those developed in [20], [21] for the Coq proof assistant, but with obvious distinctions given the very different settings. Our interpretation of coinductive types as greatest fixed points is also related to the work of Baelde [22]. The main differences are that his work uses classical linear logic and doesn't consider a proof term assignment. The former makes the system and logical relation techniques substantially different since they rely on orthogonality, whereas ours do not; the latter leads Baelde's work towards proof search related techniques, which is in sharp contrast to our work.

The work on type-based methods for ensuring termination has been generalized to include inductive and coinductive definitions in the style of Coq [23] (although the proof of strong normalization is substantially more complex due to the expressive power of CIC and type annotations). It is not clear how to adapt these type-based methods to our setting since type annotations are a measure of the size of "values", which do not immediately apply to processes. Similar difficulties arise when considering copatterns [24], which seem to be inherently tied to the term structure.

### A. Concluding Remarks

We have developed a theory of coinductive definitions for synchronous $\pi$-calculus processes with corecursion, establishing a strong logical foundation based on intuitionistic linear logic which provides guarantees of deadlock freedom, session fidelity and, crucially, compositional non-divergence by typing, ensuring that services implemented in our calculus do not "get stuck", and never become unavailable through divergent behavior.

Our type system ensures termination by restricting the occurrences of the corecursion variable in corecursive process definitions. While the restrictions we impose are not particularly oppressive, they naturally still exclude processes that are non-divergent (for instance, the binary counter example of [9]), as often is the case in these settings given that productivity is undecidable in general. Having set forth a first significant benchmark, it is certainly a challenge for future work to find less restrictive or more general conditions for guaranteeing productivity in this setting, potentially adapting type-based termination techniques (viz. [23], [24]).

By establishing a logical foundation for coinductive session-typed processes, the work set forth in this paper is an important stepping stone towards the development of a dependent type theory rich enough to allow us to express and reason about concurrent session-based concurrency. These concurrent programs are often coinductive in nature, and are typically studied using coinductive proof techniques. Since we establish typed processes as coinductive objects, we may be able to use processes as witnesses to coinductive proofs. A sound (wrt an extensional typed equivalence, c.f. [8], [12]) notion of definitional equality of corecursive processes is also part of future work.

## REFERENCES


[1] K. Honda, "Types for dyadic interaction," in *CONCUR'93*, 1993, pp. 509–523.

[2] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *ESOP*, 1998, pp. 122–138.

[3] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *POPL*, 2008, pp. 273–284.

[4] L. Caires and F. Pfenning, "Session types as intuitionistic linear propositions," in *CONCUR'10*, 2010, pp. 222–236.

[5] P. Wadler, "Propositions as sessions," in *ICFP'12*, 2012, pp. 273–286.

[6] B. Toninho, L. Caires, and F. Pfenning, "Dependent session types via intuitionistic linear type theory," in *PPDP'11*, 2011, pp. 161–172.

[7] F. Pfenning, L. Caires, and B. Toninho, "Proof-carrying code in a session-typed process calculus," in *CPP*, 2011, pp. 21–36.

[8] L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho, "Behavioral polymorphism and parametricity in session-based communication," in *ESOP*, 2013, pp. 330–349.

[9] B. Toninho, L. Caires, and F. Pfenning, "Higher-order processes, functions, and sessions: A monadic integration," in *ESOP'13*, 2013, pp. 350–369.

[10] The Coq Development Team, *The Coq Proof Assistant Reference Manual*, 2013. [Online]. Available: http://coq.inria.fr

[11] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[12] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho, "Linear logical relations for session-based concurrency," in *ESOP'12*, 2012, pp. 539–558.

[13] D. Sangiorgi and D. Walker, *The $\pi$-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[14] S. Gay and M. Hole, "Subtyping for Session Types in the Pi Calculus," *Acta Informatica*, vol. 42, no. 2-3, pp. 191–225, 2005.

[15] N. Yoshida, M. Berger, and K. Honda, "Strong normalisation in the pi -calculus," *Inf. Comput.*, vol. 191, no. 2, pp. 145–202, 2004.

[16] S. Gay and V. T. Vasconcelos, "Linear type theory for asynchronous session types," *J. Funct. Programming*, vol. 20, no. 1, pp. 19–50, 2010.

[17] J. Aranda, C. D. Giusto, C. Palamidessi, and F. D. Valencia, "On recursion, replication and scope mechanisms in process calculi," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006*, ser. Lecture Notes in Computer Science, vol. 4709. Springer, 2006, pp. 185–206.

[18] A. Abel, "Strong normalization and equi-(co)inductive types," in *Typed Lambda Calculi and Applications*, 2007, pp. 8–22.

[19] N. P. Mendler, "Inductive types and type constraints in the second-order lambda calculus," *Annals of Pure and Applied Logic*, vol. 51, no. 12, pp. 159 – 172, 1991.

[20] E. Giménez, "Codifying guarded definitions with recursive schemes," in *TYPES '94*, pp. 39–59.

[21] E. Gimenez, "Structural recursive definitions in type theory," in *Automata, Languages and Programming, 25th International Colloquium, ICALP98*. Springer-Verlag, 1998, pp. 13–17.

[22] D. Baelde, "Least and greatest fixed points in linear logic," *ACM Transactions on Computational Logic*, vol. 13, no. 1, Jan. 2012. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/baelde12tocl.pdf

[23] B. Grégoire and J. L. Sacchini, "On strong normalization of the calculus of constructions with type-based termination," in *LPAR*, 2010, pp. 333–347.

[24] A. Abel and B. Pientka, "Wellfounded recursion with copatterns: A unified approach to termination and productivity," in *LICS*, 2013.