Corecursion and Non-Divergence in Session-Typed Processes

Bernardo Toninho[1,2], Luis Caires[1], and Frank Pfenning[2]

[1] Universidade Nova de Lisboa, Portugal
[2] Carnegie Mellon University, USA

Abstract. Session types are widely accepted as an expressive discipline for structuring communications in concurrent and distributed systems. In order to express infinitely unbounded sessions, session typed languages often include general recursion which may 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 π-calculus with general (co)recursion, while still allowing interesting infinite behaviors to be definable. Our approach builds on a Curry-Howard correspondence between our type system and 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. We also establish, using a logical relation argument, that well-typed processes are compositionally non-divergent, that is, that no well-typed composition of processes, including those dynamically assembled via name passing, can result in divergent behavior.

1 Introduction

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 characterizing and analyzing system behavior have been used to verify properties such as deadlock freedom, protocol compliance, and availability in distributed and service based systems.

Among type-based approaches to verification, a rather successful technique has been that of *session types* [1–3]. Session types structure message-based concurrency based on 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 session types and linear logic in the style of Curry-Howard [4], linking proofs with typed processes and proof reduction with 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 typing guarantees such as deadlock freedom and session fidelity.

Although in the untyped setting infinite behavior is encodable using (π-calculus) replication, more "practical" session typed languages often introduce general recursion at both the program and type level [2]. In a typed setting, replication can only capture *finite* session behaviors, even if replicated arbitrarily often. It is insufficient to model *infinite session behavior*, or repeating behavioral patterns depending on evolving state.

Unfortunately, existing session type systems for languages equipped with general recursion do not avoid validating systems exhibiting undesirable *internal divergence*, e.g., infinite sequences of unobservable internal reduction steps. While this issue already arises at the level of individual systems, it becomes more serious when one needs to consider dynamically linked systems. For example, plugging together subsystems, e.g. as a result of dynamic channel passing or of linking higher-order code, may undesirably result in a system unable to offer its intended services due to divergent behavior, even if the subsystems are independently well-typed and divergence-free.

In this work, we tackle the challenge, within a session typed framework, of reconciling general recursion, enabling potentially infinite behavior to be expressed, with local termination (non-divergence), strong normalization, and compositionality (i.e. any well-typed process composition is non-divergent).

We illustrate our language with a toy example: consider a Twitter web service offering a replicated session $trends$ which is intended to produce a stream of current trends, according to some custom metrics. The service is parametrized by a filter session that given a stream of tweets produces a stream of trends. The session types involved are ($\multimap$ and $\wedge$ denote session data input and data output, respectively, and ! service offering):

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

Type TrendService specifies a replicated service that given an appropriate filter process providing the trend metrics specified by the client, transforms a stream of tweets (a recursive session that outputs tweets) into a stream of trends (a recursive session that outputs trends), produces the stream of trends the client wishes to measure. A possible client for the service is the following process (we write $\mathsf{corec}\ Z.P$ for a corecursive definition of P with recursion variable Z):

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

The Client process invokes the shared server, resulting in a fresh session on channel x, sends a handle k to the analytics package F_k, and then sits in a loop printing out (on session p) each trend received from the server on session x. In our type system, we may derive the typing judgment: trends:TrendService $\vdash$ Client :: p:Trends. We follow the formulation of linear logic based session types of [4], where typing judgments have the form $\Gamma; \Delta \vdash P :: x{:}U$. Such a judgment states of process P that it provides a session of type U at x, when composed with services / sessions as specified by $\Gamma; \Delta$. We may compose the client and the trend service (offered by a process Serv) into a closed system $\mathsf{Sys} \triangleq (\boldsymbol{\nu}\mathit{trends})(\mathsf{Serv} \mid \mathsf{Client})$. Sys will (unboundedly) print out trends on channel p. 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, as a result of being well-typed in our type system.

It is challenging to obtain an expressive and flexible typing discipline for general co-recursion (provably) ensuring the compositional non-divergence property, as we do in this work. For instance, consider the similar looking processes:

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

Both processes do not autonomously diverge. It is possible to type Good in our system, ensuring that it will never diverge, even when composed with arbitrary (well-typed) processes. However, this is not the case for Loop which produces an infinite reduction sequence after the first communication on c (see Section 4), and is not well-typed.

Our typing discipline eliminates *unproductive* internal behavior, ensuring together with global progress (actually, lock-freedom) and protocol fidelity, the *compositional non-divergence* of infinite behaviors (i.e. 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 summarize 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, ensuring that any well-typed service implemented in our calculus may never become unavailable through divergent behavior.

2 Process Model

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

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

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 π-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.

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

Fig. 1. Structural Congruence

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

Fig. 2. Reduction

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 – written as V – before communication takes place (the reduction rules for terms and the compatible closure rules are omitted). We note the standard unfolding semantics for corecursion. We write $\Rightarrow$ for the reflexive transitive closure of $\longrightarrow$ and $P \Downarrow$ iff P is non-divergent (i.e. no infinite reduction sequence starting with P).

To define the interactions of a process with its environment we adopt the early labelled transition system for the π-calculus [10] extended with the appropriate labels and rules for choice, value communication, forwarding and corecursion. A transition $P \xrightarrow{\alpha} Q$ denotes that process P may evolve to Q by performing the action represented by label α. Transition labels are defined as:

$$\alpha ::= \tau \mid x(y) \mid x(V) \mid x.l \mid \overline{x\,y} \mid \overline{x\,V} \mid \overline{x\langle y\rangle} \mid \overline{x.l}$$

Actions are channel input $x(y)$, value input $x(V)$, the offer of a labelled choice $x.l$ and their matching co-actions, respectively the channel output $\overline{x\,y}$, value output $\overline{x\,V}$ and bound output $\overline{x\langle y\rangle}$ actions, and the selection $\overline{x.l}$. The bound output $\overline{x\langle y\rangle}$ denotes extrusion of a fresh name y along x. Internal action is denoted by τ. For conciseness we highlight only the transition rules for forwarding and corecursion, which are given a silent transition semantics:

$$(\nu x)(P \mid [y \leftrightarrow x]) \xrightarrow{\tau} P\{y/x\}\ (y \neq x) \quad (\mathsf{corec}\ X(\overline{y}).P)\ \overline{c} \xrightarrow{\tau} P\{\overline{c}/\overline{y}\}\{(\mathsf{corec}\ X(\overline{y}).P)/X\}$$

The remaining rules for the labelled transitions semantics are identical to those in [4].

3 Type System

In this section we motivate and present our type system based on intuitionistic linear logic. The syntax of types is given below:

$$\tau, \sigma \quad ::= \mathtt{nat} \mid \mathtt{string} \mid \ldots$$
$$A, B, C ::= \tau \supset A \mid \tau \wedge A \mid A \multimap B \mid A \otimes B \mid \mathbf{1} \mid \&\{\overline{l_j{:}A_j}\} \mid \oplus\{\overline{l_j{:}A_j}\} \mid !A \mid \nu X.A \mid X$$

We distinguish the types of data τ, σ from sessions A, B, C. The language of session types covers the standard session constructs: data input and output ($\tau \supset A$ and $\tau \wedge A$), 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 sessions ($\nu X.A$).

From General Recursion to Corecursion While both recursive and coinductive session types denote potentially infinite session behavior, the fundamental distinction is that general recursion might generate an infinite sequence of *internal* actions and thus cause divergence, 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 – i.e. be *productive* – and thus cannot diverge. It is this *external*, observable behavior that may be infinite in a coinductively defined, session typed process. 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 impose a discipline on the occurrence of the recursion variable in processes, in line with the work on coinductive definitions in dependent type theories such as that of Coq [11] or Agda [12], but here mapped to a concurrent setting, where types describe *behavior* that is produced by processes rather than *data* that is constructed by functions, making many of the more recent type-based termination methods [13] based on indexing constructors for coinductive data types not immediately applicable. Essentially, we require that a corecursive process definition be *productive* – there is always a finite sequence of internal actions between *observable* actions.

Observable actions are those that take place on the session channel that is being *offered* by a given process, whereas internal actions are those generated through interactions with ambient sessions. Given our notion 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). However, guardedness alone is not sufficient to ensure productivity. Process Loop from Section 1 is guarded but produces divergent behavior when composed with a process that provides it with an output. The issue 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, destroying the productivity of the original definition.

Thus, ensuring non-divergence in a compositional way requires not only guardedness but also disallowing interactions with corecursive calls within corecursive definitions, a property we call *co-regular recursion*. To this end, we must impose that processes that are placed in parallel with corecursive calls may not communicate with the corecursive call, 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, ensuring that processes composed with corecursive calls may perform communication, but *not* with the corecursive call itself.

3.1 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

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

$$(\mathbf{1}\mathsf{L})\quad \frac{\Psi; \Gamma; \Delta \vdash_\eta P :: d{:}D}{\Psi; \Gamma; \Delta, c{:}\mathbf{1} \vdash_\eta P :: d{:}D} \qquad (\otimes\mathsf{R})\quad \frac{\Psi; \Gamma; \Delta_1 \vdash_\eta P_1 :: x{:}A \quad \Psi; \Gamma; \Delta_2 \vdash_\eta P_2 :: c{:}B}{\Psi; \Gamma; \Delta_1, \Delta_2 \vdash_\eta (\boldsymbol{\nu} x)c\langle x\rangle.(P_1 \mid P_2) :: c{:}A \otimes B}$$

$$(\otimes\mathsf{L})\quad \frac{\Psi; \Gamma; \Delta, x{:}A, c{:}B \vdash_\eta Q :: d{:}D}{\Psi; \Gamma; \Delta, c{:}A \otimes B \vdash_\eta c(x).Q :: d{:}D} \qquad (\multimap\mathsf{R})\quad \frac{\Psi; \Gamma; \Delta, x{:}A \vdash_\eta P :: c{:}B}{\Psi; \Gamma; \Delta \vdash_\eta c(x).P :: c{:}A \multimap B}$$

$$(\multimap\mathsf{L})\quad \frac{\Psi; \Gamma; \Delta_1 \vdash_\eta Q_1 :: x{:}A \quad \Psi; \Gamma; \Delta_2, c{:}B \vdash_\eta Q_2 :: d{:}D}{\Psi; \Gamma; \Delta_1, \Delta_2, c{:}A \multimap B \vdash_\eta (\boldsymbol{\nu} x)c\langle x\rangle.(Q_1 \mid Q_2) :: d{:}D} \qquad (\mathbf{1}\mathsf{R})\quad \frac{}{\Psi; \Gamma; \cdot \vdash_\eta \mathbf{0} :: c{:}\mathbf{1}}$$

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

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

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

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

Fig. 3. Typing Rules (abridged)

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 Δ, with the (unrestricted, or shared) session behaviors specified in Γ and where η is a mapping from (corecursive) type variables to typing contexts (we detail this further below). We note that the names in Γ, Δ and z are all pairwise distinct. We assume typing to be defined modulo structural congruence by definition. Context Ψ tracks the free variables in P that pertain to basic data values that are to be sent and received. We make use of judgment $\Psi \vdash M{:}\tau$ to denote that term M, denoting a value that is to be communicated, is well typed under the assumptions in Ψ.

The rules that define our type system are given in Fig. 3, 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. These restrictions are standard and thus enforced implicitly. As usual, in the presence of type annotations, typechecking is decidable.

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 Ψ, Δ, Γ or c. Moreover, we keep track of the contexts Ψ, Γ and Δ 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 η 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:

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

We type a process corecursion variable X by looking up in η the binding for X, which references the typing environments Ψ, Γ and Δ 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:

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

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 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. While we do not syntactically exclude processes without terminal recursion (for instance, process

Good from Section 1), a well-typed corecursive definition crucially *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. That is, we need to obtain a typing where the channel of the corecursive call is in the context, which is only possible through a cut.

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.

4 Examples

Excluding Unobservable Divergence Elaborating on the process Loop given in Section 1, we show how it can result in divergent behavior when interacting with a client and why Loop is not typeable in our system.

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):

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

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:

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

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

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

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

It is immediate 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.

The fundamental motivation for excluding these forms of unobservable divergence is that morally a process must offer some observable behavior in order to be useful. Realistically, even a divergent process should be receptive to a "kill" signal, which is encodable in our setting using choice and $\mathbf{1}$.

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

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 and then terminates. To provide the value of the counter, the coordinator propagates a val message along the network, which will compute the value as an integer and forward it back to the coordinator. 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, 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 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, from which it will receive the total counter value and send it forward to the coordinator. When an inc message is received, a node forwards it to its less significant bit neighbor, 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, signaling 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 neighbor. When a carry message reaches the coordinator, it generates a new bit, as mentioned above. The type for $\mathsf{Node}(b, x, n)$ is given by $x{:}\mathsf{CImpl} \vdash \mathsf{Node}(b, x, n) :: n{:}\mathsf{CImpl}$ with

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

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 given in Fig. 4.

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

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

To complete the system we provide the implementation of the empty bit string epsilon, which will be a closed process of type CImpl. For the val branch, it ping pongs the

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

Fig. 4. Node Process

received value. For incrementing, it emits the (first) carry message:

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

The system (offering c:Counter) is then produced by composing epsilon and Coord.

5 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, 14] to the coinductive setting, restoring the connection of our framework with the logical interpretation.

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

Theorem 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 2 (Progress). *If $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ and $live(P)$ then there is some Q with $P \to Q$ or one of the following holds:*

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

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 [14, 8], we build on Girard's technique of reducibility candidates: sets of non-divergent, well-typed terms, closed under reduction and (typed) expansion, adapted to the setting of our process calculus.

Definition 1 (Reducibility Candidates). *Given a type A and name z, a reducibility candidate at $z{:}A$, written $\mathcal{R}[z{:}A]$ is a set of closed, well-typed processes offering $z{:}A$ that satisfy the following:*

1. *If $P \in \mathcal{R}[z{:}A]$ then $P \Downarrow$.*
2. *If $P \in \mathcal{R}[z{:}A]$ and $P \Rightarrow P'$ then $P' \in \mathcal{R}[z{:}A]$*
3. *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* [10], written $\equiv_!$. The replication axioms express strong behavioral equivalences in our typed setting [14].

Definition 2. *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. $(\nu u)(!u(y).Q \mid P) \equiv_! P \quad$ *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.

We define a logical predicate on processes by induction on types and the size of typing contexts. 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 3 (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 ω is a mapping from type variables to reducibility candidates, as the set of processes inductively defined as:*

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

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 [14].

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

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

Fig. 5. Logical Predicate - Closed Processes

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 ψ of the appropriate coinductive type that are in the interpretation of the *open* type A, when X is mapped to ψ itself, enabling a principle of proof by coinduction.

Definition 4 (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. 5.*

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 (Theorem 3).

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

Theorem 3 (Greatest Fixpoint). *$\mathcal{L}^{\omega}[z{:}\nu X.A]$ is a greatest fixpoint of ϕ_A.*

The combination of these results enables us to obtain our main result (Theorem 4): all well-typed processes are in the logical predicate, from which follows that all well-typed processes are compositionally non-divergent.

Theorem 4. *If $\Psi;\Gamma;\Delta \vdash_{\eta} P :: z{:}A$ then $P \in \mathcal{L}[\Psi;\Gamma;\Delta \vdash_{\eta} z{:}A]$.*

Proof. We proceed by induction on typing. 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 3), we proceed by coinduction: we produce a set of processes $\mathcal{C}_{P'}$, containing the body of the corecursive definition P' where the corecursion variable has been instantiated with an unfolding of the corecursive definition (let us refer to this process as P''), closed under reduction and (typed) 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 to focus on P''.

Showing this property relies crucially on the fact that type variables are ultimately offered by the unfolding of the corecursive definition. The key 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 well-typedness and progress) through internal reductions.

Corollary 1 (Non-Divergence). *If* $\Psi; \Gamma; \Delta \vdash_\eta P :: z{:}A$ *then* $P \Downarrow$

Combining type safety (Theorems 1 and 2) and non-divergence (Corollary 1) we conclude that typing enables strong guarantees on processes written in our calculus. A well-typed process is 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) service composition produces a deadlock-free, non-divergent service.

6 Related Work and Concluding Remarks

Expressiveness relationships between replication and recursion in the context of π-calculi have been investigated (e.g, [15]); in general such constructs are not reducible to each other, in particular in the presence of name scoping constructs. In our context, replication codifies behavior that may be repeated arbitrarily often on independent channels, whereas recursion denotes potentially infinite behavior on the same channel.

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, 16]). From the perspective of more traditional work on session typed languages, [17] 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 [18], but does not consider coinductive sessions.

In prior work, we have studied strong normalization for session types based on linear logic, including extensions to behavioral polymorphism [14, 8], however, this work is the first to address general corecursion in the framework. While there are works that address termination by typability in the π-calculus, they either do not have explicit recursion [19] or consider simpler type systems [20] without notions of session fidelity or lock freedom and impose more intricate syntactic restrictions on processes.

The particular issues related to coinductive session types have to the best of our knowledge been overlooked in the traditional literature, which typically deals with general recursive types and definitions [2, 3, 18], including divergence by construction. More recently, [21] considers least and greatest fixpoints in classical linear logic (in the sense of [22]) as primitive recursors and corecursors in a session typed calculus, respectively. Since, in their setting, a corecursor can only synchronize with a recursor, it is not straightforward how to encode transducers that transform one coinductive session into another. This is a common programming pattern, where a service is offered through composition and transformation of a collection of other coinductive sessions. Moreover, their work does not develop a logical relation proof technique nor establish a non-divergence result, as we do.

In this regard our work is closer to that on termination for λ-calculi with inductive and coinductive types [23]. For instance, [24] develops a strong normalization result for the second-order λ-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 [25] 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 does not 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 [26] (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 [13], which seem to be inherently tied to the term structure.

Concluding Remarks We have developed a theory of coinductive definitions for a synchronous π-calculus with corecursion, establishing a logical foundation based on intuitionistic linear logic which provides guarantees of deadlock freedom, session fidelity and, crucially, compositional non-divergence by typing. Thus, services implemented in our calculus do not "get stuck", nor become unavailable due to divergent behavior.

Our type system ensures termination by eliminating unproductive internal behavior 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. [26, 13]).

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 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, 14]) notion of definitional equality of corecursive processes is also part of future work.

References

1. Honda, K.: Types for dyadic interaction. In: CONCUR'93. (1993) 509–523
2. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. (1998) 122–138
3. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. (2008) 273–284
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR'10. (2010) 222–236
5. Wadler, P.: Propositions as sessions. In: ICFP'12. (2012) 273–286
6. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: PPDP'11. (2011) 161–172
7. Pfenning, F., Caires, L., Toninho, B.: Proof-carrying code in a session-typed process calculus. In: CPP. (2011) 21–36
8. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: ESOP. (2013) 330–349
9. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: ESOP'13. (2013) 350–369
10. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
11. The Coq Development Team: The Coq Proof Assistant Reference Manual. (2013)
12. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
13. Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: LICS. (2013)
14. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: ESOP'12. (2012) 539–558
15. Aranda, J., Giusto, C.D., Palamidessi, C., Valencia, F.D.: On recursion, replication and scope mechanisms in process calculi. In: FMCO. (2006) 185–206
16. Gay, S., Hole, M.: Subtyping for Session Types in the Pi Calculus. Acta Informatica **42**(2-3) (2005) 191–225
17. Yoshida, N., Berger, M., Honda, K.: Strong normalisation in the pi -calculus. Inf. Comput. **191**(2) (2004) 145–202
18. Gay, S., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Programming **20**(1) (2010) 19–50
19. Deng, Y., Sangiorgi, D.: Ensuring termination by typability. In: IFIP TCS. (2004) 619–632
20. Sangiorgi, D.: Termination of processes. Math. Struct. in Comp. Sci. **16**(1) (2006) 1–39
21. Morris, J.G., Lindley, S., Wadler, P.: The least must speak with the greatest (2014) Draft.
22. Baelde, D.: Least and greatest fixed points in linear logic. TOCL **13**(1) (January 2012)
23. Abel, A.: Strong normalization and equi-(co)inductive types. In: TLCA. (2007) 8–22
24. Mendler, N.P.: Inductive types and type constraints in the second-order lambda calculus. Annals of Pure and Applied Logic **51**(12) (1991) 159 – 172
25. Gimenez, E.: Structural recursive definitions in type theory. In: ICALP. (1998) 13–17
26. Grégoire, B., Sacchini, J.L.: On strong normalization of the calculus of constructions with type-based termination. In: LPAR. (2010) 333–347