Inductive and Coinductive Session Types in Higher-Order Concurrent Programs

Bernardo Toninho
Carnegie Mellon University &
Universidade Nova de Lisboa
Pittsburgh, PA, USA
btoninho@cs.cmu.edu

Luís Caires
Universidade Nova de Lisboa
Lisbon, Portugal
luis.caires@di.fct.unl.pt

Frank Pfenning
Carnegie Mellon University
Pittsburgh, PA, USA
fp@cs.cmu.edu

Abstract

We develop a theory of inductive and coinductive session types in a computational interpretation of linear logic, enabling the representation of potentially infinite interactions in a compositionally sound way that preserves logical soundness, a major stepping stone towards a full dependent type theory for expressing and reasoning about session-based concurrent higher order distributed programs. The language consists of a λ-calculus with inductive types and a contextual monadic type encapsulating session-based concurrency, treating monadic values as first-class objects. We consider general fixpoint and cofixpoint constructs, subject to natural syntactic constraints, as a means of producing inductive and coinductive definitions of session-typed processes, that until now have only been considered using general recursion, which is incompatible with logical consistency and introduces compositional divergence. We establish a type safety result for our language, including protocol compliance and progress of concurrent computation, and also show, through a logical relations argument, that all well-typed programs are compositionally non-divergent. Our results entail the logical soundness of the framework, and enable compositional reasoning about useful infinite interactive behaviors, while ruling out unproductive infinite behavior.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures, Recursion; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Program and recursion schemes, Type structure

1. Introduction

We live in an age of concurrent 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. It thus becomes necessary to be able to precisely represent this concurrent cooperating behavior, in a way that is amenable to compositional reasoning and verification, so that we can ensure desirable properties of these services such as deadlock freedom, protocol compliance, high availability. Historically, concurrency theorists have used process calculi as models for these interactions, developing reasoning techniques for these languages based on the precise identification of system behavior. A rather successful technique has been that of *session types* [12–14]. 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.

Traditional session type systems typically include general recursion, required to express infinitely unbounded exchange of messages occurring in a distributed system. Superficially, it might seem that infinite behavior is already encodable using replication (which is indeed the case in the untyped π-calculus [20]). In a typed setting, replication can only capture *finite session behavior* that can be repeated arbitrarily often. It is not rich enough to model *infinite session behavior*, nor repeating behavioral patterns that depend some evolving state. For instance, we cannot use replication to encode an infinite ascending output of integers, since there is no way of representing the statefulness of the behavior. Thus, the inclusion of recursion is not optional in terms of expressiveness, and usefully complements replication. However, traditional session types for (channel-passing, higher-order) distributed systems with recursion fail to enforce the highly desirable property of *non-divergence*, meaning absence of non-terminating internal (unobservable) computations. Plugging together subsystems, for instance as a result of dynamic channel passing or of linking downloaded (higher-order) code to local services, may result in a system actually unable to offer its intended services due to divergent behavior, even if the component subsystems themselves are well-typed and divergence-free.

The recent discovery of a correspondence between synchronous session types and propositional linear logic in the style of Curry-Howard [5], linking proofs with typed processes and proof reduction as communication, has sparked a renewed interest in session types [24] and in the idea of exploiting this logical foundation to provide more powerful reasoning techniques about concurrent and distributed systems. This line of work has addressed concepts such as value-dependent types [22], proof-carrying code [19] and behavioral genericity [6], moving towards a more general type theory for message-passing concurrency, with strong guarantees by virtue of typing such as deadlock freedom and session fidelity.

Recently, a monadic integration of this linear logic interpretation in a general λ-calculus has been developed [23], enabling concurrent and functional programs to be combined in a clean and natural way. Moreover, this integration represents concurrent programs as first-class objects that can be passed in functions but also

[Copyright notice will appear here once 'preprint' option is removed.]

communicated and executed (mobile code), in the style of higher-order processes, in a deadlock-free way. This monadic integration, while closer to a more general theory of session-based programs, has one significant limitation when compared to earlier work insofar as it foregoes the full logical foundation for the sake of expressiveness, by incorporating general recursive types (which are unsound from a logical point of view) and through them, divergence.

In this paper we overcome this challenge by developing a theory of inductive and coinductive session types in a monadic integration of a computational interpretation of linear logic that preserves logical soundness, a major stepping stone towards a full dependent type theory for expressing and reasoning about session-based concurrent higher order programs.

Our introduction of recursion remains faithful to proper logical principles, insofar as we restrict general recursive session types to much more canonical inductive and coinductive session types (and their corresponding definitions), recovering non-divergence and establishing as a consequence the consistency of the underlying logical system. In addition to this fundamental result, our development also has non-trivial practical consequences, since it establishes by typing the *compositional non-divergence* of infinite behaviors (i.e. there is no well-typed context under which a well-typed infinite behavior will result in divergent behavior), which is (together with global progress and protocol fidelity) an important property of session-based systems out of the scope of existing session type systems with recursive types.

Contributions The contributions of this work are as follows:

- We develop a theory of inductive and coinductive definitions for concurrent higher-order synchronous processes embedded in a λ-calculus through a contextual monad.
- We study natural syntactic restrictions that guarantee the properties necessary to establish fixpoint and cofixpoint constructs as inductive and coinductive session-typed process definitions, considering the general computational unfolding rules.
- We show type safety of our type system, which entails protocol compliance and deadlock freedom, even in the presence of higher-order, inductive and coinductive concurrent programs.
- We prove that well-typed programs are *compositionally non-divergent* by employing logical relations, extended to inductive and coinductive session types, which combined with type safety ensures that any well-typed, distributed service implemented in our language will not only never "get stuck", but will also never become unavailable through divergent behavior.

2. Language

We now present our language, combining λ-calculus and concurrent computation through a *contextual monad* that encapsulates session-based concurrency (Fig. 1). We assume familiarity with the λ-calculus and its typical formulation. We assume less familiarity with session types and process calculi, to which we devote more of our presentation effort.

2.1 λ-terms and Types

Our λ-calculus is a typical λ-calculus extended with inductive types and the monadic type. At the level of λ-terms (ranged over by M, N), we have term constructors $\mathcal{C}$ and pattern matching (we refer to patterns with p) for inductive types (note that no pattern matching is possible on terms of monadic type), a constructor for monadic process expressions $a \leftarrow \{P\} \leftarrow \overline{b}$, where P is a process expression. We use a to refer to any session channel, but we syntactically distinguish between linear channels (c, x, y, z) and persistent channels $(!u, !v, !w)$. In the monadic objects and type, a denotes the offered session channel and the list $\overline{b}$ denotes the used ambient channels. For the sake of presentation we omit the ambient channel region when no such channels are used. We also include fixpoint and cofixpoint constructs, $\mathsf{fix}\ f{:}\tau = M$ and $\mathsf{cofix}\ f{:}\tau = M$, respectively. Fixpoints allow for the definition of functional objects of inductive type, and are the only construct that allow us to define process expressions of inductive session type, which represent repeating, but necessarily finite behavior along a specific session channel. Cofixpoints are restricted to monadic expressions of coinductive type, and can be intuitively seen as coinductive process definitions with (functional) parameters. Moreover, cofixpoint definitions in our λ-calculus are the only way we can define process expressions of coinductive type, which correspond to infinite, or potentially infinite, behavior along a specific session channel.

We range over λ-calculus types (Fig. 1) with τ, σ, which include the usual function types $\tau \to \sigma$, inductive types $\mu X.\tau$ (with X occurring positively in τ), inductive type variables X and the monadic type $\{\overline{b{:}A_i} \vdash a{:}A\}$, where A is a session type.

2.2 Sessions and Process Expressions

As was highlighted in the introduction, session types are essentially descriptions of communication behavior. Here, following the linear logic interpretation, we use intuitionistic linear logic connectives as the constructors for session types, which are assigned to channels. Thus, a session type describes the communication behavior of a process expression on a particular channel, over time. Since we are in an intuitionistic setting, we make a distinction between the ambient sessions (the left-hand-side context) and the session that is being offered (this is key to ensure non-divergent behavior of corecursive definitions). This is made explicit in the monadic type $\{\overline{b{:}A_i} \vdash a{:}A\}$, where the $\overline{b{:}A_i}$ denotes the session context (a set of channel identifiers and their respective types) in which a term of the monadic type will be able to offer the session behavior A on channel a. As we will see, this session context must be explicitly provided in order for the monadic term to be executed.

Our syntax (Fig. 1) of session types (ranged over with A, B, C) allows us to represent the following interaction behaviors (we use linear logic syntax and terminology) along a channel: an inactive or terminated session is represented by type $\mathbf{1}$, denoting a session channel along which no further interaction is possible; a persistent, replicated server offering behavior A an arbitrary number of times is represented via the linear logic exponential $!A$; channel input capability is expressed by linear implication $A \multimap B$, denoting a session that will input a session channel of type A and proceed as a session of type B; dually, channel output capability is expressed through multiplicative conjunction, $A \otimes B$, where A is the type of the *fresh* channel that is output and B is the type for the remainder of the session; the communication of functional data (including monadic objects) is expressed through the types $\tau \supset A$ and $\tau \wedge A$, where the former denotes a session that inputs a value of type τ and continues as A and the latter denotes the dual output behavior; a (guarded) choice of behaviors is offered by a labelled choice type $\&\{\overline{l_i{:}A_i}\}$, representing a session that waits on a choice between the several session behaviors A_i, distinguished with their respective labels l_i; dually, alternative behavior is denoted by the $\oplus\{\overline{l_i{:}A_i}\}$ session type, identifying a session that after signaling the appropriate label will proceed as specified by one of the session types A_i. Finally, inductive and coinductive session types are represented with types $\mu X.A$ and $\nu X.A$, respectively, restricted to strictly positive occurrences of X in A. As mentioned above, an inductive session type denotes a repetitive finite behavior (where the actual number of repetitions is undefined). Thus, a process expression implementing an inductive session will eventually reach an inactive or terminated state ($\mathbf{1}$). On the other hand, a coinductive

$$
\begin{array}{rcl}
M, N & ::= & \lambda{:}\tau.M \mid M\ N \mid x \mid \mathsf{case}\ M\ \mathsf{of}\ \overline{p_i \Rightarrow N_i} \\
& \mid & \mathsf{fix}\ f{:}\tau = M \mid \mathsf{cofix}\ f{:}\tau = M \mid a \leftarrow \{P\} \leftarrow \overline{b} \\
& \mid & \mathcal{C}(\overline{M}) \\
\tau, \sigma & ::= & \tau \to \sigma \mid \mu X.\tau \mid X \mid \{\overline{b{:}A_i} \vdash a{:}A\} \\
P, Q & ::= & c\langle M\rangle.P \mid \overline{c}\langle x\rangle.(P \mid Q) \mid c(x).P \mid c.l; P \\
& \mid & c.\mathsf{case}(\overline{l_i \Rightarrow P_i}) \mid !u(x).P \mid [x \leftrightarrow y] \mid \mathsf{close}_c \\
& \mid & \mathsf{wait}_c.P \mid \mathsf{spawn}(M; \overline{d}; a.P) \mid \overline{u}_!\langle x\rangle.P \\
A, B & ::= & !A \mid A \multimap B \mid A \otimes B \mid \tau \supset A \mid \tau \wedge A \mid \mathbf{1} \\
& \mid & \&\{\overline{l_i{:}A_i}\} \mid \oplus\{\overline{l_i{:}A_i}\} \mid \mu X.A \mid \nu X.A \mid X
\end{array}
$$

Figure 1. Syntax: Terms, Processes and Types

session type denotes a potentially infinite session, and so a process expression implementing such a behavior is not guaranteed to reach an inactive state. Non-divergence of coinductive sessions is thus not equated with reaching an inactive state (termination *per se*), but rather a guarantee that a coinductively defined process expression will always have a finite sequence of internal actions before offering some behavior along its specified session channel.

Sessions are implemented by process expressions, the syntax of which is specified in Fig. 1 and ranged over with P, Q. Value output and input (along channel c) is implemented by the constructs $c\langle M\rangle.P$ and $c(x).P$ respectively, where x is bound in the continuation process P. We note that communication actions are synchronous, or blocking. Session channel output is implemented by the expression $\overline{c}\langle x\rangle.(P \mid Q)$, where x is always a fresh channel name, P implements the required session behavior on channel x and Q implements the continuation behavior (typing will ensure disjointness of channel names between P and Q). Channel input is overloaded with the same input construct as before. Offering a choice along channel c is embodied by the $c.\mathsf{case}(\overline{l_i \Rightarrow P_i})$. where each label represents one possible choice with its associated process behavior P_i. Performing a choice (or a signaling alternative behavior) is done with the $c.l; P$ construct, where l is the label representing the appropriate choice. Replication is always input-guarded, so of the form $!u(x).P$, modelling a shared server spawning threads for client requests. Server invocations are performed by the $\overline{u}_!\langle x\rangle.P$ construct, spawning a fresh server session instance bound to x in P. These constructs correspond respectively to the accept $u(x)$ in P and request $u(x)$ in P forms of [13].

We allow for channel forwarding $[x \leftrightarrow y]$, which essentially consists of using an ambient channel to offer a service along another channel (from a logical point of view, this corresponds to the identity principle). We offer primitives for closing a session channel (close_c) and its dual $\mathsf{wait}_c.P$, which waits for termination of an ambient session channel and continues as P. Finally, we have a composition construct $\mathsf{spawn}(M; \overline{d}; a.P)$, that given a monadic term M will spawn its underlying process expression in parallel with the process P, which will be able to interact through the (fresh) channel a. Since our monad is contextual, M may require some ambient sessions in order to actually execute, which are provided via the $\overline{d}$ parameter of the construct (omitted when not used).

2.3 Operational Semantics

The operational semantics of our language are defined in a standard way for both λ-terms and process expressions. We define a call-by-value reduction relation on both λ-terms and process expressions, written $M \longrightarrow M'$ and $P \longrightarrow P'$, respectively, whose rules are given in Fig. 3 (we use V to denote a term that has been reduced to a value). We omit the compatible closure rules for the sake of presentation. The normal forms for the λ-terms are λ-abstractions,

$$
\begin{array}{ll}
x \notin fn(P) \Rightarrow P \,|\, (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \,|\, Q) & P \equiv_\alpha Q \Rightarrow P \equiv Q \\
P \mid (Q \,|\, R) \equiv (P \,|\, Q) \mid R & P \,|\, Q \equiv Q \,|\, P \\
(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P & [y \leftrightarrow x] \equiv [x \leftrightarrow y]
\end{array}
$$

Figure 2. Structural Congruence

$$
\begin{array}{l}
(\lambda x : \tau.M)\, V \longrightarrow M\{V/x\} \\
\mathsf{fix}\ f = M \longrightarrow M\{\mathsf{fix}\ f = M/f\} \\
\mathsf{cofix}\ f = M \longrightarrow M\{\mathsf{cofix}\ f = M/f\} \\
\mathsf{case}\ c_i(\overline{M})\ \mathsf{of}\ \overline{c_i(\overline{x_i}) \Rightarrow N_i} \longrightarrow N_i\{\overline{M}/\overline{x_i}\} \\
\\
(\nu c)(c\langle V\rangle.P \mid c(x).Q) \longrightarrow (\nu c)(P \mid Q\{V/x\}) \\
(\nu c)(\overline{c}\langle y\rangle.P \mid c(x).Q) \longrightarrow (\nu c)(\nu y)(P \mid Q\{y/x\}) \\
(\nu u)(\overline{u}_!\langle y\rangle.P \mid !u(x).Q) \longrightarrow \\
\qquad (\nu c)(\nu y)(P \mid Q\{y/x\} \mid !u(x).Q) \\
(\nu c)(c.\mathsf{case}(\overline{l_i \Rightarrow P_i}) \mid c.l_i; Q) \longrightarrow (\nu c)(P_i \mid Q) \\
(\nu c)(\mathsf{close}_c \mid \mathsf{wait}_c.P) \longrightarrow P \\
(\nu \overline{a})(\mathsf{spawn}(c \leftarrow \{P\} \leftarrow \overline{b}; \overline{a}; c.Q)) \longrightarrow \\
\qquad (\nu c)(\nu \overline{a})(P\{\overline{a}/\overline{b}\} \mid Q) \\
(\nu \overline{w})(\mathsf{spawn}(!u \leftarrow \{P\} \leftarrow \overline{!v}; \overline{w}; u.Q)) \longrightarrow \\
\qquad (\nu u)(\nu \overline{w})(!u(x).P\{x/u, \overline{w}/\overline{v}\} \mid Q) \\
(\nu x)(P_y \mid [y \leftrightarrow x]) \longrightarrow P\{y/x\} \\
\text{If } P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q' \equiv Q \text{ then } P \longrightarrow Q
\end{array}
$$

Figure 3. Reduction: λ-terms and Process Expressions

constants and, crucially, monadic terms. Note how the reduction rules for fixpoints and cofixpoints have no special restrictions in place, as one might perhaps expect in a language that is terminating (at least for the rule governing cofix). In fact, both rules are identical. The difference is in the well-typed objects one can define using the two constructs, which for fix will always denote finite behavior, but for cofix may not. However, unrolling a cofixpoint as specified in the reduction rules is not problematic since we restrict the form of cofixpoint definitions in such a way that the occurrence of f in M will be under a monadic constructor (which is a normal form). Moreover, the reduction rules for process expressions are the typical reduction rules for a synchronous π-calculus with guarded choice, which do not allow for reductions under prefixes. These rules arise naturally from the logical interpretation as cut reduction steps, closed under a structural congruence relation that captures fundamental structural equivalences (Fig. 2).

The rules for the spawn construct capture the process composition principles of our framework. Given a monadic value $c \leftarrow \{P\} \leftarrow \overline{b}$, the construct $\mathsf{spawn}(c \leftarrow \{P\} \leftarrow \overline{b}; \overline{a}; c.Q)$ will execute P and Q in parallel, instantiating the channel names $\overline{b}$ in P with the actual ambient channels $\overline{a}$ necessary for P to run, enabling the two processes to interact along the fresh channel c. For the case where we are spawning a persistent session (used in Q accordingly), the spawn operational semantics must compose the two processes by guarding P with a replicated input along the fresh channel u. To clarify the behavior of spawn, we exhibit a sample reduction below,

$$
\begin{array}{l}
(\nu a)(a\langle 0\rangle.\mathsf{close}_a \mid \mathsf{spawn}(c \leftarrow \{b(x).c\langle x\rangle.\mathsf{wait}_b.\mathsf{close}_c\} \leftarrow b; a; c.P)) \\
\quad \longrightarrow (\nu a)(\nu c)(a\langle 0\rangle.\mathsf{close}_a \mid a(x).c\langle x\rangle.\mathsf{wait}_a.\mathsf{close}_c \mid P)
\end{array}
$$

The monadic term expects a channel b along which to input a value that will be output along c. At runtime, b is instantiated with the *actual* channel a, as specified in the operand of spawn, resulting in a process that will synchronize along the channel a and then send the value along c to P.

In the remainder of this paper we write $\Rightarrow$ as the reflexive transitive closure of $\longrightarrow$. We also write $P \Downarrow$ (resp. $M \Downarrow$) when P

(resp. M) is non-divergent, that is, no infinite sequence of reduction steps exists starting from P (resp. M).

We also define a labelled transition system for process expressions, which is key to defining our linear logical relation. The rules define a so-called early transition system, specifying how processes interact with their environment, and are detailed in Fig. 4.

2.4 Coregular Recursion and Guardedness

How exactly can we define a recursive (be it inductive or coinductive) process in our framework, given that we have no recursion primitives in the process expression language? Previously, we mentioned that this was achieved through the fix and cofix constructs of our λ-calculus, but how exactly do we make a recursive call? Given that we can only define process expressions in the λ-calculus through the monad (and thus with an object of monadic type), the only way to actually perform the recursive call is in a spawn operation. However, spawning a monadic object is done along a fresh session channel, so to "close the loop" and have the recursive call continue on the initial session channel we must employ a forwarder.

The reader may wonder how we obtain a non-divergence result if recursive calls are spawned in parallel, potentially allowing for recursive processes to become "infinitely wide" (in the coinductive case). The answer to this question is twofold: for inductive definitions, we require all recursive calls to be performed on structurally smaller arguments, which is similar to the syntactic restrictions imposed in functional programs; for coinductive definitions we impose two syntactic restrictions that approximate productivity: *coregular recursion* and *guardedness*. These are related to the syntactic restrictions imposed in (terminating) coinductive functional programs in Coq [21], wich require recursive calls to occur under a constructor and as a top level subterm of the constructor [10]. In our setting, since session types are not defined in terms of constructors, these conditions do not apply.

Coregular recursion restricts the occurrence of the recursion variable in such a way that it can only occur in a terminal position in a process expression. Since recursive calls are made through the spawn construct, this means that the recursive must appear in a spawn in which the continuation is a forwarder, linking the channel of the recursive call back to the session that is being offered.

Guardedness requires *at least* one action on the channel that is being offered before the recursive call is made (guardedness further ensures that corecursive definitions are always functions that produce monadic objects). The precise definitions of guardedness and coregular recursion are given below. The former is presented as a predicate $\mathsf{Guarded}_f(P; c)$, stating that occurrences of f in P are guarded wrt session channel c (we use $\mathsf{Guarded}_f(M; c)$ as the compatible extension to λ-terms); the latter is presented as a predicate $\mathsf{coreg}_f(M)$.

Definition 2.1 (Guardedness)**.** *We state that a corecursive process definition* $\mathsf{cofix}\ f = M$ *is guarded if* M *is of the form* $\lambda x.c \leftarrow \{P\} \leftarrow \overline{d}$ *and satisfies the inductively defined predicate* $\mathsf{Guarded}_f(P; c)$ *(Fig. 5).*

Definition 2.2 (Coregular Recursion)**.** *We state that a corecursive definition* $\mathsf{cofix}\ f{:}\tau = M$*, written* $\mathsf{coreg}_f(M)$ *satisfies coregular recursion whenever occurrences of* f *in* M *are of the form* $\mathsf{spawn}(f\, N; \overline{d}; x.[x \leftrightarrow c])$.

For instance, the term below satisfies both guardedness and coregular recursion,

$$\mathsf{cofix}\ f = \lambda x.c \leftarrow \{d(n).c\langle n\rangle.\mathsf{spawn}(f\ x; d; y.[y \leftrightarrow c])\} \leftarrow d$$

whereas M_2 is not guarded but coregular, and M_3 is neither:

$$M_2 \triangleq \mathsf{cofix}\ f = \lambda x.c \leftarrow \{\ d().\mathsf{spawn}(f\, x; d; y.[y \leftrightarrow c])\} \leftarrow d$$

$$M_3 \triangleq \mathsf{cofix}\ f = \lambda x.c \leftarrow \{\mathsf{spawn}(f\, x; y.\mathsf{spawn}(f x; z.[z \leftrightarrow c]))\}$$

Guardedness ensures that a coinductive definition can always offer an action, since process expressions are always defined as offering a session on a channel and using some session context. As we will see in the following section, closed, well-typed process expressions have a single free (observable) channel name – the one along which they offer its session behavior. Therefore, to ensure productivity of a coinductive process we must enforce that only a finite number of internal actions can take place before an observable process action is made on the session channel. This is reasonable since process actions are synchronous, and thus blocking. Coregular recursion is necessary to disallow "infinitely wide" processes (an unbounded number of parallel compositions of the recursive process) and situations where processes that run in parallel with the recursive call, in some sense, destroy productivity of the coinductive definition. For instance, the process expression M_4 below,

$$M_4 \triangleq \mathsf{cofix}\ f = \lambda x.c \leftarrow \{c\langle\rangle.\mathsf{spawn}(f\, x; d; e.e().[e \leftrightarrow c]\} \leftarrow d$$

superficially seems non-problematic, since it offers an action on the offered channel c before the recursive call. However, after the spawn operation, the continuation will always consume the previously offered action and therefore the process will "loop" after performing the first output action. Intuitively, our guardedness restriction is the analogue of constructor guardedness in the functional setting, whereas coregular recursion is the analogue to requiring that recursive calls occur as a top level subterm of a constructor.

2.5 Type System

Having established the necessary conditions that guarantee the well-formedness of inductive and coinductive definitions in our setting, we now present the full type system of our language. We assume a signature of declarations of inductive types Σ, which is well-formed if each declaration is strictly positive and well-typed. In this simply-typed setting this is rather simple and we omit the details of this for conciseness. The typing rules for our language are given in Fig. 6. We define two typing judgments: $\Psi; \Gamma; \Delta \vdash_\Sigma P :: z{:}A$ for process expressions and $\Psi \vdash_\Sigma M : \tau$ for λ-terms. In the former, Δ is a linear context of session channels and Γ is a persistent context of session channels, whereas Ψ is an intuitionistic context which tracks variables of λ-terms. The meaning of the former typing judgment is that the process expression P will provide the session behavior of type A along channel z when composed (through spawn, cut and $\mathsf{cut}^!$, which enforce composition with name hiding) with processes providing the (linear) sessions specified in Δ and the (persistent) sessions specified in Γ, with free term variables specified in Ψ. We recall that inductive and coinductive types have strictly positive occurrences of the type variable.

Typing λ-terms The rules for λ-terms are somewhat straightforward. The rule for the case construct requires the type of the term that is being case analyzed to be some inductive type I with a constructor matching the pattern. We elide a more precise characterization of pattern matching (such as exhaustiveness checking) since it is rather orthogonal to the work we develop here and is a well-understood topic, given that we do not allow pattern matching over coinductive objects (nor process expressions in general). The rule for constants simply looks up the appropriate type in the signature and verifies the types for subterms accordingly.

The rules for inductive types are the usual equi-recursive rules. The rule for the cofixpoint construct follows the pattern we have previously described. It enforces that the definition is both guarded and that recursion is coregular. Moreover, it restricts the type of the corecursive expression to be a function with a monadic domain type, essentially making a cofixpoint a corecursive process definition with parameters. The monadic introduction rule $\{\}I$ simply

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

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

$$(\nu x)([x \leftrightarrow y] \mid P) \xrightarrow{\tau} P\{y/x\} \qquad x.l; P \xrightarrow{\overline{x.l}} P \qquad x.\mathsf{case}(\overline{l_i \Rightarrow P_i}) \xrightarrow{x.l_i} P_i \qquad u_!\langle c\rangle.P \xrightarrow{(\boldsymbol{\nu} c)u\langle c\rangle} P \qquad (\nu x)(\mathsf{close}_x \mid \mathsf{wait}_x.P) \xrightarrow{\tau} P$$

$$(\nu \overline{a})(\mathsf{spawn}(c \leftarrow \{P\} \leftarrow \overline{b}; \overline{a}; c.Q)) \xrightarrow{\tau} (\nu c)(\nu \overline{a})(P\{\overline{a}/\overline{b}\} \mid Q) \qquad (\nu \overline{w})(\mathsf{spawn}(!u \leftarrow \{P\} \leftarrow !\overline{v}; \overline{w}; u.Q)) \xrightarrow{\tau} (\nu u)(\nu \overline{w})(!u(x).P\{x/u, \overline{w}/\overline{v}\} \mid Q)$$

Figure 4. Labelled Transition System

$$\frac{\Omega, x \vdash \mathsf{Guarded}_f(P; c) \quad \Omega, x \vdash \mathsf{Guarded}_f(M; c)}{\Omega \vdash \mathsf{Guarded}_f(x\langle M\rangle.P; c)} \qquad \frac{\Omega, x \vdash \mathsf{Guarded}_f(P; c) \quad \Omega, x \vdash \mathsf{Guarded}_f(Q; c)}{\Omega \vdash \mathsf{Guarded}_f(\overline{x}\langle a\rangle.(P \mid Q); c)} \qquad \frac{\Omega, x \vdash \mathsf{Guarded}_f(P; c)}{\Omega \vdash \mathsf{Guarded}_f(x(y).P; c)}$$

$$\frac{\Omega, x \vdash \mathsf{Guarded}_f(P_0; c) \quad \ldots \quad \Omega, x \vdash \mathsf{Guarded}_f(P_n; c)}{\Omega \vdash \mathsf{Guarded}_f(x.\mathsf{case}(\overline{l_i \Rightarrow P_i}); c)} \qquad \frac{\Omega, x \vdash \mathsf{Guarded}_f(P; c)}{\Omega \vdash \mathsf{Guarded}_f(x.l; P; c)} \qquad \frac{\Omega, u \vdash \mathsf{Guarded}_f(P; c)}{\Omega \vdash \mathsf{Guarded}_f(!u(x).P; c)}$$

$$\frac{\Omega, u \vdash \mathsf{Guarded}_f(P; c)}{\Omega \vdash \mathsf{Guarded}_f(u_!\langle a\rangle.P; c)} \qquad \frac{c \in \Omega \quad f \notin fn(M) \quad \Omega \vdash \mathsf{Guarded}_f(Q; c)}{\Omega \vdash \mathsf{Guarded}_f(\mathsf{spawn}(f\ M; \overline{d}; x.P); c)} \qquad \frac{\Omega \vdash \mathsf{Guarded}_f(M; c) \quad \Omega \vdash \mathsf{Guarded}_f(Q; c)}{\Omega \vdash \mathsf{Guarded}_f(\mathsf{spawn}(M; \overline{d}; x.Q); c)}$$

$$\frac{\Omega \vdash \mathsf{Guarded}_f(P; c)}{\Omega \vdash \mathsf{Guarded}_f(\mathsf{wait}_x.P; c)} \qquad \frac{}{\Omega \vdash \mathsf{Guarded}_f(\mathsf{close}_x; c)} \qquad \frac{}{\Omega \vdash \mathsf{Guarded}_f([x \leftrightarrow y]; c)}$$

Figure 5. Guardedness Predicate

checks that the process expression P is well-typed in the appropriate contexts, which are generated from the channel list in the type specification by a simple syntactic check ($\mathsf{lin}(a_i)$ denotes the linear channels in the specification and $\mathsf{un}(a_i)$ the persistent ones).

Typing Process Expressions The rules governing process expressions are fundamentally the rules of a proof term assignment for intuitionistic linear logic sequent calculus using a two context region (one linear, another unrestricted) formulation, extended with the monadic elimination rules and rules for an intuitionistic form of conjunction and implication, which correspond to term output and input. The rules are divided into so-called right rules (marked with R), which specify how a process expression can offer a particular session type; left rules (marked with L), which determine how a process may use an ambient session of a given type; and structural rules, which pertain to the nature of the hypothetical judgment such as the id, copy and cut rules. The id rule maps to our channel forwarding construct, whereas the cut rules map to a notion of composition plus hiding. Intuitively, cut composes two processes P and Q that interact along a channel x according to the session A. Since Q expects the full behavior A, the composition must restrict the scope of x to only the two processes, so that no other process may use interfere with the behavior A.

We highlight a few process typing rules: the $\otimes$R rule specifies that to offer $A \otimes B$ along channel c we must output a (fresh) channel y along c and then produce a process P, offering A along y and another process Q which offers B along c. Note that the channels of P and Q are disjoint due to the multiplicative context split. Dually, to use an ambient channel c of type $A \otimes B$ we must input some channel x along c, after which c will be offering behavior B and x the behavior A. The remaining process rules follow similar patterns. We highlight the rules governing replication, where the !R rule requires the linear context to be empty (otherwise replication would violate linearity), the !L rule simply moves the left hand side channel to the appropriate context region and the copy rule which triggers replication. We now focus on the monadic elimination rules $\{\}$L and $\{\}_!$L. The $\{\}$L rule allows us to spawn the underlying process of a monadic term M, offering A along c, in a way that mimics closely the cut rule (in fact, this rule is justified meta-theoretically as reducing to an instance of cut). However, since our monad is contextual, the underlying process of M is not necessarily closed and so can only be executed in the appropriate context, as specified by the monadic type. Thus, for the monadic elimination form to be well-typed we must provide a list $\overline{x}$ of channels which implement the required context. Doing so enables the process P to be able to use c as a session of type A. Just as the $\{\}$L rule corresponds to a cut, the $\{\}_!$L corresponds to cut$^!$. The rule is identical to the $\{\}$L rule, but focuses specifically on monadic objects that offer persistent channels, composing it accordingly with P which will have available the behavior A on a persistent channel u. Note that our operational semantics forbid actions on P until M is fully evaluated to a monadic object and the actual composition takes place.

2.6 Examples

We detail some examples showcasing the expressiveness of our language. To do so in a way that is more pleasant to read, we opt for a less austere notation and some stylistic conventions (as in [23]), which can be easily written in the syntax of the previous sections.

Given that processes are monadic objects, we will use a *do style* notation, with each process action written in the form,

```
x <- action [channel] [parameters]; Q
```

where `Q` is the continuation process (when one is required), `action` denotes the operation performed by the process (along `channel`, with some `parameters`), binding its result to `x` in `Q`, when one exists. We often omit the semi-colon and use new lines to clearly separate the continuations. The constructors with no continuation are forwarding (`fwd x y`) and termination (`close c`). The spawn construct is written as,

$$\frac{\Psi, x{:}\tau \vdash_\Sigma M : \sigma}{\Psi \vdash_\Sigma \lambda x{:}\tau.M : \tau \to \sigma} \to I \quad \frac{\Psi \vdash_\Sigma M : \tau \to \sigma \quad \Psi \vdash_\Sigma N : \tau}{\Psi \vdash_\Sigma M\ N : \sigma} \to E \quad \frac{}{\Psi, x{:}\tau \vdash_\Sigma x{:}\tau}\ \mathsf{var}$$

$$\frac{I \in \Sigma \quad \mathcal{C}(\overline{\tau_i}) \in I \quad \Psi \vdash_\Sigma M_1 : \tau_1 \quad \ldots \quad \Psi \vdash_\Sigma N_n : \tau_n}{\Psi \vdash_\Sigma \mathcal{C}(\overline{M_i}) : I}\ \mathsf{const}$$

$$\frac{I \in \Sigma \quad \Psi \vdash_\Sigma M : I \quad c_0(\overline{\tau_j}) \in I \quad \ldots \quad c_n(\overline{\tau_k}) \in I \quad \Psi, \overline{x_0 : \tau_j} \vdash_\Sigma N_0 : \sigma \quad \ldots \Psi, \overline{x_n : \tau_k} \vdash_\Sigma N_n : \sigma}{\Psi \vdash_\Sigma \mathsf{case}\ M\ \mathsf{of}\ \overline{c_i(\overline{x_i}) \Rightarrow N_i} : \sigma}\ \mathsf{case}$$

$$\frac{\Psi, f{:}\tau \vdash_\Sigma M{:}\tau \quad \mathsf{coreg}_f(M) \quad \mathsf{Guarded}_f(M; c) \quad \tau = \tau_1 \to \{\overline{a{:}A_i} \vdash_\Sigma c{:}\nu X.A\}}{\Psi \vdash_\Sigma \mathsf{cofix}\ f{:}\tau = M{:}\tau}\ \mathsf{cofix}$$

$$\frac{\Delta = \mathsf{lin}(a_i) \quad \Gamma = \mathsf{un}(a_i) \quad \Psi; \Gamma; \Delta \vdash_\Sigma P :: a{:}A}{\Psi \vdash_\Sigma a \leftarrow \{P\} \leftarrow \overline{a_i} : \{\overline{a_i : A_i} \vdash_\Sigma a{:}A\}}\ \{\}I$$

$$\frac{\Psi, f{:}\tau \vdash_\Sigma M{:}\tau \quad \mathsf{Decreasing}_f(M) \quad \tau = \tau_1 \to \tau_2}{\Psi \vdash_\Sigma \mathsf{fix}\ f{:}\tau = M{:}\tau}\ \mathsf{fix} \quad \frac{\Psi \vdash M : \tau\{\nu X.\tau/X\}}{\Psi \vdash M : \nu X.\tau}\ \nu I \quad \frac{\Psi \vdash M : \nu X.\tau}{\Psi \vdash M : \tau\{\nu X.\tau/X\}}\ \nu E$$

$$\frac{\Psi \vdash_\Sigma M : \tau \quad \Psi; \Gamma; \Delta \vdash_\Sigma P :: c{:}A}{\Psi; \Gamma; \Delta \vdash_\Sigma c\langle M\rangle.P :: c{:}\tau \wedge A}\ \wedge R \quad \frac{\Psi, x{:}\tau; \Gamma; \Delta, c{:}A \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta, c{:}\tau \wedge A \vdash_\Sigma c(x).P :: d{:}C}\ \wedge L$$

$$\frac{\Psi, x{:}\tau; \Gamma; \Delta \vdash_\Sigma P :: c{:}A}{\Psi; \Gamma; \Delta \vdash_\Sigma c(x).P :: c{:}\tau \supset A}\ \supset R \quad \frac{\Psi \vdash_\Sigma M : \tau \quad \Psi; \Gamma; \Delta, c{:}A \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta, c{:}\tau \supset A \vdash_\Sigma c\langle M\rangle.P :: d{:}C}\ \supset L$$

$$\frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: y{:}A \quad \Psi; \Gamma; \Delta' \vdash_\Sigma Q :: c{:}B}{\Psi; \Gamma; \Delta, \Delta' \vdash_\Sigma \overline{c}\langle y\rangle.(P \mid Q) :: c{:}A \otimes B}\ \otimes R \quad \frac{\Psi; \Gamma; \Delta, x{:}A, c{:}B \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta, c{:}A \otimes B \vdash_\Sigma c(x).P :: d{:}C}\ \otimes L$$

$$\frac{\Psi; \Gamma; \Delta, x{:}A \vdash_\Sigma P :: c{:}B}{\Psi; \Gamma; \Delta \vdash_\Sigma c(x).P :: c{:}A \multimap B}\ \multimap R \quad \frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: a{:}A \quad \Psi; \Gamma; \Delta', c{:}B \vdash_\Sigma Q :: d{:}C}{\Psi; \Gamma; \Delta, \Delta', c{:}A \multimap B \vdash_\Sigma \overline{c}\langle a\rangle.(P \mid Q) :: d{:}C}\ \multimap L$$

$$\frac{}{\Psi; \Gamma; \cdot \vdash_\Sigma \mathsf{close}_c :: c{:}\mathbf{1}}\ \mathbf{1}R \quad \frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: d{:}D}{\Psi; \Gamma; \Delta, c{:}\mathbf{1} \vdash_\Sigma (\mathsf{wait}_c.P) :: d{:}D}\ \mathbf{1}L$$

$$\frac{\forall i.\Psi; \Gamma; \Delta \vdash_\Sigma P_i : A_i}{\Psi; \Gamma; \Delta \vdash_\Sigma c.\mathsf{case}(\overline{l_i \Rightarrow P_i}) :: c{:} \& \{\overline{l_i : A_i}\}}\ \&R \quad \frac{\Psi; \Gamma; \Delta, c{:}A_i \vdash_\Sigma P :: d{:}D}{\Psi; \Gamma; \Delta, c{:} \& \{\overline{l_i : A_i}\} \vdash_\Sigma c.l_i; P :: d{:}D}\ \&L$$

$$\frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: c{:}A_i}{\Psi; \Gamma; \Delta \vdash_\Sigma c.l_i; P :: c{:} \oplus \{\overline{l_i : A_i}\}}\ \oplus R \quad \frac{\forall i.\Psi; \Gamma; \Delta, c{:}A_i \vdash_\Sigma P_i :: d{:}C}{\Psi; \Gamma; \Delta, c{:} \oplus \{\overline{l_i : A_i}\} \vdash_\Sigma c.\mathsf{case}(\overline{l_i \Rightarrow P_i}) :: d{:}C}\ \oplus L$$

$$\frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: c{:}A\{\mu X.A/X\}}{\Psi; \Gamma; \Delta \vdash_\Sigma P :: c{:}\mu X.A}\ \mu R \quad \frac{\Psi; \Gamma; \Delta, c{:}A\{\mu X.A/X\} \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta, c{:}\mu X.A \vdash_\Sigma P :: d{:}C}\ \mu L$$

$$\frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: c{:}A\{\nu X.A/X\}}{\Psi; \Gamma; \Delta \vdash_\Sigma P :: c{:}\nu X.A}\ \nu R \quad \frac{\Psi; \Gamma; \Delta, c{:}A\{\nu X.A/X\} \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta, c{:}\nu X.A \vdash_\Sigma P :: d{:}C}\ \nu L$$

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \Gamma \supseteq \mathsf{shd}(\overline{a_i{:}A_i}) \quad var(\Delta) \subseteq \overline{x} \quad \Psi \vdash_\Sigma M : \{\overline{a_i{:}A_i} \vdash_\Sigma c{:}A\} \quad \Psi; \Gamma; \Delta', c{:}A \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta, \Delta' \vdash_\Sigma \mathsf{spawn}(M; \overline{x}; c.P) :: d{:}C}\ \{\}L$$

$$\frac{\Gamma \supseteq \overline{!u_i{:}B_i} \quad \overline{x} \subseteq var(\Gamma) \quad \Psi \vdash_\Sigma M : \{\overline{!u_i{:}B_i} \vdash_\Sigma !u{:}A\} \quad \Psi; \Gamma, u{:}A; \Delta \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma; \Delta \vdash_\Sigma \mathsf{spawn}(M; \overline{x}; u.P) :: d{:}C}\ \{\}_!L$$

$$\frac{\Psi; \Gamma; \cdot \vdash_\Sigma P :: d{:}A}{\Psi; \Gamma; \cdot \vdash_\Sigma !u(d).P :: u{:}!A}\ !R \quad \frac{\Psi; \Gamma, u{:}A; \Delta \vdash_\Sigma P\{u/c\} :: d{:}C}{\Psi; \Gamma; \Delta, c{:}!A \vdash_\Sigma P :: d{:}C}\ !L \quad \frac{\Psi; \Gamma, u{:}A; \Delta, c{:}A \vdash_\Sigma P :: d{:}C}{\Psi; \Gamma, u{:}A; \Delta \vdash_\Sigma u_!\langle c\rangle.P :: d{:}C}\ \mathsf{copy}$$

$$\frac{}{\Psi; \Gamma; c{:}A \vdash_\Sigma [c \leftrightarrow d] :: d{:}A}\ \mathsf{id} \quad \frac{\Psi; \Gamma; \Delta \vdash_\Sigma P :: x{:}A \quad \Psi; \Gamma; \Delta', x{:}A \vdash_\Sigma Q :: z{:}C}{\Psi; \Gamma; \Delta, \Delta' \vdash_\Sigma (\nu x)(P \mid Q) :: z{:}C}\ \mathsf{cut}$$

$$\frac{\Psi; \Gamma; \dot{\vdash}_\Sigma P :: x{:}A \quad \Psi; \Gamma, u{:}A; \Delta \vdash_\Sigma Q :: z{:}C}{\Psi; \Gamma; \Delta \vdash_\Sigma (\nu u)(!u(x).P \mid Q) :: z{:}C}\ \mathsf{cut}^!$$

Figure 6. Typing Rules

```
c <- f M <- d;Q
```

encoding the process expression $\mathsf{spawn}(f\ M; d; c.Q)$. When writing corecursive process definitions we use the notation,

```
c <- name x <- d = {P}
```

to represent the term cofix `name` $= \lambda \mathtt{x}.c \leftarrow \{P\} \leftarrow d$ (we omit `<- d` when the contextual region of the monad is empty). Since all well-typed corecursive processes must have only instances of coregular recursion, meaning that after a recursive call we must necessarily have a forwarder to the offered session channel,

```
natstream = nat /\ natstream

f : nat -> {c:natstream}
c <- f M = { _ <- output c M
             d <- f M
             fwd c d}
```

we use the shorthand `c <- f M` for the last two process actions. The process definition above consists of a corecursive process implementing a session of type $\nu X.\mathtt{nat} \wedge X$ along channel c, given a natural number M: an output stream of natural numbers. We represent the choice and alternative behavior session types, $\&\{l_1 \Rightarrow A_1, l_2 \Rightarrow A_2\}$ and $\oplus\{l_1 \Rightarrow A_1, l_2 \Rightarrow A_2\}$, as `Choice{l1:A1 , l2:A2}` and `Or{l1:A1 , l2:A2}`, respectively. Selection of a label l on c is written `c.l`. The corresponding `case` construct for a channel c offering a choice type (or using an alternative behavior type) is written as:

```
case c of l1 => P1 , ... , ln => Pn
```

We use `output x {d <- P};Q` as the process construct that corresponds to session channel output, for which we write the type as `A*B`. P denotes the process offering A along the fresh channel d and Q the continuation, offering B. Finally, we use `x <- copy !u` to denote the invocation of the server on channel `!u`, bound to `x`.

We detail the following extended examples: First, we present a process network implementation of a binary bitwise counter, where each node in the network implements a single bit of an arbitrary length bit string, encoded using a little endian representation, that can only communicate with its neighbor processes. This example showcases a rather general design pattern that we use throughout our examples and is an alternative implementation of the bit counter of [23], with the key difference that this definition has coregular recursion, whereas the one in [23] does not. Our second example highlights the difference between recursion and replication. Our final example is a simple implementation of a process network in a tree topology, allowing for parallelized implementations of map and reduce operations on data.

2.6.1 Little Endian Bit Counter

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

```
Counter = Choice(val:int /\ Counter,
                 inc:Counter,
                 halt:1}
```

The type `Counter` is a coinductive session type because we do not want to impose an upper bound on the counter value, which would be necessary if it were an inductive type.

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

```
CImpl = Choice{val:int => int /\ CImpl,
               inc:Or{carry:CImpl, done:CImpl},
               halt:1}
```

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

```
Node : int -> {x:CImpl |- n:CImpl}
n <- Node b <- x =
{ case n of
   val  => x.val
           m <- input n
           _ <- output x (2*m+b)
           m <- input x
           _ <- output n m
           n <- Node b <- x
   inc  => x.inc
           case x of
            carry => if (b=1) then
                       n.carry
                       n <- Node 0 <- x
                     else
                       n.done
                       n <- Node 1 <- x
             done => n <- Node b <- x
   halt => x.halt
           wait x
           close n}
```

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

```
Coord : unit -> {x:CImpl |- z:Counter}
z <- Coord () <- x =
{ case z of
   val  => x.val
           _ <- output x 0
           n <- input x
           _ <- output z n
           z <- Coord () <- x
   inc  => x.inc
           case x of
            carry => n <- Node 1 <- x
                     z <- Coord () <- n
            done  => z <- Coord () <- x
   halt => x.halt
```

```
          wait x
          close z}
```

The only remaining component is the representation of the empty bit string `epsilon`, which will be a closed process expression of type `x:Impl`, implement the "base cases" of the protocol. For polling the value, it just ping pongs the received value. For incrementing, it just signals the `carry` message. We can then put the `Counter` system together by composing `epsilon` and `Coord`.

```
epsilon:unit -> {x:CImpl}
x <- epsilon () =
{ case x of
   val  => n <- input x
           _ <- output x n
           x <- epsilon ()
   inc  => x.carry
           x <- epsilon ()
   halt => close x}

z <- Counter =
{ x <- epsilon ()
  z <- Coord () <- x}
```

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

2.6.2 Replication and Recursion

In this example we highlight the differences between replication and recursion by specifying an App Store-like replicated service. The store offers a variety of different applications, themselves concurrent programs that rely on a remote API. Each store instance sends its client the application of choice as a monadic object and then terminates. The client can then run the application (in this scenario, all applications rely on a remote `API`, but the `weather` application also relies on a `GPS` module). The session types for the app. store and the weather app. are as follows:

```
Weather = Or{sun:Weather ,
             rain:Weather , cloudy:Weather}
AppStore = !Choice{weather: payInfo =>
                   {d:API,g:GPS |- c:Weather} /\ 1,
                   travel: payInfo =>
                       {d:API |- c:Travel} /\ 1,
                   game: payInfo =>
                       {d:API |- c:Game} /\ 1}
```

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

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

```
ActivateGPS : unit -> {g:GPS}
WeatherClient : unit ->
                  {!u:AppStore,d:API |- c:Weather}
c <- WeatherClient () <- !u,d =
{ s <- copy !u
  _ <- s.weather
  w <- input s
  _ <- wait s
  g <- ActivateGPS ()
  c <- w <- d,g }
```

2.6.3 Computation Trees

We have seen how to represent networks of processes in a chain-like topology in the example of Section 2.6.1. An obvious question is if our framework is general enough to encode more sophisticated topologies. Here, we show that this is indeed the case by representing a network of processes with a tree topology, where each process holds a piece of data (for simplicity, this data is an integer).

```
PTree = Choice{go:Or{some:int /\ (PTree * PTree)
                     none:1}
               halt:1}
```

In this particular example, the type `PTree` denotes a session which offers two operations: `go` gives us access to the process tree node, which can either either be a leaf (`none`) or a node (`some`); whereas `halt` terminates the entire process tree outright.

```
map : (int -> int) -> {y:PTree |- x:PTree}
x <- map f <- y = {
 case x of
  go => y.go
        case y of
         some => x.some
                 v <- input y
                 z <- input y
                 _ <- output x (f v)
                 _ <- output x {d <- map f <- z}
                 x <- map f <- y
          none => x.none
                  wait y
                  close x
  halt => y.halt
          wait y
          close x }
```

Using this process tree network, we can easily encode a parallel version of a `map` on the tree, where some function `f` is applied to each node in a given network (accessed through channel `y`). The key point is the structure of the recursive calls, each done on a separate, independent, computation branch.

We can also encode a parallel fold operation on the process network, that reduces the tree topology to a single value (of type `b`) and outputs this value. We define `fold` *inductively* (using a fixpoint, not a cofixpoint), by using an additional natural number argument which represents the maximum depth of the process tree that will be explored. Thus our fold operation takes a natural number `n` which is the decreasing argument, the reduce function `f` and the initial reduction value `z`, and will produce a process that given a process tree along `x` will output the resulting reduction along `c` and terminate.

```
fold : nat -> (int -> b -> b -> b)
                 -> b -> {x:PTree |- c:b /\ 1}
c <- fold n f z <- x =
 case n of
   s(m) => { x.go
             case x of
              some => v <- input x
                      y <- input x
                      c1 <- fold m f z <- y
                      c2 <- fold m f z <- x
```

```
                    r1 <- input c1
                    r2 <- input c2
                    _ <- output c (f v r1 r2)
                    close c
             none => _ <- output c z
                    wait x
                    close c}
  0    => { x.halt
            _ <- output c z
            wait x
            close c }
```

We define `fold` inductively since it allows us to maximize the parallelism, by performing the two recursive calls in parallel, which is not an instance of coregular recursion.

2.7 Type Safety

Type preservation for our language follows from a simulation between process reductions and proof conversions that arise naturally in the proof theory of linear logic, again ensuring our interpretation is not arbitrary but deeply connected to the dynamics of proofs. The proof itself follows very closely that of [5], requiring a series of reduction lemmas, relating process reductions with parallel composition through the cut rule. One such representative lemma is given below (Lemma 2.3).

Lemma 2.3. *Assume*

(a) $\Psi;\Gamma;\Delta_1 \vdash P :: x{:}A \otimes B$ *with* $P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P'$

(b) $\Psi;\Gamma;\Delta_2, x{:}A \otimes B \vdash Q :: z{:}C$ *with* $Q \xrightarrow{x(y)} Q'$

Then: $\Psi;\Gamma;\Delta_1,\Delta_2 \vdash (\boldsymbol{\nu} x)(P' \mid Q') :: z{:}C$

Theorem 2.4 (Type Preservation)**.**

- *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *and* $P \longrightarrow Q$ *then* $\Psi;\Gamma;\Delta \vdash Q :: z{:}A$.
- *If* $\Psi \vdash M{:}\tau$ *and* $M \longrightarrow N$ *then* $\Psi \vdash N{:}\tau$.

Traditionally, session typed languages either do not satisfy progress, or require sophisticated technical machinery to establish progress through causal dependency analysis [7]. In our logically grounded setting, we not only have global progress but we can also carry the proof through in a straightforward way. The proof is an extension of that in [5], requiring a series of inversion lemmas and the following notion of *live* process. For any process expression P, $live(P)$ if and only if $P \equiv (\nu\overline{n})(\pi.Q \mid R)$, for some process R, a sequence of names $\overline{n}$, and a *non-replicated* guarded process $\pi.Q$ or $P \equiv (\nu\overline{n})(\mathsf{spawn}(M;\overline{n};c.Q) \mid R)$. Liveness essentially excludes closed sessions and residual replicated server processes.

Theorem 2.5 (Progress)**.**

- *If* $\cdot;\Gamma;\Delta \vdash P :: x{:}A$ *and* $live(P)$ *then* $\exists Q$ *s.t. either* $P \to Q$ *or* $P \xrightarrow{\alpha} Q$.
- $\cdot \vdash M{:}\tau$ *then either* $M \longrightarrow N$ *or* M *is a value.*

Our statement of progress for process expressions is slightly different from that of [5] (ours is closer to what they call contextual progress), which defines progress as the ability of *live* processes of type $x{:}\mathbf{1}$ to always have an internal action (no labelled actions are available due to the type). In the presence of coinductive processes, restricting the type A to $\mathbf{1}$ is not be general enough, since we cannot in general type $(\nu y)(P \mid Q) :: x{:}\mathbf{1}$ with $P :: y{:}\nu X.B$, for some B, since no Q can fully use the behavior of P if it is infinite.

3. Logical Relations

We now build up the necessary technical machinery to show that all well-typed programs in our language are non-divergent. We do so using a logical relations argument, combining logical relations for λ-calculi with the linear logical relations of [6] for process expressions, extended to the inductive and coinductive setting. As usual, we define a logical predicate that implies non-divergence and show that all well-typed terms (and process expressions) are in the predicate at the appropriate types. This means that not only do our session-typed programs adhere strictly to the specified session-behavior in a deadlock free way, but they do so without engaging into an infinite sequence of computations, despite the fact they may encode infinite behavior.

We begin by defining a notion of reducibility candidate on typed λ-terms and process expressions. These consist of sets of terminating terms (resp. processes) satisfying natural closure conditions.

Definition 3.1 (Reducibility Candidates - Terms)**.** *Given a type τ, a reducibility candidate at type τ, written $\mathcal{R}[\tau]$ is a set of terms with the following properties:*

1. *If $M \in \mathcal{R}[\tau]$ then $\cdot \vdash M : \tau$.*
2. *If $M \in \mathcal{R}[\tau]$ then $M \Downarrow$.*
3. *If $M \in \mathcal{R}[\tau]$ and $M \longrightarrow M'$ then $M' \in \mathcal{R}[\tau]$.*
4. *If $M \in \mathcal{R}[\tau]$ and $M_0 \longrightarrow M$ then $M_0 \in \mathcal{R}[\tau]$.*

Definition 3.2 (Reducibility Candidates - Process Expressions)**.** *Given a type A and a name z, a reducibility candidate at $z{:}A$, written $\mathcal{R}[z{:}A]$ is a set of process expressions satisfying the following properties:*

1. *If $P \in \mathcal{R}[z{:}A]$ then $\cdot;\cdot;\cdot \vdash P :: z{:}A$.*
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. Similarly for candidates for λ-terms, for which we use $\mathbb{R}[\tau]$.

The logical predicate uses the following extension to structural congruence with the so-called *sharpened replication axioms* [20].

Definition 3.3. *We write $\equiv_!$ for the least congruence relation on process expressions which results from extending structural congruence $\equiv$ (Fig. 2) 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. $(\nu u)(!u(y).P \mid (\nu v)(!v(z).Q \mid R)) \equiv_! (\nu v)((!v(z).(\nu u)(!u(y).P \mid Q)) \mid (\nu u)(!u(y).P \mid R))$
3. $(\nu u)(!u(y).Q \mid P) \equiv_! P \quad$ *if* $u \notin fn(P)$

Intuitively, $\equiv_!$ allows us to properly "split" processes: axioms (1) and (2) represent the distribution of shared servers among processes, while (3) captures garbage collection of shared servers which can no longer be invoked by any process. We note that $\equiv_!$ expresses sound behavioral equivalences in our typed setting.

Logical Predicate We define a logical predicate on terms and processes by induction on types and the size of typing contexts in the case of processes. Intuitively, the predicate captures the computational behavior of non-divergent terms and process expressions, as defined by their typing. The definitions are mutually inductive but we separate them into the three cases for ease of presentation and readability. For open well-typed terms $\Psi \vdash M{:}\tau$, we take a substitution ψ that is *compatible* with Ψ, assigning to variables recorded in Ψ closed terms in the logical predicate at the appropriate type, and so we can restrict ourselves to the predicate on closed terms by applying ψ to M, written $\hat{\psi}(M)$. We note that the definition is well-founded since either the type or the context gets smaller.

Definition 3.4 (Logical Predicate - Terms)**.** *For any type τ we define $\mathcal{L}^{\omega}[\tau]$ as the set of all terms M such that $M\Downarrow$ and $\vdash M : \tau$,*

where ω is a mapping of type variables to reducibility candidates, satisfying the rules of Fig. 7.

The interesting rules in Def. 3.4 are the ones defining the interpretation of inductive types $\mu X.\tau$ and of the monadic type. The former consists of the intersection of all candidates ψ at the appropriate type containing the terms in the interpretation of τ in which X is mapped to ψ itself, essentially defining a (meta-level) induction principle, justified by a fixpoint theorem which we shall establish later; the latter requires that a term in the predicate at monadic type reduce in a finite number of steps to a monadic value, with some underlying process expression P that is required to be in the logical predicate at its appropriate session typing.

Definition 3.5 (Logical Predicate - Open Process Expressions)**.** *For any typing $\Psi;\Gamma;\Delta \vdash T$ with a non-empty left hand side environment, we define $\mathcal{L}^{\omega}[\Psi;\Gamma;\Delta \vdash T]$, where ω is a mapping from type variables to reducibility candidates, as the set of process expressions inductively defined by the rules of Fig. 8.*

The definition of the logical predicate for open processes inductively composes the process with the appropriate witnesses in the logical predicate at the types specified in the three contexts.

Definition 3.6 (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 $\cdot;\cdot;\cdot \vdash P :: T$ satisfying the rules of Fig. 9.*

The definition of the predicate for closed process expressions is an extension of the logical predicate of [18]. The cases for the propositional connectives of linear logic, interpreted as session types, capture the observable behavior of the process as specified by its type in a very precise way, ignoring internal actions (this is also known as weak observations). For instance, the predicate at type $z{:}A \multimap B$ contains all processes that can offer an input along z of a fresh channel y (after an arbitrary but finite number of τ-actions), such that the resulting continuation when composed with a process in the logical predicate at type $y{:}A$ will be in the predicate at type $z{:}B$. The remaining connectives follow an identical pattern, maintaining the invariant of only considering closed processes. For the term input type, we require that for any term in the logical predicate at the appropriate type τ, the process must be able to offer an input action such that the process resulting from inputting such a term will itself be in the predicate at the continuation type $z{:}A$. Dually, for the term output type we require that the process have an output capability of a term in the predicate at type τ, resulting in a process in the logical predicate at the continuation type $z{:}A$.

The interpretation of inductive session types $\mu X.A$ is the intersection of all process reducibility candidates Ψ at the appropriate inductive type containing the interpretation of $z{:}A$, with X mapped to Ψ. Notice how Ψ is not bound to z outright, which is the key difference from the interpretation of inductive types. This is related to the fact that session types are offered along specific channels, which we do not know *a priori* (e.g. a corecursive call will always spawn a process on a fresh channel), therefore we must delay the particular choice of the channel until the appropriate point (which is the recursive type variable). The interpretation of coinductive sessions $\nu X.A$ is, in some sense, dual. We interpret coinductive session types as the union of reducibility candidates ψ at the appropriate coinductive type that are in the interpretation of the session type A, when X is mapped to ψ itself. Just as the interpretation of inductive types enables a form of meta-level induction through a (pre) fixpoint, the interpretation of coinductive types enables a form of meta-level coinduction through a (post) fixpoint.

Having defined the logical predicate, we need to establish certain soundness conditions required to show the so-called fundamental lemma (i.e. all well-typed terms and process expressions are in the logical predicate). We begin by establishing that the logical predicates satisfy the reducibility candidate criteria of Definitions 3.1 and 3.2.

Theorem 3.7 (Logical Predicate is a Reducibility Candidate)**.**

- $\mathcal{L}[\tau] \subseteq \mathcal{R}[\tau]$.
- $\mathcal{L}[z{:}A] \subseteq \mathcal{R}[z{:}A]$

The interpretations of inductive and coinductive types are effectively least and greatest fixpoints of a monotone operator. The operator in question consists of a mapping (indexed by an open type) between sets of reducibility candidates, taking a set of reducibility candidates s to the interpretation of the open type, where the type variable is mapped to s. We define two such operators, one for types in the λ-calculus (Def. 3.8) and another for session types (Def. 3.9), showing that the interpretation of inductive and coinductive types is a least (resp. greatest) fixpoint of the respective operators.

Definition 3.8. *Let $\mu X.\tau$ be a strictly positive type. We define the operator $\phi_\tau : \mathbb{R} \to \mathbb{R}$ as:*

$$\phi_\tau(s) \triangleq \mathcal{L}^{\omega[X\mapsto s]}[\tau]$$

Definition 3.9. *Let $\mu X.A$ or $\nu X.A$ be a strictly positive session type. Let $\phi_A : \mathbb{R} \to \mathbb{R}$ defined as:*

$$\phi_A(s) \triangleq \mathcal{L}^{\omega[X\mapsto s]}[z{:}A]$$

To establish the fixpoint properties we need to first show that the operators are monotonic. Recall that an operator ϕ is monotonic if $s \subseteq s'$ implies $\phi(s) \subseteq \phi(s')$, for any s, s'.

Lemma 3.10 (Monotonicity of ϕ_τ and ϕ_A)**.** *The operators ϕ_τ and ϕ_A are monotonic.*

By making key use of Lemma 3.10 we can establish that indeed, the interpretation of inductive and coinductive types satisfies the appropriate fixpoint properties (proofs are available in the supplementary materials).

Theorem 3.11 (Least Fixpoint - Inductive Types)**.**
$\mathcal{L}^{\omega}[\mu X.\tau]$ is a least fixpoint of ϕ_τ.

Theorem 3.12 (Least Fixpoint - Inductive Session Types)**.**
$\mathcal{L}^{\omega}[z{:}\mu X.A]$ is a least fixpoint of ϕ_A.

Theorem 3.13 (Greatest Fixpoint - Coinductive Session Types)**.**
$\mathcal{L}^{\omega}[z{:}\nu X.A]$ is a greatest fixpoint of ϕ_A.

We need one final piece of machinery in order to show the fundamental lemma, which relates syntactically unfolded inductive (and coinductive types) to the interpretation of the open type when the type variable is mapped to the appropriate logical predicate, establishing a form of compositionality.

Theorem 3.14 (Compositionality)**.** *Let $\mu X.A$, $\nu X.B$ and $\mu X.\tau$ be well-formed types. The following properties hold:*

- $P \in \mathcal{L}^{\omega}[z{:}A\{\mu X.A/X\}]$ *iff* $P \in \mathcal{L}^{\omega[X\mapsto\mathcal{L}^{\omega}[-:\mu X.A]]}[z{:}A]$
- $P \in \mathcal{L}^{\omega}[z{:}A\{\nu X.A/X\}]$ *iff* $P \in \mathcal{L}^{\omega[X\mapsto\mathcal{L}^{\omega}[-:\nu X.A]]}[z{:}A]$
- $M \in \mathcal{L}^{\omega}[\tau\{\mu X.\tau/X\}]$ *iff* $M \in \mathcal{L}^{\omega[X\mapsto\mathcal{L}^{\omega}[\mu X.\tau]]}[\tau]$

We are now in a position to show the fundamental lemma, that all well-typed terms and process expressions are in the logical predicate, from which follows that all well-typed programs in our language are non-divergent.

Theorem 3.15 (Fundamental Lemma)**.** *If $\Psi \vdash M{:}\tau$ and $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ then for any mapping ψ s.t. $x{:}\tau_0 \in \Psi$ iff $\psi(x) \subseteq \mathcal{L}[\tau_0]$:*

1. $\hat{\psi}(M) \subseteq \mathcal{L}[\tau]$
2. $P \subseteq \mathcal{L}[\Psi;\Gamma;\Delta \vdash z{:}A]$

The proof proceeds by mutual induction on the given typing derivation, making key use of Theorem 3.14 in the cases for the

$$
\begin{array}{lcl}
\mathcal{L}^{\omega}[\tau \to \sigma] & \triangleq & \{M \mid M \Longrightarrow \lambda x{:}\tau.M' \text{ and } \forall N \in \mathcal{L}^{\omega}[\tau].M'\{N/x\} \in \mathcal{L}^{\omega}[\sigma]\} \\
\mathcal{L}^{\omega}[\mu X.\tau] & \triangleq & M \in \bigcap\{\psi \in \mathbb{R}[\mu X.\tau] \mid \mathcal{L}^{\omega[X \mapsto \psi]}[\tau] \subseteq \psi\} \\
\mathcal{L}^{\omega}[X] & \triangleq & M \in \omega(X) \\
\mathcal{L}^{\omega}[\{\overline{a{:}A_i} \vdash c{:}A\}] & \triangleq & \{M \mid M \Longrightarrow a \leftarrow \{P\} \leftarrow \overline{a} \text{ and } P \in \mathcal{L}^{\omega}[\mathsf{shd}(\overline{a{:}A_i}); \mathsf{lin}(\overline{a{:}A_i}) \vdash c{:}A]\}
\end{array}
$$

Figure 7. Logical Predicate - Terms (See Definition 3.4)

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

Figure 8. Logical Predicate - Open Process Expressions (See Definition 3.5)

$$
\begin{array}{lcl}
\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{:}\mu X.A] & \triangleq & \bigcap\{\Psi \in \mathbb{R}[-{:}\mu X.A] \mid \mathcal{L}^{\omega[X \mapsto \Psi]}[z{:}A] \subseteq \Psi\} \\
\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_! \mathsf{close}_z\} \\
\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 \textstyle\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 \textstyle\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}^{\omega}[\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}^{\omega}[\tau] \wedge P \overset{z(M)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A]\}
\end{array}
$$

Figure 9. Logical Predicate - Closed Process Expressions (See Definition 3.6)

rules for inductive and coinductive types. We only sketch the proof (the details can be found in the companion material). The technical aspects of the proof for the fragment of the language that is not specific to inductive and coinductive definitions follow that of [18] for the process expression setting. Intuitively, we define a notion of logical representative of the dependencies that are denoted by the typing contexts, enabling us to compose an open process with these representatives and restrict our reasoning to closed processes.

The more challenging aspects of the proof pertain to the cases for fixpoint and cofixpoint constructs, which allow us to define inductive and coinductive processes through the contextual monad. For the fixpoint construct, we exploit the fact that all instances of the recursive occurrence of the fixpoint variable are in applications to structurally smaller arguments in some inductively defined structure, allowing for an inner induction on the size of the argument to discharge the required proof obligation. This is justified by the fact that fixpoints, even when defining recursive processes, are always "counting down" on some inductively defined structure. For the cofixpoint construct, we take advantage of the fact that a corecursive definition is essentially a coinductive session-typed process P with (λ-term) parameters. Since the interpretation of coinductive types $\nu X.A$ is a greatest fixpoint (Theorem 3.13), we close the proof through a coinductive argument: we generate a set of processes $\mathcal{C}_P$, containing P, 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 because we have established that $\mathcal{L}[c{:}\nu X.A]$ is the largest such set. Showing that $\mathcal{C}_P$ satisfies this condition relies crucially on coregular recursion, which enforces that the unfolding of the coinductive type is offered by the recursive call, followed by a forwarder, reducing to a process in $\mathcal{C}_P$ (roughly, P itself).

Corollary 3.16 (Non-Divergence)**.**

- *If* $\Psi \vdash M{:}\tau$ *then* $M \Downarrow$
- *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then* $P \Downarrow$

Combining type safety and termination we show that typing enables strong guarantees on concurrent programs in the presence of rich language features such as mobile code, and potentially infinite stateful behavior. A well-typed program will always be able to fulfill 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.

4. Related Work

From the perspective of more traditional work on session typed languages, our language has some similarities to that of [17], which presents two session typed calculi for higher-order processes. The key differences are the absence of recursive definitions (even general recursive types), and the lack of a logical foundation, which greatly simplifies the metatheory and allows us to establish progress, which they do not. There is also no similar non-divergence result for their system, nor are we aware of any such result for higher-order session typed languages. For the first-order setting, [25] establishes a strong normalization result for action types, which are similar to session types, but the theory itself is substantially more complex due to the lack of a logical foundation.

The particular issues related to inductive and coinductive session types have to the best of our knowledge been overlooked in the literature, which typically deals with general recursive types and definitions [8, 13, 14] and so is inherently non-divergent in na-

ture. In this regard our work is closer to that on termination for λ-calculi with inductive and coinductive types [1]. For instance, [16] develops a strong normalization result for the second-order λ-calculus with explicit inductive and coinductive types using logical relations. Besides the obvious differences due to the language itself, their model of the λ-calculus is untyped and moreover the terms of inductive and coinductive type are generated using recursor and corecursor constants, following the standard encoding of inductive and coinductive types in System F, whereas we use fixpoint definitions subject to syntactic restrictions. These restrictions are related to those developed in [9, 10] for the Coq proof assistant, but with some fundamental differences (discussed in Section 2.4). Strong normalization proofs for such definitions exist in various forms, mostly in work related to proving consistency of the Calculus of (Co)Inductive Constructions [4, 15], the underlying theory of Coq. The work on type-based methods for ensuring termination has been generalized to include inductive and coinductive definitions in the style of Coq [11] (although the proof of strong normalization is substantially more complex due to the expressive power of CIC and type annotations). It is not clear if it is possible to adapt these type-based methods to our session typed setting since type annotations are a measure of the size of "values", which do not apply to processes. Similar difficulties arise when considering copatterns [2, 3] that seem to be inherently tied to the term structure.

From the logic based approach to session typed programming, our work is related to [24], 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 [8], that is inherently linearly typed, and does not consider primitive inductive or coinductive sessions. We opt for a stratified system using our contextual monad, isolating linearity to the process expression layer, and work in an intuitionistic framework. The latter is particularly important given our goals of developing a full dependent type theory. Moreover, the stratification using the contextual monad is not a purely stylistic design choice, since it also plays a role in making the future transition to dependent types simpler by avoiding the need to have coinduction in the functional theory.

5. Concluding Remarks

We have presented a theory of inductive and coinductive definitions for session-typed higher order concurrent programs, with a sound logic foundation based on intuitionistic linear logic, establishing strong guarantees of deadlock freedom, session fidelity and compositional non-divergence by typing.

The work set forth in this paper is a fundamental stepping stone towards the development of a dependent type theory rich enough to allow us to express and reason about concurrent session-based programs. These programs are in most cases inductive or coinductive in nature, and are typically studied using coinductive proof techniques. In type theory, coinduction can raise some issues since it is difficult to obtain both strong normalization and subject reduction with dependent pattern matching and the lazy evaluation of codata. Since in this work we establish process expressions as coinductive objects, in a future move to a dependently typed setting we may potentially be able to use them as witnesses to coinductive proofs, hopefully avoiding the some of the challenges of coinductive types in type theory since we do not require restricted reduction rules for cofixpoints to preserve non-divergence, nor allow pattern matching on processes. The final step that we need to develop is a definitional equality of coinductive process expressions that is expressive enough for our needs. Naturally, this equality must be sound wrt an extensional typed equivalence for processes, such as that of [6, 18].

References

[1] A. Abel. Strong normalization and equi-(co)inductive types. In *Typed Lambda Calculi and Applications*, pages 8–22. 2007.

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *LICS*, 2013. to appear.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *POPL'13*, pages 27–38, 2013.

[4] T. Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

[5] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, pages 222–236, 2010.

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

[7] M. Dezani-Ciancaglini, U. de' Liguoro, and N. Yoshida. On Progress for Structured Communications. In *TGC'07*, pages 257–275, 2008.

[8] S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Programming*, 20(1):19–50, 2010.

[9] E. Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94*, pages 39–59.

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

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

[12] K. Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523, 1993.

[13] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, pages 122–138, 1998.

[14] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, 2008.

[15] P.-A. Mellis and B. Werner. A generic normalisation proof for pure type systems. In *Types for Proofs and Programs*, pages 254–276, 1998.

[16] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(12):159 – 172, 1991.

[17] D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA07*, pages 321–335, 2007.

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

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

[20] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[21] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4pl2*, 2013. URL `http://coq.inria.fr`.

[22] B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP'11*, pages 161–172, 2011.

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

[24] P. Wadler. Propositions as sessions. In *ICFP'12*, pages 273–286, 2012.

[25] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi -calculus. *Inf. Comput.*, 191(2):145–202, 2004.