The Least Must Speak with the Greatest

J. Garrett Morris, Sam Lindley, and Philip Wadler

The University of Edinburgh
{Garrett.Morris,Sam.Lindley,Philip.Wadler}@ed.ac.uk

Abstract. We extend a propositions-as-types correspondence between linear logic and session types to include recursive sessions. Our extension takes least and greatest fixed points as dual, an idea well-known to theorists, but which has not previously appeared in the treatment of recursive sessions. We preserve the freedom from races, deadlock, and livelock that is a hallmark of the propositions-as-types approach, and avoid a problem with preservation of duality that appears in the standard treatment of recursive session types. We treat two systems, one based on classical linear logic, derived from work of Caires and Pfenning and of Wadler, and one based on a linear functional language, derived from work of Gay and Vasconcelos. Our treatment of recursion is inspired by Baelde.

1 Introduction

Previous work on session types suffers from a mismatch. Session types capture communication protocols. The two ends of a channel must be *dual*, which ensures that when one end of a channel performs output the other performs input, and when one end offers a choice of actions the other selects from these alternatives. Session types are of limited use without *recursion*, which permits defining sessions of arbitrary size. Recursion traditionally comes in two dual forms, least fixed point and greatest fixed point. However, in previous work on session types recursion is introduced in such a way that both ends of a channel are treated as least fixed points, a mismatch with the treatment of the other connectives. Here, we introduce a notion of recursion for session types that restores the traditional notion that least fixed points are dual to greatest fixed points.

Honda (1993) introduced session types as a type system for process calculi, and Gay and Vasconcelos (2010) reformulated session types in the context of a functional language with linear types. Caires and Pfenning (2010) uncovered a propositions-as-types correspondence between session types and intuitionistic linear logic, and Wadler (2012) adapted the correspondence to classical linear logic. Wadler defines two systems, a process calculus CP (inspired by Caires and Pfenning) and a linear functional language GV (inspired by Gay and Vasconcelos), and presents a type-preserving translation from GV to CP. Subsequently, Lindley and Morris (2014) extended GV to support translation in the other direction, giving type-preserving translations not only from GV to CP but also from CP to GV. A hallmark of the propositions-as-types approach is that it yields systems guaranteed free of races and deadlock; both CP and GV benefit from these properties. Toninho et al. (2013, 2014) also treat recursion in systems inspired by a propositions-as-types correspondence, but they have no notion of duality for least and greatest fixed points.

There is much previous work that supports recursive session types, starting with Honda et al. (1998), and including Yoshida and Vasconcelos (2007) and Demangeon and Honda (2011), all of which suffer from the mismatch mentioned above. Here, drawing on work of Baelde (2012), we extend both CP and GV with recursive session types, taking least and greatest fixed points as duals. We show that the type-preserving translations extend to the new systems, and that the new systems are still guaranteed free of races, deadlock, and livelock.

Bono and Padovani (2012) and Bernardi and Hennessy (2013) independently observed that the standard treatment of recursive session types fails to preserve duality in the case that the recursive variable appears within a carried type. Their solution relies on an unusual form of substitution that applies only to the carried types within a session type, while ours avoids the problem while using only standard forms of substitution.

This paper makes the following contributions.

- Section 2 extends GV to support recursion, yielding μGV; and demonstrates its power with a series of examples of streams and a calculator. In μGV, unlike in previous session type systems, least fixed points are dual to greatest fixed points and duality is preserved in all cases.
- Section 3 extends CP to support recursion, yielding μCP, and presents a translation of some of the same examples. We show μCP is guaranteed free of races, deadlock, and livelock.
- Section 4 presents type-preserving translations from μGV into μCP and inversely. The translation guarantees that GV also is free of races, deadlock, and livelock. As a technical aid, we also present a type-preserving translation from μGV into a subset of itself, similar to one in Lindley and Morris (2014), showing how function and product types can be represented by session types.

Section 5 surveys related work and Section 6 concludes.

2 A Session-Typed Functional Language

In this section, we present μGV, a simple functional language with session types patterned on the language of Gay and Vasconcelos (2010) (which we call LAST) and Wadler's language GV. Throughout the paper we highlight the novel parts of μGV and μCP by shading them in gray.

2.1 Types

Types in μGV are given by the following grammar:

Types	T, U, V	$::= S \mid T \otimes U \mid T \multimap U \mid T \to U$
Session Types	S	$::= !T.S \mid ?T.S \mid \mathsf{end}_! \mid \mathsf{end}_? \mid \oplus\{l_i : S_i\}_i \mid \&\{l_i : S_i\}_i \mid \sharp S \mid \flat S$
		$\mid X \mid \overline{X} \mid \mu G \mid \nu G$
Type Operators	G	$::= X.S$

The types comprise session types, linear pairs ($T \otimes U$), and both linear ($T \multimap U$) and unlimited ($T \to U$) functions. Session types include input ($?T.S$), output

$(!T.S)$, selection $(\oplus\{l_i : S_i\}_i)$, choice $(\&\{l_i : S_i\}_i)$, and closed channels ($\mathsf{end}_?$ and $\mathsf{end}_!$). There are two variations on the closed channel ($\mathsf{end}_?$) and ($\mathsf{end}_!$); these arise from our interpretation of session types in classical linear logic, where there is no self-dual proposition corresponding to closed channels. We include a notion of replicated sessions, corresponding to exponentials in linear logic: a channel of type $\sharp S$ is a service, offering any number of channels of type S; a channel of type $\flat S$ is the server providing such channels. For simplicity, we omit polymorphism as it is an orthogonal concern. We include session variables (X) and their duals $(\overline{X})$, and types corresponding to bounded recursion $(\mu X.S)$ and unbounded corecursion $(\nu X.S)$. If G is an operator $X.S$, then $G(S')$ denotes the standard capture-avoiding substitution of S' for X in S, which we write as $S[S'/X]$.

Each type T is either linear $(lin(T))$ or unlimited $(un(T))$. All types are linear except unlimited functions $T \to U$, replicated channels $\sharp S$, and closed input channels $\mathsf{end}_?$. We extend the standard notion of duality to recursive and corecursive session types:

$$\begin{array}{llllll} \overline{!T.S} = ?T.\overline{S} & \overline{\mathsf{end}_?} = \mathsf{end}_! & \overline{\oplus\{l_i : S_i\}_i} = \&\{l_i : \overline{S_i}\}_i & \overline{\sharp S} = \flat\overline{S} & \overline{\mu G} = \nu\overline{G} \\ \overline{?T.S} = !T.\overline{S} & \overline{\mathsf{end}_!} = \mathsf{end}_? & \overline{\&\{l_i : S_i\}_i} = \oplus\{l_i : \overline{S_i}\}_i & \overline{\flat S} = \sharp\overline{S} & \overline{\nu G} = \mu\overline{G} \end{array}$$

where $\overline{\overline{X}} = X$ and we define the dual of an operator $G = X.S$ as $\overline{G} = X.\overline{G(\overline{X})}$. Observe that $\overline{G(S)} = \overline{G}(\overline{S})$. Unlike many notions of duality for session types, our definition preserves duality when recursive session types are unrolled, even when the recursion occurs in the carried types. For example, consider the operator $G = X.?X.\mathsf{end}_?$, its dual $\overline{G} = X.!\overline{X}.\mathsf{end}_!$, and the dual session types μG and $\nu\overline{G}$. Unrolling μG yields $G(\mu G) = ?\mu G.\mathsf{end}_?$; unrolling $\nu\overline{G}$ yields $\overline{G}(\nu\overline{G}) = !\overline{\nu\overline{G}}.\mathsf{end}_! = !\mu G.\mathsf{end}_!$, which is the dual of $?\mu G.\mathsf{end}_?$.

To ensure that fixed points exist, we require that all operators $X.S$ be *monotonic*, that is, X may appear only in positive subformulas of S. Thus, the operator $X.!X.\mathsf{end}_!$ is not monotonic, but the operator $X.!(X \to \mathsf{end}_!).\mathsf{end}_!$ is. Formally, $X.S$ is monotonic iff the predicate $pos_{X,S}$ holds, where *pos* and *neg* are defined by the homomorphic extensions of the following equations:

$$\begin{array}{rlrl} pos_{X,X} &= \text{true} & neg_{X,X} &= \text{false} \\ pos_{X,\overline{X}} &= \text{false} & neg_{X,\overline{X}} &= \text{true} \\ pos_{X,!T.S} &= neg_{X,T} \wedge pos_{X,S} & neg_{X,!T.S} &= pos_{X,T} \wedge neg_{X,S} \\ pos_{X,T\multimap U} &= neg_{X,T} \wedge pos_{X,U} & neg_{X,T\multimap U} &= pos_{X,T} \wedge neg_{X,U} \\ pos_{X,T\to U} &= neg_{X,T} \wedge pos_{X,U} & neg_{X,T\to U} &= pos_{X,T} \wedge neg_{X,U} \end{array}$$

2.2 Typing Rules

We let Φ range over type environments. The judgement $\Phi \vdash M : T$ states that term M has type T in type environment Φ. Figure 1 gives the typing rules of μGV. The structural rules account for variables, and for weakening and contraction of unlimited types. The rules for the functional terms are standard; note that, to account for linearity, the context is split in the rules for application and pair introduction and elimination, and is restricted to unlimited types in the introduction of unlimited arrows.

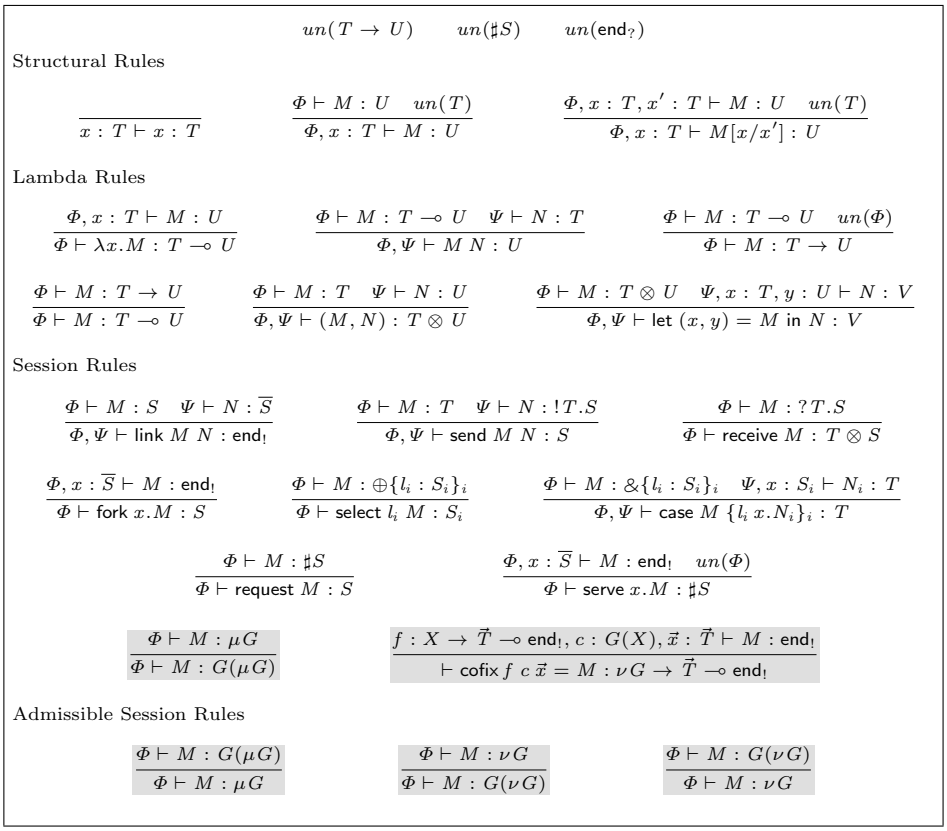

Fig. 1: Typing Rules for μGV

The typing of input, output, choice and selection are those of Gay and Vasconcelos (2010). Following our earlier work (Lindley and Morris, 2014), the term $\mathsf{link}\ M\ N$ implements channel forwarding. The fork construct provides session initiation. The rule for $\mathsf{serve}\, x.M$ parallels that for fork: it defines a server which replicates M, and returns the channel by which it may be used (of type $\overline{!S} = \sharp\overline{S}$).

The novelty of μGV is in its recursive and corecursive channels; we attempt to remain close to existing presentations of recursive session types (Honda et al., 1998). The construct $\mathsf{cofix}\, f\ c\ \vec{x} = M$ is used to define corecursive sessions. We provide for the rolling and unrolling of fixed points μG and νG, implementing the equivalences among equirecursive types. As we observe in Section 4.3, only the first of these coercion rules is essential. The remaining rules are admissible, albeit at the cost of introducing additional computation.

The remainder of the section presents several examples illustrating the use of cofix. In addition, we have implemented a prototype of μGV plus various extensions, and we have implemented many examples, including all of those of Toninho et al. (2013). All of the code is available at the following URL:

`https://github.com/jgbm/cpgv`.

2.3 Streams

We will write $\mathsf{let}\, x = M \,\mathsf{in}\, N$ for $(\lambda x.M)\, N$ and assume an extension of μGV with a type of naturals (Nat), literals $(0, 1, \dots)$, addition $(+)$, and multiplication $(\times)$.

A canonical example of a corecursive data type is a stream. Let us consider a session type for producing a stream of naturals.

$$Source = \nu X.\, \&\{next : !Nat.X, stop : \mathsf{end}_!\}$$

A channel of type *Source* can either produce the next number or stop. The dual of a source is a sink.

$$Sink = \mu X.\, \oplus\{next : ?Nat.X, stop : \mathsf{end}_?\}$$

Using cofix we can define a function that sends a stream of zeros along a corecursive channel:

$$\begin{array}{l} Zeros : Source \to \mathsf{end}_! \\ Zeros = \mathsf{cofix}\, f\ c = \mathsf{case}\ c\ \{next\ c.\mathsf{let}\ c = \mathsf{send}\ 0\ c\ \mathsf{in}\ f\ c; stop\ c.c\} \end{array}$$

If the next number is chosen, then a zero is sent along the channel and we recurse. Otherwise we stop. We define a helper macro to read the next value from a stream:

$$\begin{array}{l} GetNext : Sink \to Nat \otimes Sink \\ GetNext\ c = \mathsf{receive}\ (\mathsf{select}\ next\ c) \end{array}$$

and now:

$$\begin{array}{l} \mathsf{let}\ c = \mathsf{fork}\ c.Zeros\ c\ \mathsf{in} \\ \mathsf{let}\ (x, c) = GetNext\ c\ \mathsf{in}\ \mathsf{let}\ (y, c) = GetNext\ c\ \mathsf{in}\ \mathsf{let}\ (z, c) = GetNext\ c\ \mathsf{in} \\ \mathsf{let}\ c = \mathsf{select}\ stop\ c\ \mathsf{in}\ (x, (y, z)) \end{array}$$

begins by forking a stream of zeros, and then reads from the corresponding sink several times, returning $(0, (0, 0))$.

Productivity. We might expect to assign c the type $\&\{next : !Nat.Source, stop : \mathsf{end}_!\}$ and f the type $Source \to \mathsf{end}_!$ in the body of *Zeros*, but then productivity would not be assured and deadlock would be possible. For example, under those typing assumptions, the following would also be well-typed:

$$\mathsf{cofix}\, f\ c = f\ c$$

We restrict the typing rule for cofix in order to guarantee productivity. We use a fresh type variable X to abstract the recursive behavior, so the channel c has type $G(X)$ (in our example $\&\{next : !Nat.X, stop : \mathsf{end}_!\}$) instead of $G(\nu G)$, and f has type $X \to \mathsf{end}_!$ instead of $\nu G \to \mathsf{end}_!$. The body of cofix is thus required to provide exactly one step of the corecursive behavior.

Coinvariants. We allow recursive sessions to maintain internal state (a *coinvariant*). In the cofix typing rule this is captured by the additional arguments $\vec{x}$. For example, we can construct a stream of consecutive naturals:

$$\begin{array}{l} \mathit{Nats} : \mathit{Source} \to \mathsf{end}_! \\ \mathit{Nats} = \mathsf{cofix}\ \mathit{nats}\ c\ x = \mathsf{case}\ c\ \{\mathit{next}\ c.\ \mathsf{let}\ c = \mathsf{send}\ x\ c\ \mathsf{in}\ \mathit{nats}\ c\ (x+1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{stop}\ c.\ c\} \end{array}$$

The variable x tracks the next value to send to the stream, and is accordingly incremented in the recursive call. Now, the following:

$$\begin{array}{l} \mathsf{let}\ c = \mathsf{fork}\ c.\mathit{Nats}\ c\ 0\ 1\ \mathsf{in} \\ \mathsf{let}\ (x, c) = \mathit{GetNext}\ c\ \mathsf{in}\ \mathsf{let}\ (y, c) = \mathit{GetNext}\ c\ \mathsf{in}\ \mathsf{let}\ (z, c) = \mathit{GetNext}\ c\ \mathsf{in} \\ \mathsf{let}\ c = \mathsf{select}\ \mathit{stop}\ c\ \mathsf{in}\ (x, (y, z)) \end{array}$$

returns $(0, (1, 2))$. We can maintain coinvariants of arbitrary complexity; for example, we can define a stream of the Fibonacci numbers as follows:

$$\begin{array}{l} \mathit{Fibs} : \mathit{Source} \to \mathsf{end}_! \\ \mathit{Fibs} = \mathsf{cofix}\ \mathit{fibs}\ c\ m\ n = \mathsf{case}\ c\ \{\mathit{next}\ c.\mathsf{let}\ c = \mathsf{send}\ m\ c\ \mathsf{in}\ \mathit{fibs}\ c\ n\ (m+n) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{stop}\ c.\ c\} \end{array}$$

2.4 Multi-function Calculator

We now define a basic multi-function calculator. We aim to implement a process that accepts a stream of addition and multiplication requests, and can at any time provide the accumulated result:

$$\mathit{Calc} = \nu X.\ \&\ \{\mathit{add} : ?\mathit{Nat}.X, \mathit{mul} : ?\mathit{Nat}.X, \mathit{result} : !\mathit{Nat}.X, \mathit{stop} : \mathsf{end}_!\}.$$

We implement a provider of calculator functionality as follows:

$$\begin{array}{l} \mathit{MakeCalc} : \mathit{Calc} \to \mathit{Nat} \multimap \mathsf{end}_! \\ \mathit{MakeCalc} = \mathsf{cofix}\ \mathit{calc}\ c\ \mathit{accum} = \\ \quad \mathsf{case}\ c\ \mathsf{of}\ \{\mathit{add}\ c. \quad \mathsf{let}\ (x, c) = \mathsf{receive}\ c\ \mathsf{in}\ \mathit{calc}\ c\ (\mathit{accum} + x) \\ \qquad\qquad \mathit{mul}\ c. \quad \mathsf{let}\ (x, c) = \mathsf{receive}\ c\ \mathsf{in}\ \mathit{calc}\ c\ (\mathit{accum} \times x) \\ \qquad\qquad \mathit{result}\ c. \mathsf{let}\ c = \mathsf{send}\ \mathit{accum}\ c\ \mathsf{in}\ \mathit{calc}\ c\ \mathit{accum} \\ \qquad\qquad \mathit{stop}\ c. \quad c\} \end{array}$$

Here is an example of using a calculator:

$$\begin{array}{l} \mathsf{let}\ c = \mathsf{fork}\ c.\mathit{MakeCalc}\ c\ 0\ \mathsf{in} \\ \mathsf{let}\ c = \mathsf{send}\ 6\ (\mathsf{select}\ \mathit{add}\ c)\ \mathsf{in}\ \mathsf{let}\ c = \mathsf{send}\ 7\ (\mathsf{select}\ \mathit{mul}\ c)\ \mathsf{in} \\ \mathsf{let}\ (x, c) = \mathsf{receive}\ (\mathsf{select}\ \mathit{result}\ c)\ \mathsf{in}\ \mathsf{let}\ c = \mathsf{select}\ \mathit{stop}\ c\ \mathsf{in}\ x \end{array}$$

We begin by constructing an instance of the calculator with the accumulator starting at 0. We perform several calculations, adding 6 to the accumulator and multiplying by 7. Finally, we obtain the result (42) and close the channel.

$$\frac{}{x \leftrightarrow w \vdash x : A, w : A^{\perp}} \qquad \frac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, y : A^{\perp}}{\mathsf{new}\ y\ (P \mid Q) \vdash \Gamma, \Delta} \qquad \frac{}{x[].0 \vdash x : 1}$$

$$\frac{P \vdash \Gamma, y : A, x : B}{x(y).P \vdash \Gamma, x : A \parr B} \qquad \frac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}$$

$$\frac{P \vdash \Gamma, x : A_i}{x[in_i].P \vdash \Gamma, x : A_1 \oplus A_2} \qquad \frac{P \vdash \Gamma, x : A \quad Q \vdash \Gamma, x : B}{\mathsf{case}\ x\ \{P; Q\} \vdash \Gamma, x : A \,\&\, B} \qquad \frac{}{\mathsf{case}\ x\ \{\} \vdash \Gamma, x : \top}$$

$$\frac{P \vdash \Gamma}{P \vdash \Gamma, x : ?A} \qquad \frac{P \vdash y : A, \Gamma}{?x[y].P \vdash \Gamma, x : ?A} \qquad \frac{P \vdash x : ?A, x' : ?A, \Gamma}{P[x/x'] \vdash \Gamma, x : ?A}$$

$$\frac{P \vdash y : A, ?\Gamma}{!x(y).P \vdash \Gamma, x : !A} \qquad \frac{P \vdash \Gamma, x : F(\mu F)}{\mathsf{rec}\ x.P \vdash \Gamma, x : \mu F} \qquad \frac{P \vdash \Gamma, y : A \quad Q \vdash y : A^{\perp}, x : F(A)}{\mathsf{corec}^{F}\, x\langle y\rangle(P, Q) \vdash \Gamma, x : \nu F}$$

Fig. 2: Typing Rules for μCP

3 A Linear Logic-Based Process Calculus

The types of μCP are the propositions of classical linear logic, extended with type operators F and fixed points $\mu F, \nu F$.

$$\begin{array}{lll} \text{Types} & A, B ::= & A \otimes B \mid A \parr B \mid 1 \mid \bot \mid A \oplus B \mid A \,\&\, B \mid 0 \mid \top \mid ?A \mid !A \\ & & \mid\ X \mid X^{\perp} \mid \mu F \mid \nu F \\ \text{Type Operators} & F ::= & X.A \end{array}$$

If $F = X.A$, define $F(B) = A[B/X]$. The standard notion of classical linear logic duality is extended to fixed points in the expected fashion:

$$\begin{array}{llll} (A \otimes B)^{\perp} = A^{\perp} \parr B^{\perp} & 1^{\perp} = \bot & (A \oplus B)^{\perp} = A^{\perp} \,\&\, B^{\perp} & \top^{\perp} = 0 \\ (A \parr B)^{\perp} = A^{\perp} \otimes B^{\perp} & \bot^{\perp} = 1 & (A \,\&\, B)^{\perp} = A^{\perp} \oplus B^{\perp} & 0^{\perp} = \top \\ & (!A)^{\perp} = ?A^{\perp} & (\nu F)^{\perp} = \mu F^{\perp} & \\ & (?A)^{\perp} = !A^{\perp} & (\mu F)^{\perp} = \nu F^{\perp} & \end{array}$$

where $X^{\perp\perp} = X$, and we define the dual of a type operator by $F^{\perp} = X.(F(X^{\perp}))^{\perp}$. Similarly to μGV, $(F(A))^{\perp} = F^{\perp}(A^{\perp})$. We let Γ, Δ range over type environments. The judgement $P \vdash \Gamma$ states that process P uses channels Γ. Typing rules for μCP terms are given in Fig. 2. We write $fv(P)$ for the free variables used in process P; in the typing rules, new bound variables are designated y.

Structural Rules. μCP has two structural rules, axiom and cut. We interpret the axiom $x \leftrightarrow w$ as channel forwarding: actions on channel x are mirrored on w, and vice versa. Thus, x and w must have dual type. Cut $\mathsf{new}\ y\ (P \mid Q)$ is interpreted as communication between processes P and Q on channel y; the duality of the typing of y assures that its uses in P and Q are compatible. We identify μCP up to structural equivalence:

$$\begin{array}{rl} x \leftrightarrow w \equiv w \leftrightarrow x & \\ \mathsf{new}\ y\ (P \mid Q) \equiv \mathsf{new}\ y\ (Q \mid P) & \\ \mathsf{new}\ y\ (P \mid \mathsf{new}\ z\ (Q \mid R)) \equiv \mathsf{new}\ z\ (\mathsf{new}\ y\ (P \mid Q) \mid R), & \text{if } y \notin fv(R) \end{array}$$

Input and Output. The multiplicative connectives $\parr$ and $\otimes$ are interpreted as input and output. The process $x(y).P$ inputs channel y on channel x, and continues as P. The process $x[y].(P \mid Q)$ is interpreted as bound output: it sends a fresh variable y along x, spawns a process P in which y is used, and continues as process Q in which x is used. It amounts to the π-calculus term $(\nu y)\overline{x}(y).(P \mid Q)$. We write $x[y].P$ as syntactic sugar for $x[y'].(y \leftrightarrow y' \mid P)$. The units of $\parr$ and $\otimes$, $\bot$ and 1, are interpreted as nullary input and nullary output, respectively.

Selection and Choice. The additive connectives $\oplus$ and $\&$ are interpreted as selection and choice. The process $\mathsf{case}\ x\ \{P_1; P_2\}$ offers a choice of processes P_1 and P_2; dually, the process $x[in_i].P_i$ chooses the i-th alternative. The unit for choice is 0, indicating absurdity. Note that there is no term proving 0. The dual of absurdity is $\top$, and provides arbitrary behavior; as there is no term proving 0, no term relying on $\top$ can reduce.

Replication and Dereliction. The exponential connectives ! and ? in linear logic provide restricted access to the classical rules of weakening and contraction. We interpret them as serving (!) and requesting (?) replicated processes. For a server channel of type $!A$, the process P proving A may be replicated arbitrarily, so each channel that P uses must be replicable as well. We write $?\Gamma$ to assert that all types in Γ be of the form $?B$. As well as the introduction rule for a request channel of type $?A$, there are also implicit rules for weakening and contraction.

Recursion and Corecursion. We introduce least fixed points μF and greatest fixed points νF to CP, following Baelde's (2012) proof theoretic treatment of fixed points in linear logic. The proof rules can be understood from traditional two-sided rules for least and greatest fixed points, combined with the duality between the fixed points. We begin with a two-sided presentation:

$$\frac{F(A) \vdash A \quad \Gamma, A \vdash B}{\Gamma, \mu F \vdash B} \qquad \frac{\Gamma, F(\nu F) \vdash B}{\Gamma, \nu F \vdash B}$$

$$\frac{\Gamma \vdash F(\mu F)}{\Gamma \vdash \mu F} \qquad \frac{A \vdash F(A) \quad \Gamma \vdash A}{\Gamma \vdash \nu F}$$

Functional programmers may recognize the first as the rule for a fold, and the fourth as the rule for an unfold. We adapt the above rules to a one sided presentation as follows. (As Γ denotes any context, we write Γ instead of $\Gamma^\bot$).

$$\frac{\vdash F^\bot(A^\bot), A \quad \vdash \Gamma, A^\bot, B}{\vdash \Gamma, \nu F^\bot, B} \qquad \frac{\vdash \Gamma, F^\bot(\mu F^\bot), B}{\vdash \Gamma, \mu F^\bot, B}$$

$$\frac{\vdash \Gamma, F(\mu F)}{\vdash \Gamma, \mu F} \qquad \frac{\vdash A^\bot, F(A) \quad \vdash \Gamma, A}{\vdash \Gamma, \nu F}$$

However, now we can observe that Γ, B is itself an instance of a context, and $F^\bot$ a type operator, and so the top-right rule is just an instance of the bottom-left rule, and the top-left rule is just an instance of the bottom-right rule. The bottom two

$$
\begin{aligned}
\mathsf{new}\ x\ (w \leftrightarrow x \mid P) &\Longrightarrow P[w/x] \\
\mathsf{new}\ x\ (x[y].(P \mid Q) \mid x(y).R) &\Longrightarrow \mathsf{new}\ y\ (Q \mid \mathsf{new}\ x\ (P \mid R)) \\
\mathsf{new}\ x\ (x[].0 \mid x().P) \vdash \Gamma &\Longrightarrow P \\
\mathsf{new}\ x\ (x[in_i].P \mid \mathsf{case}\ x\ \{Q_1; Q_2\}) &\Longrightarrow \mathsf{new}\ x\ (P \mid Q_i) \\
\mathsf{new}\ x\ (!x(y).P \mid ?x[y].Q) &\Longrightarrow \mathsf{new}\ y\ (P \mid Q) \\
\mathsf{new}\ y\ (!x(y).P \mid Q) &\Longrightarrow Q, \qquad x \notin fv(Q) \\
\mathsf{new}\ x\ (!x(y).P \mid Q[x/x']) &\Longrightarrow \mathsf{new}\ x\ (!x(y).P \mid \mathsf{new}\ x'\ (!x'(y).P \mid Q)) \\
\mathsf{new}\ x\ (\mathsf{corec}^F x\langle y\rangle(P, Q) \mid \mathsf{rec}\ x.R) &\Longrightarrow \mathsf{new}\ y\ (P \mid \mathsf{new}\ x\ (Q \mid \mathsf{new}\ x'\ (Q' \mid R[x'/x]))) \\
&\qquad \text{where } Q' = \mathsf{map}^F_{x,x'}(\mathsf{corec}^F x\langle y\rangle(x \leftrightarrow y, Q[x'/x]))
\end{aligned}
$$

Fig. 3: Principal Cut Elimination Rules

$$
\begin{aligned}
\mathsf{map}^{X.C}_{x,y}(P) &= x \leftrightarrow y, \qquad \text{if } X \text{ is not free in } C \\
\mathsf{map}^{X.X}_{x,y}(P) &= P \\
\mathsf{map}^{X.C_1 \otimes C_2}_{x,y}(P) &= x(x').y[y'].(\mathsf{map}^{X.C_1}_{x',y'}(P[x'/x, y'/y]) \mid \mathsf{map}^{X.C_2}_{x,y}(P)) \\
\mathsf{map}^{X.C_1 \oplus C_2}_{x,y}(P) &= \mathsf{case}\ x\ \{y[in_1].\mathsf{map}^{X.C_1}_{x,y}(P); y[in_2].\mathsf{map}^{X.C_2}_{x,y}(P)\} \\
\mathsf{map}^{X.?C}_{x,y}(P) &= !x(x').?y[y'].P[x'/x, y'/y] \\
\mathsf{map}^{X.\mu F}_{x,y}(P) &= \mathsf{corec}^F x\langle y'\rangle(y \leftrightarrow y', \mathsf{rec}\ y'.\mathsf{map}^{X.F(\mu F)}_{x,y'}(P[y'/y]))
\end{aligned}
$$

Fig. 4: Definition of map for Positive Combinators

rules are the typing rules for the μCP terms $\mathsf{rec}\ x.P$ and $\mathsf{corec}^F x\langle y\rangle(P, Q)$. We will often omit the F annotation. As in μGV, corec terms maintain a coinvariant y, of type A. The coinvariant can be seen as being sent from P to Q; thus, its type is dual in P and Q. As carried types are not dual in μGV, no corresponding duality appears in the typing rule for cofix.

3.1 Cut Elimination

Cut elimination corresponds to synchronous process reduction. The principal cut reductions are given in Fig. 3. The majority of these are standard; for instance, cut reduction of & against $\oplus$ corresponds to picking one of the offered alternatives. Cut reduction for fixed points corresponds to unrolling one iteration from the corec term, directed by the type of the fixed point operator F. It depends on a proof construction known as functoriality, which derives the following proof rule for any type operator F:

$$\frac{\vdash A, B}{\vdash F^{\perp}(A), F(B)}$$

We call the term implementing this construction map by analogy with a similar construct in functional programming. The positive cases of map are given in Fig. 4; the remaining cases are obtained by exchanging channels x and y.

Lemma 1. *If* $P \vdash x : A, y : B$ *then* $\mathsf{map}^F_{x,y}(P) \vdash x : F^{\perp}(A), y : F(B)$.

The commuting conversions push communication under process prefixes, and are given in Fig. 5. The standard meta theoretic properties of classical linear

$$
\begin{array}{rl}
\mathsf{new}\ z\ (x[y].(P \mid Q) \mid R) & \Longrightarrow x[y].(\mathsf{new}\ z\ (P \mid R) \mid Q), \qquad \text{if } z \notin fv(Q) \\
\mathsf{new}\ z\ (x[y].(P \mid Q) \mid R) & \Longrightarrow x[y].(P \mid \mathsf{new}\ z\ (Q \mid R)), \qquad \text{if } z \notin fv(P) \\
\mathsf{new}\ z\ (x(y).P \mid Q) & \Longrightarrow x(y).\mathsf{new}\ z\ (P \mid Q) \\
\mathsf{new}\ z\ (x[in_i].P \mid Q) & \Longrightarrow x[in_i].\mathsf{new}\ z\ (P \mid Q) \\
\mathsf{new}\ z\ (\mathsf{case}\ x\ \{P; Q\} \mid R) & \Longrightarrow \mathsf{case}\ x\ \{\mathsf{new}\ z\ (P \mid R); \mathsf{new}\ z\ (Q \mid R)\} \\
\mathsf{new}\ z\ (?x[y].P \mid Q) & \Longrightarrow ?x[y].\mathsf{new}\ z\ (P \mid Q) \\
\mathsf{new}\ z\ (!x(y).P \mid Q) & \Longrightarrow !x(y).\mathsf{new}\ z\ (P \mid Q) \\
\mathsf{new}\ z\ (\mathsf{rec}\ x.P \mid Q) & \Longrightarrow \mathsf{rec}\ x.\mathsf{new}\ z\ (P \mid Q) \\
\mathsf{new}\ z\ (\mathsf{corec}\ x\langle y\rangle(P, Q) \mid R) & \Longrightarrow \mathsf{corec}\ x\langle y\rangle(\mathsf{new}\ z\ (P \mid R), Q)
\end{array}
$$

Fig. 5: Commuting Conversions

logic with fixed points apply directly to μCP. Therefore deadlock and livelock freedom follow directly from cut elimination, and race freedom follows directly from confluence.

3.2 Streams

As we did for μGV, we assume that our language is extended with constants implementing unlimited natural numbers (we could implement *Nat* within μCP, but choose not to due to lack of space). In particular, assume there exists a proposition *Nat*, such that $Nat^{\perp}$ is subject to contraction and weakening, and terms:

$$
\begin{array}{c}
Zero_x \vdash x : Nat \\
Inc_{y,x} \vdash y : Nat^{\perp}, x : Nat
\end{array}
$$

We can now use fixed points in μCP to encode streams of naturals.

$$
\begin{array}{rl}
Source &= \nu X.\ \&\ \{next : Nat \otimes X, stop : 1\} \\
Sink &= \mu X.\ \oplus\ \{next : Nat^{\perp} \parr X, stop : \perp\}
\end{array}
$$

We can define a process which generates a stream of zeros. In this example, we make no use of the coinvariant, and so assign it the type 1:

$$
\begin{array}{l}
Zeros_y \vdash y : Source \\
Zeros_y = \mathsf{corec}\ y\langle z\rangle(z[].0, z().\mathsf{case}\ y\ \{next : y[x].(Zero_x | y[].0); stop : y[].0\}).
\end{array}
$$

Now we define a process which generates a stream of naturals. As in the corresponding μGV example, we use the coinvariant represent the next number in the stream.

$$
\begin{array}{l}
Nats_y \vdash y : Source \\
Nats_y = \mathsf{cofix}\ y\langle z\rangle(Zero_z, \mathsf{case}\ y\ \{next : y[x].(z \leftrightarrow x \mid \mathsf{new}\ w\ (Inc_{z,w} \mid w \leftrightarrow y)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad stop : y[].0\})
\end{array}
$$

In the *next* case, we rely on contraction to copy the coinvariant z. We then send one copy along the channel y and increment the other giving w, which we use to re-establish the coinvariant. In the *stop* case, we rely on weakening for $Nat^{\perp}$ to dispose of the coinvariant.

The following receives the first three numbers from the stream of naturals and sends them on z; thus, z has type $Nat^{\perp} \otimes (Nat^{\perp} \otimes Nat^{\perp})$.

$$
\begin{array}{c}
\mathsf{new}\ y\ (Nats_y \mid \mathsf{rec}\ y.y[next].y(a).\mathsf{rec}\ y.y[next].y(b).\mathsf{rec}\ y.y[next].y(c). \\
\qquad\qquad \mathsf{rec}\ y.y[stop].y().z[a].z[b].z \leftrightarrow c)
\end{array}
$$

$$
\begin{aligned}
(\lambda x.M)^\star &= \mathsf{fork}\ z.\mathsf{let}\ (x,z) = \mathsf{receive}\ z\ \mathsf{in}\ \mathsf{link}\ (M)^\star\ z \\
(L\,M)^\star &= \mathsf{send}\ (M)^\star\ (L)^\star \\
(M,N)^\star &= \mathsf{fork}\ z.\mathsf{link}\ (\mathsf{send}\ (M)^\star\ z)\ (N)^\star \\
(\mathsf{let}\ (x,y) = M\ \mathsf{in}\ N)^\star &= \mathsf{let}\ (x,y) = \mathsf{receive}\ (M)^\star\ \mathsf{in}\ (N)^\star \\
(L : T \to U)^\star &= \mathsf{serve}\ z.\mathsf{link}\ (L)^\star\ z \\
(L : T \multimap U)^\star &= \mathsf{request}\ (L)^\star \\
(\mathsf{receive}\ M)^\star &= (M)^\star \\
(\mathsf{cofix}\ f\ c\ \vec{x} = M)^\star &= \mathsf{cofix}\ p\ c\ z = \mathsf{let}\ f = \lambda\, c\ \vec{x}.\mathsf{send}\ \vec{x}\ (\mathsf{send}\ c\ (\mathsf{request}\ p))\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \vec{x} = z\ \mathsf{in}\ (M)^\star
\end{aligned}
$$

Fig. 6: Translation of μGV Terms to μGVπ

4 Relating μGV and μCP

In our previous work (Lindley and Morris, 2014) we give translations between CP and HGV, an extension of GV, corresponding to μGV without recursive session types. In this section, we extend these translations to incorporate recursion.

4.1 Translation from μGV to μGVπ

Following our previous work, we factor the translation of μGV into μCP through an intermediate translation. The language μGVπ is the restriction of μGV to session types; that is, μGV without $\multimap$, $\to$, or $\otimes$. In order to avoid $\otimes$, we permit $\mathsf{receive}\,M$, only fused with a pair elimination $\mathsf{let}\ (x,y) = \mathsf{receive}\,M\ \mathsf{in}\ N$. We can simulate all non-session types as session types via a translation from μGV to μGVπ. The translation on types is given by the homomorphic extension of the following equations:

$$
(T \multimap U)^\star = !(T)^\star.(U)^\star \qquad (T \to U)^\star = \sharp(!(T)^\star.(U)^\star)
$$
$$
(T \otimes U)^\star = ?(T)^\star.(U)^\star
$$

Each target type is the interface to the simulated source type. A linear function is simulated by input on a channel; its interface is output on the other end of the channel. An unlimited function is simulated by a server; its interface is the service on the other end of that channel. A tensor is simulated by output on a channel; its interface is input on the other end of that channel. This duality between implementation and interface explains the dualization of types in Wadler's original CPS translation from GV to CP. To translate away the arrows in the cofix rule, we adopt a simplified session-oriented variant of cofix for μGVπ:

$$
\frac{p : \sharp(!X.!T.\mathsf{end}_!), c : G(X), x : T \vdash M : \mathsf{end}_!}{\Phi \vdash \mathsf{cofix}\ p\ c\ x = M : \sharp(!\nu G.!T.\mathsf{end}_!)}
$$

This rule takes advantage of the translation of functions given by $(-)^\star$, and simulates multiple arguments using $\otimes$.

The translation on terms is given by the homomorphic extension of the equations in Fig. 6. Formally, this is a translation from derivations to terms. We write type annotations to indicate $\to$ introduction and elimination. The only new case is that for corec. We collect the arguments into a tuple, and simulate the interface

$$
\begin{aligned}
[\![\Phi, x : \mathsf{end}_? \vdash N : S]\!]z &= x().[\![\Phi \vdash N : S]\!]z \\
[\![\Phi, x : \mathsf{end}_? \vdash N[x/x'] : S]\!]z &= \mathsf{new}\ x'\ ([\![\Phi, x : \mathsf{end}_?, x' : \mathsf{end}_? \vdash N : S]\!]z \mid x'[].0) \\
[\![x]\!]z &= x \leftrightarrow z \\
[\![\mathsf{fork}\ x.M]\!]z &= \mathsf{new}\ x\ (\mathsf{new}\ y\ ([\![M]\!]y \mid y[].0) \mid x \leftrightarrow z) \\
[\![\mathsf{link}\ M\ N]\!]z &= z().\mathsf{new}\ x\ ([\![M]\!]x \mid [\![N]\!]x) \\
[\![\mathsf{send}\ M\ N]\!]z &= \mathsf{new}\ x\ (x[y].([\![M]\!]y \mid x \leftrightarrow z) \mid [\![N]\!]x) \\
[\![\mathsf{let}\ (x, y) = \mathsf{receive}\ M\ \mathsf{in}\ M]\!]z &= \mathsf{new}\ y\ ([\![M]\!]y \mid y(x).[\![N]\!]z) \\
[\![\mathsf{select}\ l\ M]\!]z &= \mathsf{new}\ x\ ([\![M]\!]x \mid x[l].x \leftrightarrow z) \\
[\![\mathsf{case}\ M\ \{l_i\ x.N_i\}_i]\!]z &= \mathsf{new}\ x\ ([\![M]\!]x \mid \mathsf{case}\ x\ \{l_i.[\![N_i]\!]z\}_i) \\
[\![\mathsf{request}\ M]\!]z &= \mathsf{new}\ x\ ([\![M]\!]x \mid ?x[y].y \leftrightarrow z) \\
[\![\mathsf{serve}\ y.M]\!]z &= !z(y).\mathsf{new}\ x\ ([\![M]\!]x \mid x[].0) \\
[\![\Phi \vdash M : G(\mu G)]\!]z &= \mathsf{new}\ y\ ([\![\Phi \vdash M : \mu G]\!]y \mid \mathsf{rec}\ y.y \leftrightarrow z) \\
[\![\mathsf{cofix}\ p\ c\ x = M]\!]z &= !z(y).y(c).y(w).y().\mathsf{corec}\ c\langle x\rangle(x \leftrightarrow w, \\
&\qquad \mathsf{new}\ p\ (!p(y).y(c).y(w).y().c \leftrightarrow w \mid \\
&\qquad\quad \mathsf{new}\ y\ ([\![M]\!]y \mid y[].0))) \\
[\![\Phi \vdash M : \mu G]\!]z &= \mathsf{new}\ y\ ([\![\Phi \vdash M : G(\mu G)]\!]y \\
&\qquad \mid \mathsf{corec}\ z\langle x\rangle(y \leftrightarrow x, \mathsf{map}^G_{z,x}(\mathsf{rec}\ x.x \leftrightarrow z))) \\
[\![\Phi \vdash M : G(\nu G)]\!]z &= \mathsf{new}\ y\ ([\![\Phi \vdash M : \nu G]\!]y \\
&\qquad \mid \mathsf{corec}\ y\langle x\rangle(y \leftrightarrow x, \mathsf{map}^G_{z,x}(\mathsf{rec}\ x.x \leftrightarrow z))) \\
[\![\Phi \vdash M : \nu G]\!] &= \mathsf{new}\ y\ ([\![\Phi \vdash M : G(\nu G)]\!]y \mid \mathsf{rec}\ z.y \leftrightarrow z)
\end{aligned}
$$

Fig. 7: Translation of $\mu\mathrm{GV}\pi$ Terms into $\mu\mathrm{CP}$

to the arrows using session operations as in the rest of the $(-)^\star$ translation. We use the obvious encodings for n-ary let binding, lambdas, and send. We write $(\Phi)^\star$ for the pointwise extension of $(T)^\star$.

Theorem 2. *If $\Phi \vdash M : T$ then $(\Phi)^\star \vdash (M)^\star : (T)^\star$.*

4.2 Translation from μGVπ to μCP

We now give a translation from $\mu\mathrm{GV}\pi$ to CP. Post composing this with the embedding of $\mu\mathrm{GV}$ in $\mu\mathrm{GV}\pi$ yields a semantics for $\mu\mathrm{GV}$. For uniformity, we assume the obvious encodings of n-ary $\oplus$, $\&$, and case in CP. The translation on session types is as follows:

$$
\begin{aligned}
[\![!T.S]\!] &= [\![T]\!]^\perp \otimes [\![S]\!] \\
[\![?T.S]\!] &= [\![T]\!] \parr [\![S]\!] \\
[\![\mathsf{end}_!]\!] &= 1 \\
[\![\mathsf{end}_?]\!] &= \bot
\end{aligned}
\qquad
\begin{aligned}
[\![\oplus\{l_i : S_i\}_i]\!] &= \oplus\{l_i : [\![S_i]\!]\}_i \\
[\![\&\{l_i : S_i\}_i]\!] &= \&\{l_i : [\![S_i]\!]\}_i \\
[\![\sharp S]\!] &= ?[\![S]\!] \\
[\![\flat S]\!] &= ![\![S]\!]
\end{aligned}
\qquad
\begin{aligned}
[\![X]\!] &= X \\
[\![\overline{X}]\!] &= X^\perp \\
[\![\nu G]\!] &= \nu[\![G]\!] \\
[\![\mu G]\!] &= \mu[\![G]\!] \\
[\![X.S]\!] &= X.[\![S]\!]
\end{aligned}
$$

The translation is homomorphic except for output, where the argument type is dualized. This accounts for the discrepancy between $\overline{!T.S} = ?T.\overline{S}$ and $(A \otimes B)^\perp = A^\perp \parr B^\perp$. The translation on terms (Fig. 7) is formally specified as a CPS translation on derivations as in Wadler's work. We abbreviate the derivation by its final judgement in the translations of weakening and contraction for $\mathsf{end}_?$, and for rolling and unrolling of recursive session types νG and μG, as these steps are implicit in the syntax of μGV terms, but explicit in μCP. The other

$$
\begin{aligned}
(\!|\mathsf{new}\ x\ (P \mid Q)|\!) &= \mathsf{let}\ x = \mathsf{fork}\ x.(\!|P|\!)\ \mathsf{in}\ (\!|Q|\!) \\
(\!|x \leftrightarrow y|\!) &= \mathsf{link}\ x\ y \\
(\!|x[y].(P \mid Q)|\!) &= \mathsf{let}\ x = \mathsf{send}\ (\mathsf{fork}\ y.(\!|P|\!))\ x\ \mathsf{in}\ (\!|Q|\!) \\
(\!|x(y).P|\!) &= \mathsf{let}\ (y, x) = \mathsf{receive}\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x[].0|\!) &= x \\
(\!|x().P|\!) &= (\!|P|\!) \\
(\!|x[l].P|\!) &= \mathsf{let}\ x = \mathsf{select}\ l\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|\mathsf{case}\ x\ \{l_i.P_i\}_i|\!) &= \mathsf{case}\ x\ \{l_i.(\!|P_i|\!)\}_i \\
(\!|?s[x].P|\!) &= \mathsf{let}\ x = \mathsf{request}\ s\ \mathsf{in}\ (\!|P|\!) \\
(\!|!s(x).P|\!) &= \mathsf{link}\ s\ (\mathsf{serve}\ x.(\!|P|\!)) \\
(\!|\mathsf{rec}\ x.P|\!) &= (\!|P|\!) \\
(\!|\mathsf{corec}^F\ c\langle x\rangle(P, Q)|\!) &= \mathsf{send}\ c\ (\mathsf{send}\ (\mathsf{fork}\ x.(\!|P|\!)) \\
&\qquad\qquad (\mathsf{request}\ (\mathsf{cofix}\ p\ c\ x = Q_{F,p,c,x}))) \\
\text{where } Q_{F,p,c,x} &= \mathsf{let}\ c' = \mathsf{fork}\ c.(\!|Q|\!)\ \mathsf{in}\ (\!|\mathsf{map}^F_{c',c}(?p[z].z[c].z[c'].z[].0)|\!)
\end{aligned}
$$

Fig. 8: Translation of μCP Terms into μGVπ

constructs depend only on the immediate syntactic structure, so we abbreviate their translations as mappings on plain terms. The majority of translations are as in our previous work. The rolling and unrolling of fixed points in μGV can be translated to cuts against the μCP proofs of the corresponding equivalences. The translation of cofix is a corecursive process with coinvariant x in which p abstracts re-establishing the coinvariant at the end of each step.

Theorem 3. *If $\Phi \vdash M : T$, then $[\![M]\!]z \vdash [\![\Phi]\!], z : [\![T]\!]^{\perp}$.*

As we have already argued that μCP is deadlock and livelock free, and we now have that all μGV terms correspond to μCP derivations, we can also conclude that μGV is deadlock and livelock free. Note that the type environment Φ is not dual in the result of the translation. This reflects the different interpretations of types in μCP and μGV: a μGV type $!T.S$ designates a channel that expects output, whereas the μCP type $T \otimes S$ designates a process that performs output.

4.3 Translating from μCP to μGVπ

We now present the translation $(\!|-|\!)$ from μCP to μGVπ. On types:

$$
\begin{aligned}
(\!|A \otimes B|\!) &= !\overline{(\!|A|\!)}.(\!|B|\!) & (\!|+\{l_i : A_i\}_i|\!) &= +\{l_i : (\!|A_i|\!)\}_i \\
(\!|A \parr B|\!) &= ?(\!|A|\!).(\!|B|\!) & (\!|\&\{l_i : A_i\}_i|\!) &= \&\{l_i : (\!|A_i|\!)\}_i \\
(\!|1|\!) &= \mathsf{end}_! & (\!|?A|\!) &= \sharp(\!|A|\!) \\
(\!|\bot|\!) &= \mathsf{end}_? & (\!|!A|\!) &= \flat(\!|A|\!)
\end{aligned}
\qquad
\begin{aligned}
(\!|X|\!) &= X \\
(\!|X^{\perp}|\!) &= \overline{X} \\
(\!|\nu F|\!) &= \nu(\!|F|\!) \\
(\!|\mu F|\!) &= \mu(\!|F|\!) \\
(\!|X.A|\!) &= X.(\!|A|\!)
\end{aligned}
$$

The translation on terms makes use of let expressions to simplify the presentation; these are expanded to μGVπ as follows:

$$\mathsf{let}\ x = M\ \mathsf{in}\ N \mapsto ((\lambda x.N)M)^{\star} = \mathsf{send}\ M\ (\mathsf{fork}\ z.\mathsf{let}\ (x, z) = \mathsf{receive}\ z\ \mathsf{in}\ \mathsf{link}\ N\ z)$$

The translation is given in Fig. 8. The interesting cases are for rec and corec. The translation of $\mathsf{rec}\ x.P$ relies on exactly one use of the rule that unrolls

μF to $F(\mu F)$ applied to the variable x. The translation of corec simulates the expansion of Q, using p to re-establish the coinvariant after each expansion, and thus depends on the translation of functoriality from μCP to μGV.

Theorem 4. *If* $P \vdash \Gamma$, *then* $(\!|\Gamma|\!) \vdash (\!|P|\!) : \mathsf{end}_!$.

As the semantics of μGV is defined by translation to μCP, the following soundness theorem tells us that μGV and μCP are equally expressive.

Theorem 5. *If* $P \vdash \Gamma$ *then* $\mathsf{new}\ z\ (z[].0 \mid [\![(\!|P|\!)]\!]z) \Longrightarrow^* P$.

Remark. The admissibility of the final three rules of Fig. 1 follows from Theorem 5, as neither $[\![-]\!]$ nor $(\!|-|\!)$ uses any of these rules.

5 Related Work

Session types were originally introduced by Honda (1993), and were further extended by Takeuchi et al. (1994), Honda et al. (1998), and Yoshida and Vasconcelos (2007). The systems of Honda et al. (1998) and Yoshida and Vasconcelos (2007) include recursive session types. They do not distinguish between recursion and corecursion, and do not exhibit deadlock freedom. Bono and Padovani (2012) and Bernardi and Hennessy (2013) independently observed that duality in those systems was not preserved under unrolling of recursive types; Bernardi et al. (2014) studies the role of duality in session types, including those with recursion. This work adopts a different solution from ours, however, and we hope to investigate the consequences of these different approaches in future work.

Caires and Pfenning (2010) showed the first complete propositions-as-types correspondence between intuitionistic linear logic (ILL) and session types. Their work shows both an interpretation of session types as ILL propositions, and a computational interpretation of ILL proofs as π-calculus processes. As a result, they are able to show that well-typed processes are free of races, deadlock, and livelock, by analogy with corresponding cut-elimination results for ILL. Toninho et al. (2013) demonstrate an embedding of their calculus within a functional language; the resulting system admits unrestricted recursion in the functional setting, and can thus provide recursive communication behavior, but does not guarantee that the evaluation of function terms terminates.

Girard (1987) speculated that linear logic would be well-suited to reasoning about concurrency. Abramsky (1994) and Bellin and Scott (1994) explored the interpretation of linear logic proofs as concurrent programs. Kobayashi et al. (1996) introduced the use of linear typing to the π-calculus, and demonstrated a form of linear channels similar in usage to session-typed channels; Dardha et al.'s (2012) extensions to this work include full session types.

6 Conclusion and Future Work

We have demonstrated a propositions-as-types correspondence linking recursive session types and fixed points in linear logic. Unlike previous work on recursive

session types, our presentation distinguishes between recursive and corecursive processes. As a consequence of cut elimination in linear logic, all well-typed processes are free of races, deadlock, and livelock. We identify several areas of future work. We would like to give an asynchronous semantics to μGV, following the original presentation of LAST by Gay and Vasconcelos (2010), and show that it is equivalent to the synchronous semantics provided by cut elimination in μCP. We would like to extend μGV with recursive and corecursive types in general, and investigate whether we can extend the $(-)^\star$ translation to simulate these types as recursive and corecursive session types. We would like to explore further extensions of μCP and their consequences for μGV, including: the addition of second-order polymorphism, as studied by Wadler (2012) for CP and Lindley and Morris (2014) for GV; the addition of the MIX_0 and MIX_2 rules, providing additional notions of parallel composition and unifying the types $\mathsf{end}_!$ and $\mathsf{end}_?$; and identify natural extensions of CP that give rise to non-determinism, and thus allow programs to exhibit more interesting concurrent behavior, while preserving the underlying connection to linear logic.

References

S. Abramsky. Proofs as processes. In *Selected Papers of the Conference on Meeting on the Mathematical Foundations of Programming Semantics, Part I : Linear Logic: Linear Logic*, MFPS '92, pages 5–9, Oxford, United Kingdom, 1994. Elsevier.

D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Logic*, 13(1): 2:1–2:44, Jan. 2012. ISSN 1529-3785.

G. Bellin and P. J. Scott. On the π-Calculus and linear logic. *Theoretical Computer Science*, 135 (1):11–65, 1994.

G. Bernardi and M. Hennessy. Using higher-order contracts to model session types. *CoRR*, abs/1310.6176, 2013.

G. Bernardi, O. Dardha, S. J. Gay, and D. Kouzapas. On duality relations for session types, 2014. Draft.

V. Bono and L. Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8(1), 2012.

L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, CONCUR '10, pages 222–236, 2010.

O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, 2012.

R. Demangeon and K. Honda. Full abstraction in a subtyped π-calculus with linear types. In *CONCUR*, pages 280–296, 2011.

S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, Jan. 1987.

K. Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523, 1993.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138, 1998.

N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996.

S. Lindley and G. Morris. Sessions as propositions. In *PLACES*, 2014.

K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.

B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pages 350–369, Rome, Italy, 2013. Springer-Verlag. ISBN 978-3-642-37035-9.

B. Toninho, L. Caires, and F. Pfenning. Corecursion in session-typed processes, 2014. Draft.

P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286. ACM, 2012.

N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.