9-27-2010

Lolliproc: to Concurrency from Classical Linear Logic via Curry-Howard and Control

Karl Mazurak
University of Pennsylvania

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Karl Mazurak and Steve Zdancewic. **Lolliproc: to Concurrency from Classical Linear Logic via Curry-Howard and Control**. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2010.
doi>10.1145/1863543.1863551

Lolliproc: to Concurrency from Classical Linear Logic via Curry-Howard and Control

Abstract

While many type systems based on the intuitionistic fragment of linear logic have been proposed, applications in programming languages of the full power of linear logic-including double-negation elimination-have remained elusive. Meanwhile, linearity has been used in many type systems for concurrent programs-e.g., session types-which suggests applicability to the problems of concurrent programming, but the ways in which linearity has interacted with concurrency primitives in lambda calculi have remained somewhat ad-hoc. In this paper we connect classical linear logic and concurrent functional programming in the language Lolliproc, which provides simple primitives for concurrency that have a direct logical interpretation and that combine to provide the functionality of session types. Lolliproc features a simple process calculus "under the hood" but hides the machinery of processes from programmers. We illustrate Lolliproc by example and prove soundness, strong normalization, and confluence results, which, among other things, guarantees freedom from deadlocks and race conditions.

Disciplines

Computer Sciences

Comments

Karl Mazurak and Steve Zdancewic. **Lolliproc: to Concurrency from Classical Linear Logic via Curry-Howard and Control**. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2010.

doi>10.1145/1863543.1863551

Lolliproc: to Concurrency from Classical Linear Logic via Curry-Howard and Control

Karl Mazurak Steve Zdancewic
University of Pennsylvania
{mazurak,stevez}@cis.upenn.edu

Abstract

While many type systems based on the intuitionistic fragment of linear logic have been proposed, applications in programming languages of the full power of linear logic—including double-negation elimination—have remained elusive. Meanwhile, linearity has been used in many type systems for concurrent programs—*e.g.*, session types—which suggests applicability to the problems of concurrent programming, but the ways in which linearity has interacted with concurrency primitives in lambda calculi have remained somewhat ad-hoc. In this paper we connect classical linear logic and concurrent functional programming in the language Lolliproc, which provides simple primitives for concurrency that have a direct logical interpretation and that combine to provide the functionality of session types. Lolliproc features a simple process calculus "under the hood" but hides the machinery of processes from programmers. We illustrate Lolliproc by example and prove soundness, strong normalization, and confluence results, which, among other things, guarantees freedom from deadlocks and race conditions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Linear logic, Concurrency, Type systems

1. Introduction: Linearity and Concurrency

Since its introduction by Girard in the 1980's [22], linear logic has suggested applications in type system support for concurrency. Intuitively, the appeal of this connection stems from linear logic's strong notion of resource management: if two program terms use distinct sets of resources, then one should be able to compute them both in parallel without fear of interference, thereby eliminating problems with race conditions or deadlock. Moreover, linear logic's ability to account for stateful computation [42], when combined with the concurrency interpretation above, suggests that it is a good fit for describing stateful communication protocols in which the two endpoints must be synchronized.

Indeed, there have been many successful uses of linearity in type systems for concurrent programming. Ideas from linearity play a crucial role in *session types* [12, 15, 25, 38, 40], for example, where they are used to ensure that two end-points of a channel agree on which side is to send the next message and what type of data should be sent. Linearity is also useful for constraining the behavior of π-calculus processes [4, 28], and can be strong enough to yield fully-abstract encodings of (stateful) lambda-calculi [45].

Given all this, it is natural to seek out programming-language constructs that correspond directly to linear logic connectives via the Curry-Howard correspondence [26]. In doing so, one would hope to shed light on the computational primitives involved and, eventually, to apply those insights in the contexts of proof theory and programming-language design. Here too, there has been much progress, which falls, roughly, into three lines of work.

First, there has been considerable effort to study various *intuitionistic* fragments of linear logic [6, 11, 29–31, 39]. This has yielded type systems and programming models that are relatively familiar to functional programmers and have applications in managing state and other resources [2, 13, 16, 24, 41, 47]. However, such intuitionistic calculi do not exploit concurrency (or non-standard control operators) to express their operational semantics.

A second approach has been to formulate proof terms for the *sequent calculus* presentation of linear logic. This path leads to proof nets, as in Girard's original work [22] and related calculi [1, 18]. This approach has the benefit of fully exposing the concurrency inherent in linear logic, and it takes full advantage of the symmetries of the logical connectives to provide a parsimonious syntax. Yet the resulting type systems and programming models, with their fully symmetric operations, are far removed from familiar functional programming languages.

A third approach studies *natural deduction* formulations of linear logic [10, 14], following work on term assignments for classical (though not linear) logic [35–37]. These calculi typically use typing judgments with multiple conclusions, which can be read computationally as assigning types to variables that name first-class continuations. Their operational semantics encode the so-called *commuting conversions* which shuffle (delimited) continuations in such a way as to effectively simulate parallel evaluation. This approach offers type systems that are relatively similar to those used in standard functional programming languages at the expense of obscuring the connections to concurrent programming.

Contributions This paper introduces Lolliproc, a language in the natural deduction tradition that takes a more direct approach to concurrency. Lolliproc is designed first as a core calculus for concurrent functional programming; it gives a Curry-Howard interpretation of *classical*—as opposed to intuitionistic—linear logic[1] that is nonetheless suggestive of familiar functional languages.

There are two key ideas to our approach. First, in contrast with the work mentioned previously, we move from an intuitionistic to a classical setting by adding a witness for double-negation elimi-

[1] Girard would say "full linear logic" or simply "linear logic".

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

nation, which we call **yield**. Second, to recover the expressiveness of linear logic, we introduce an operation **go**, which corresponds logically to the coercion from the intuitionistic negation ($\rho \multimap \bot$) to $\widetilde{\rho}$, ρ's dual as defined analogously to de Morgan's laws in classical logic. Operationally, **go** spawns a new process that executes in parallel to the main thread while **yield** waits for a value sent by another process. These constructs are novel adaptations of Felleisen & Hieb's **control** operator [17] to our linear setting.

The search for appropriate operational semantics for these constructs leads us to a simple process language—reminiscent of Milner's π-calculus [32]—hidden behind an abstract interface. Programs are written entirely in a standard linear λ-calculus augmented with the **go** and **yield** operations and elaborate to processes at run time. As a consequence, our type system isolates the classical multiple-conclusions judgments (captured by our typing rules for processes) so that they are not needed to type check source program expressions. This situation is somewhat analogous to how reference cells are treated in ML—location values and heap typings are needed to describe the operational semantics, but source program type checking doesn't require them.

Organization The next Section introduces Lolliproc informally, covering both what we take from the standard intuitionistic linear λ-calculus and our new constructs. Given our goal of enabling concurrent programming in a traditional functional setting, we demonstrate Lolliproc's functionality by example in Section 3; we show how a system that seems to permit communication in only one direction can in fact be used to mimic bidirectional session types.

Section 4 gives the formal typing rules and operational semantics for Lolliproc and presents our main technical contributions: a proof of *type soundness*, which implies both deadlock-freedom and adherence to session types; a proof of *strong normalization*, ruling out the possibility of livelocks or other non-terminating computations; and a proof of *confluence*, showing that there are no race conditions in our calculus.

Lolliproc does remain quite restricted, however—we have deliberately included only the bare minimum necessary to demonstrate its concurrent functionality. Section 5 discusses additions to the language that would relax these restrictions, including unrestricted (*i.e.*, non-linear) types, general recursion via recursive types, and intentional nondeterminism. This approach adheres to our philosophy of starting from a core language with support for well-behaved concurrency, then explicitly introducing potentially dangerous constructs (which, for instance, might introduce race conditions) in a controlled way. This section also concludes with a discussion of related work and a comparison of Lolliproc to more conventional classical linear logics.

2. An overview of Lolliproc

As shown in Figure 1, the types τ of Lolliproc include linear functions $\tau_1 \multimap \tau_2$, additive products $\tau_1 \,\&\, \tau_2$ (sometimes pronounced "with"), the unit type $\mathbf{1}$, multiplicative products $\tau_1 \otimes \tau_2$, and additive sums $\tau_1 \oplus \tau_2$. These types form an *intuitionistic* subset of linear logic, and they come equipped with standard introduction and elimination forms and accompanying typing rules. In addition, we have the type $\bot$, which is notably *not* the falsity from which everything follows.[2] Its purpose will become apparent later.

Our syntax for expressions is given by the grammar e in Figure 1, and their standard evaluation semantics is summarized in Figure 2.[3] In Lolliproc, all variables are treated linearly and functions

$$
\begin{array}{rcll}
\tau & ::= & \tau \multimap \tau \mid \tau \,\&\, \tau \mid \mathbf{1} \mid \tau \otimes \tau \mid \tau \oplus \tau \mid \bot & \textit{types} \\
\rho & ::= & \tau \multimap \rho \mid \rho \,\&\, \rho \mid \bot & \textit{protocol types} \\
\\
i & ::= & 1 \mid 2 & \textit{indices} \\
e & ::= & x \mid \lambda x{:}\tau.\, e \mid e\ e \mid \langle e, e\rangle \mid e.i & \textit{expressions} \\
 & \mid & () \mid e; e \mid (e, e) \mid \mathbf{in}_i^{\tau\oplus\tau}\ e & \\
 & \mid & \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e & \\
 & \mid & \mathbf{case}\ e\ \mathbf{of}\ \mathbf{in}_1\ x \mapsto e \mid \mathbf{in}_2\ y \mapsto e & \\
 & \mid & \mathbf{go}^{\rho}\ e \mid \mathbf{yield}\ e & \textit{new primitives} \\
 & \mid & \uparrow a \uparrow \mid \downarrow a \downarrow \mid \rfloor a \lfloor & \textit{channel endpoints} \\
\\
v & ::= & \lambda x{:}\tau.\, e \mid \langle e, e\rangle \mid () \mid (v, v) \mid \mathbf{in}_i^{\tau\oplus\tau}\ v & \textit{values} \\
 & \mid & \uparrow a \uparrow \mid \downarrow a \downarrow \mid \rfloor a \lfloor & \\
\\
E & ::= & [\,] \mid E\ e \mid v\ E \mid E.i & \textit{evaluation contexts} \\
 & \mid & E; e \mid (E, e) \mid (v, E) \mid \mathbf{in}_i^{\tau\oplus\tau}\ E & \\
 & \mid & \mathbf{let}\ (x, y) = E\ \mathbf{in}\ e & \\
 & \mid & \mathbf{case}\ E\ \mathbf{of}\ \mathbf{in}_1\ x \mapsto e \mid \mathbf{in}_2\ y \mapsto e & \\
 & \mid & \mathbf{go}^{\rho}\ E \mid \mathbf{yield}\ E & \\
\\
P & ::= & e \mid P \mid P \mid \nu a{:}\rho.\, P & \textit{processes} \\
\\
\Pi & ::= & \cdot \mid \Pi, a{\cdot}\rho \mid \Pi, a\tilde{\cdot}\rho \mid \Pi, a{:}\rho & \textit{channel contexts} \\
\Delta & ::= & \cdot \mid \Delta, x{:}\tau & \textit{typing contexts}
\end{array}
$$

Figure 1. Lolliproc syntax

$$
\begin{array}{c}
[\text{E-AppLam}]\ (\lambda x{:}\tau.\, e)\ v \longrightarrow \{x \mapsto v\}e \\
\\
[\text{E-LocalChoice}]\ \langle e_1, e_2\rangle.i \longrightarrow e_i \qquad [\text{E-Unit}]\ (); e \longrightarrow e \\
\\
[\text{E-Let}]\ \mathbf{let}\ (x_1, x_2) = (v_1, v_2)\ \mathbf{in}\ e \longrightarrow \{x_1 \mapsto v_1, x_2 \mapsto v_2\}e \\
\\
[\text{E-Case}]\ \mathbf{case}\ \mathbf{in}_i^{\tau_1\oplus\tau_2}\ v\ \mathbf{of}\ \mathbf{in}_1\ x_1 \mapsto e_1 \mid \mathbf{in}_2\ x_2 \mapsto e_2 \longrightarrow \{x_i \mapsto v\}e_i
\end{array}
$$

Figure 2. Basic evaluation rules

are call-by-value. Additive pairs $\langle e_1, e_2\rangle$ use the same resources to construct both of their components and are thus evaluated *lazily* and eliminated via projection; multiplicative pairs (e_1, e_2), whose components are independent, are evaluated *eagerly* and eliminated by **let**-binding both components. We use the sequencing notation $e_1; e_2$ to eliminate units $()$ of type $\mathbf{1}$. Additive sums, eliminated by **case** expressions, are completely standard.

Our new constructs—the **go** and **yield** operations, along with channels and processes—are perhaps best understood by looking at what motivated their design. In the rest of this section we will see how the desire to capture classicality led to processes with a simple communication model and how the desire to make that communication more express led back to classical linear logic. We will also see Lolliproc's operational semantics; we defer a full account of its typing rules for Section 4.

2.1 Moving to classical linear logic

The differences between intuitionistic and classical logic can be seen in their treatment of negation and disjunction. In standard presentations of classical linear logic, negation is defined via a *dualizing operator* $(-)^{\bot}$ that identifies the de Morgan duals as

[2] Such a type in linear logic is the additive false, while $\bot$ is the multiplicative false; we have left additive units out of Lolliproc for simplicity's sake.

[3] The typical rule for handling evaluation contexts is missing, as this is done at the process level in Lolliproc.

shown below:

$$\begin{array}{rclrcl} \bot^\bot & = & \mathbf{1} & \mathbf{1}^\bot & = & \bot \\ (t_1 \,\&\, t_2)^\bot & = & t_1^\bot \oplus t_2^\bot & (t_1 \oplus t_2)^\bot & = & t_1^\bot \,\&\, t_2^\bot \\ (t_1 \multimap t_2)^\bot & = & t_1 \otimes t_2^\bot & (t_1 \otimes t_2)^\bot & = & t_1 \multimap t_2^\bot \end{array}$$

With this definition, dualization is clearly an involution—that is, $(\tau^\bot)^\bot = \tau$. Moreover, the logic is set up so that duals are logically equivalent to negation: $\tau^\bot$ is provable if and only if $\tau \multimap \bot$ is provable. In this way, classical linear logic builds *double-negation elimination* into its very definition—it is trivial to prove the theorem $((\tau \multimap \bot) \multimap \bot) \multimap \tau$, which is not intuitionistically valid.

Sequent calculus formulations of classical linear logic take advantage of these dualities by observing that the introduction of τ is equivalent to the elimination of $\tau^\bot$; this allows them to be presented with half the typing rules and syntactic forms that would otherwise be required. This symmetric approach is extremely convenient for proof theory but does not allow us to conservatively extend the existing typing rules and operational semantics for the intuitionistic fragment of linear logic already described above. For that, we need a natural-deduction formulation of the type system.

Our solution to this problem is to forget dualization (for now) and instead add double-negation elimination as a primitive. We take inspiration from type systems for Felleisen & Hieb's **control** and **abort** operators [17, 34]: in a non-linear setting, **control**, can be given the type $((\tau \to \bot) \to \bot) \to \tau$, corresponds to double-negation elimination, while **abort** is a functional variant of false elimination that takes $\bot$ to any type. The operational behavior of these constructs is as follows:

$$\begin{array}{rcl} E\big[\mathbf{control}\ (\lambda c.\ e)\big] & \longrightarrow & (\lambda c.\ e)\ (\lambda x.\ \mathbf{abort}\ E\big[x\big]) \\ E\big[\mathbf{abort}\ e\big] & \longrightarrow & e \end{array}$$

Unfortunately, **abort** clearly has no place in a linear system, as it discards evaluation context E and any resources contained therein. What can we do instead? Observe that c has the continuation type $\tau \to \bot$ (or, in a linear setting, $\tau \multimap \bot$) and that invoking c within the body e returns an "answer" to the context E. We can reconcile this behavior with a linear system by dropping **abort** and instead introducing the ability to evaluate two expression in parallel:

$$E\big[\mathbf{control}\ (\lambda c.\ e)\big] \longrightarrow E\big[\mathbf{control}\ \uparrow a \uparrow\big] \parallel (\lambda c.\ e)\ \downarrow a \downarrow$$

Here, evaluating a **control** expression spawns its argument as a child process. The connection between the original evaluation context E and the child process is now the channel a: we write $\uparrow a \uparrow$ for the receiving endpoint or *source* of a, held by the parent process, while the $\downarrow a \downarrow$ passed to the child denotes the sending endpoint or *sink*. Now evaluation can proceed in the right-hand expression until the sink is applied to a value, at which point this "answer" is passed back to the parent process:

$$E\big[\mathbf{control}\ \uparrow a \uparrow\big] \parallel E'\big[\downarrow a \downarrow v\big] \longrightarrow E\big[v\big] \parallel E'\big[\lfloor a \rfloor\big]$$

The closed channel token $\lfloor a \rfloor$ indicates that communication over a is finished; it also indicates that the child process may now terminate, but before process termination actually happens all linear resources in E' must be safely consumed. Linearity is preserved by both of our operations, as neither expressions nor evaluation contexts are duplicated or discarded.

So far, though, these constructs offer a very poor form of concurrency—in the rules above, the parent process immediately blocks waiting for the child process to return. To allow the parent and child to execute in parallel, we split can **control** into two operations. The first, which we call **go**, is responsible for generating the channel a and spawning the child process; it immediately returns a source value to the parent, which can keep running:

$$E\big[\mathbf{go}\ (\lambda c.\ e)\big] \longrightarrow E\big[\uparrow a \uparrow\big] \parallel (\lambda c.\ e)\ \downarrow a \downarrow$$

The second operation, **yield**, is used by the parent process to synchronize with the child by blocking on a source:

$$E\big[\mathbf{yield}\ \uparrow a \uparrow\big] \parallel E'\big[\downarrow a \downarrow v\big] \longrightarrow E\big[v\big] \parallel E'\big[\lfloor a \rfloor\big]$$

2.2 Typing and extending go and yield

How, then, to type check these new operations? Which is to say, what is their logical meaning?

The source $\uparrow a \uparrow$ has type $((\tau \multimap \bot) \multimap \bot)$, and such doubly-negated types appear so frequently in Lolliproc that we abbreviate them as $\uparrow \tau \uparrow$, pronounced "source of τ". Invoking **yield** on such a source returns a τ—it eliminates the double negation—so we have:

$$\mathbf{yield} \quad : \quad \uparrow \tau \uparrow \multimap \tau$$

What about **go**? At first glance, it appears that **go** takes an expression of type $\uparrow \tau \uparrow$ and returns a $\uparrow \tau \uparrow$—it is logically an identity function. This would be sound, but we can do better. The type $\tau \multimap \bot$ is usually thought of as a continuation that accepts a τ, but here it is better to think of it as expressing a very simple *protocol*, one in which a τ is sent and there is no further communication. From this point of view, we can instead think of **go** as taking a function of type $\rho \multimap \bot$, and spawning that function as a child process that must communicate according to the protocol ρ. The parent process receives from **go** a source whose type describes the other side of the protocol ρ; hence a **yield** on the source waits for information to be sent across the sink by the child process, after which both sides continue with the protocol.

Which types make sense as protocols? A protocol might be complete (*i.e.*, $\bot$), it might specify that a value of type τ be sent before continuing according to the protocol ρ (*i.e.*, $\tau \multimap \rho$), or it might specify a choice between protocols ρ_1 and ρ_2 (*i.e.*, $\rho_1 \,\&\, \rho_2$). For each such protocol type ρ we define a dual type $\widetilde{\rho}$, as follows:[4]

$$\begin{array}{rcl} \widetilde{\bot} & = & \mathbf{1} \\ \widetilde{\rho_1 \,\&\, \rho_2} & = & \uparrow \widetilde{\rho_1} \oplus \widetilde{\rho_2} \uparrow \\ \widetilde{\tau \multimap \rho} & = & \uparrow \tau \otimes \widetilde{\rho} \uparrow \end{array}$$

Aside from the extra double-negations—corresponding operationally to points at which we must synchronize with **yield** and logically to explicitly marking where classical reasoning will take place—this is exactly the left-hand column of the definition of $(-)^\bot$.[5] Additionally, since $\uparrow \tau \uparrow$ is defined in terms of implication, both $\uparrow \widetilde{\rho_1} \oplus \widetilde{\rho_2} \uparrow$ and $\uparrow \tau \otimes \widetilde{\rho} \uparrow$ are themselves protocol types, a fact which will become important as we go on.

Thus **go** witnesses the logical isomorphism between the intuitionistic negation of a type and its dual:

$$\mathbf{go} \quad : \quad (\rho \multimap \bot) \multimap \widetilde{\rho}$$

The channel endpoints $\downarrow a \downarrow$ and $\uparrow a \uparrow$, then, must have the types ρ and $\widetilde{\rho}$. Their types will change over the course of evaluation, as communication proceeds over the channel a; when communication is finished, the $\downarrow a \downarrow$ of type $\bot$ will be replaced by $\lfloor a \rfloor$ of that same type, while the $\uparrow a \uparrow$ of type $\mathbf{1}$ will simply step to $()$.

With this plumbing in place, we can define our operational semantics for processes as shown in Figure 3. At the process level we bind channels with $\nu a{:}\rho.\ P$; these binders are generated by rule EP-Go and require that we annotate **go** expressions as $\mathbf{go}^\rho\ e$. Evaluation blocks when **yield**ing on sources or eliminating sinks

[4] The choice to define $\widetilde{\bot}$ as $\mathbf{1}$ rather than $\uparrow \mathbf{1} \uparrow$ is a simple optimization that saves us from unnecessary synchronization at channel shutdown; our link^τ example in the next section shows how this can come in handy.

[5] In linear logic, the protocol connectives are said to be *negative*, meaning that their introduction forms are invertible. That is, no additional choice is made in their construction—in contrast to the choice of injection for $\oplus$ and the choice of resource split for $\otimes$, which are both *positive* connectives.

$$\text{[EP-Go]}\ \frac{a \text{ not free in } E\big[\mathbf{go}^{\rho}\ v\big]}{E\big[\mathbf{go}^{\rho}\ v\big] \longrightarrow \nu a{:}\rho.\ (E\big[\uparrow a\uparrow\big] \mid v\ \downarrow a\downarrow)}$$

$$\text{[EP-AppSink]}\ \nu a{:}\tau \multimap \rho.\ E_1\big[\mathbf{yield}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow\ v\big] \longrightarrow \nu a{:}\rho.\ E_1\big[(v, \uparrow a\uparrow)\big] \mid E_2\big[\downarrow a\downarrow\big]$$

$$\text{[EP-RemoteChoice]}\ \nu a{:}\rho_1\ \&\ \rho_2.\ E_1\big[\mathbf{yield}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow.i\big] \longrightarrow \nu a{:}\rho_i.\ E_1\big[\mathbf{in}_i^{\rho_1 \oplus \rho_2}\ \uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow\big]$$

$$\text{[EP-Close]}\ \nu a{:}\bot.\ E_1\big[\uparrow a\uparrow\big] \mid E_2\big[\downarrow a\downarrow\big] \longrightarrow E_1\big[()\big] \mid \nu a{:}\bot.\ E_2\big[\downarrow a\downarrow\big] \qquad \text{[EP-Done]}\ P \mid \nu a{:}\bot.\ \downarrow a\downarrow \longrightarrow P$$

$$\text{[EP-Eval]}\ \frac{e \longrightarrow e'}{E\big[e\big] \longrightarrow E\big[e'\big]} \qquad \text{[EP-Par]}\ \frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad \text{[EP-New]}\ \frac{P \longrightarrow P'}{\nu a{:}\tau.\ P \longrightarrow \nu a{:}\tau.\ P'}$$

Figure 3. Process evaluation rules

$$\text{[E-YieldOther]}\ \frac{v \neq \uparrow a\uparrow}{\mathbf{yield}\ v \longrightarrow \mathbf{let}\ (z, u) = \mathbf{yield}\ (\mathbf{go}^{\tau \multimap \bot}\ v)\ \mathbf{in}\ u; z} \qquad \text{[E-AppSource]}\ \uparrow a\uparrow\ v \longrightarrow v\ (\mathbf{yield}\ \uparrow a\uparrow)$$

Figure 4. Expression congruence rules

until a matching pair is in play, at which point the argument or choice bit is relayed across the channel (rules EP-AppSink and EP-RemoteChoice). Note that such communication has the effect of updating the type of the channel at its binding site to reflect the new state of the protocol. The rule EP-Close is similar, but exists only to facilitate typing of completed channels and thus does not require a **yield**. EP-Done eliminates completed processes (reminiscent of **0** in the π-calculus) and their binders. EP-Eval integrates evaluation contexts and expression evaluation with process evaluation, while EP-Par and EP-New allow evaluation within processes. (We also define the standard notion of process equivalence, given in Section 4.)

Two final points must be addressed by operational semantics: the type $\uparrow\tau\uparrow$ can be inhabited by more than just sources, and thus we need evaluation rules for **yield**ing on other sorts of values; similarly, our sources all technically have function types, so we must be able to apply them. Figure 4 gives the appropriate congruence rules. For the first case, we recall our earlier intuition concerning the simpler (but less useful) language where **yield** and **go** are combined into **control**. Rule E-YieldOther thus synthesizes a **go** in such cases, although we must also synthesize a **let** binding, as we have transformed a value of type $\uparrow\tau\uparrow$ into one of type $\uparrow\tau \otimes \mathbf{1}\uparrow$.

When a source appears in the function position of an application, we appeal to the intuition from other systems for classical logics [22, 35] that the interaction of a term with type τ and another with type $\tau^{\bot}$ should not depend on the order of those terms. Thus, applying $\uparrow a\uparrow$ of type $(\tau \multimap \bot) \multimap \bot$ to v of type $\tau \multimap \bot$ should be the equivalent of first **yield**ing on $\uparrow a\uparrow$, then supplying the result to v. Rule E-AppSource makes this so, and it is easy to verify that this property also holds in the case of other applications at those types.

Although these congruence rules are a bit unusual, the fact that Lolliproc does not introduce a new family of types for channel endpoints turns out to be a very useful property of the system: for instance, it allows us to bootstrap bidirectional communication from what appears, at first glance, to be a unidirectional language. We will see how this transpires in the next section.

3. Examples

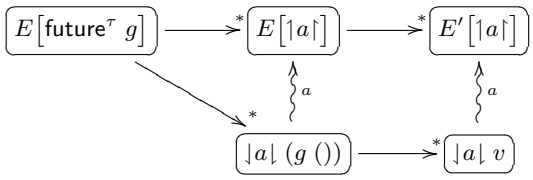
Here we demonstrate some of what can be done with Lolliproc by introducing several concurrency routines of increasing complexity. For ease of explanation and consistency, we write foo^{τ} when the function foo is parameterized by the type τ, and we use capitalized type abbreviations, *e.g.*, $\mathsf{Bar}\ \tau$. In a real language we would of course want polymorphism—either ML-style or the full generality of System F with linearity [31].

Futures A *future* [33] is simply a sub-computation to be calculated in a separate thread; the main computation will wait for this thread to complete when its value is needed. This is one of the simplest forms of concurrency expressible in Lolliproc. We can define

$$\begin{array}{rcl} \mathsf{Future}\ \tau & = & \uparrow\tau \otimes \mathbf{1}\uparrow \\ \\ \mathsf{future}^{\tau} & : & (\mathbf{1} \multimap \tau) \multimap \mathsf{Future}\ \tau \\ \mathsf{future}^{\tau} & = & \lambda x{:}\mathbf{1} \multimap \tau.\ \mathbf{go}^{\tau \multimap \bot}\ \lambda k{:}\tau \multimap \bot.\ k\ (x\ ()) \\ \\ \mathsf{wait}^{\tau} & : & \mathsf{Future}\ \tau \multimap \tau \\ \mathsf{wait}^{\tau} & = & \lambda f{:}\mathsf{Future}\ \tau.\ \mathbf{let}\ (z, u) = \mathbf{yield}\ f\ \mathbf{in}\ u; z \end{array}$$

The main process passes a thunk to its newly spawned child; this child applies the thunk and sends back the result.

More pictorially, the run-time behavior of $E\big[\mathsf{future}^{\tau}\ g\big]$, where $g\ () \longrightarrow^{*} v$ and $E\big[-\big] \longrightarrow^{*} E'\big[-\big]$, is



The connection between endpoints of a channel at a given moment in time are given by $\rightsquigarrow$ arrows. Similarly, for such some $\uparrow a\uparrow$ of type

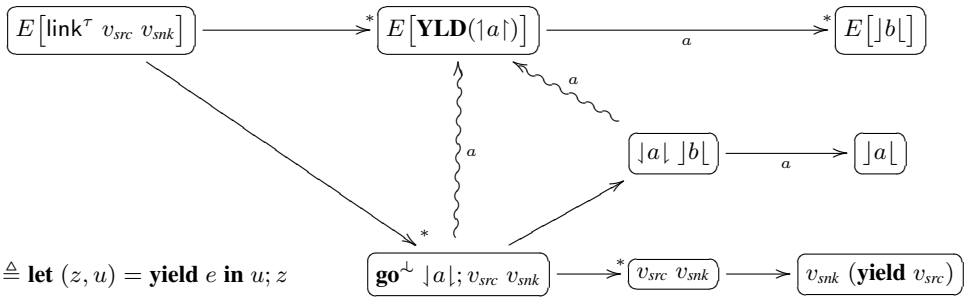
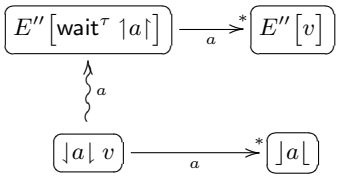


Figure 5. Evaluation of $\mathsf{link}^\tau\ v_{src}\ v_{snk}$

Future τ, we have



Here the a subscript on evaluation arrows indicates that communication over a has occurred. Since a supports no further communication afterwards—its sink has been replaced by the closed channel token $\lfloor a\rfloor$—the $\rightsquigarrow$ connection is then removed. Recall that such a lone $\lfloor a\rfloor$ indicates a completed process; the child process in this example is now complete and will disappear.

Linking channel endpoints Given a v_{src} of type $\uparrow\tau\uparrow$ and v_{snk} of type $\tau \multimap \perp$—which may or may not be a literal source and sink—we might want to join the two such that v_{src} flows to v_{snk} without making the parent process wait by **yield**ing on v_{src}. In doing so, however, we must still somehow produce a value of type $\perp$; it can't be the value that applying v_{snk} would produce, so it must come from some other process.

Our solution relies on the ability to pass process completion tokens from one process to another:

$$\begin{array}{lcl} \mathsf{link}^\tau & : & \uparrow\tau\uparrow \multimap (\tau \multimap \perp) \multimap \perp \\ \mathsf{link}^\tau & = & \lambda x{:}\uparrow\tau\uparrow.\ \lambda f{:}\tau \multimap \perp.\ \mathbf{yield}\ \lambda g{:}\perp \multimap \perp.\ \mathbf{go}^{\perp}\ g; x\ f \end{array}$$

Note that the final $x\ \ f$ will step to $f\ \ (\mathbf{yield}\ \ x)$ via rule E-AppSource; similarly, rule E-YieldOther will insert a $\mathbf{go}^{\perp\multimap\perp}$ immediately following the **yield**. A call to $\mathsf{link}^\tau\ \ v_{src}\ \ v_{snk}$ thus spawns two processes: the first spawns the second with the trivial protocol, then proceeds to wait and link the original arguments; the second uses the sink created for the first child to immediately return control to the parent process. This is illustrated in Figure 5; we use the abbreviation $\mathbf{YLD}(e)$ for the now common pattern of **yield**ing to receive a product, immediately unpacking the resulting pair, and eliminating the left component.

Reversing directions So far we have seen only child processes that send information back to their parents. While our constructs show bias towards this sort of communication, Lolliproc does allow exchanges in both directions; a few complications arise, however, due to the unidirectional nature of our so-called dualization.

For instance, while the dual of $\tau \multimap \rho$ is $\uparrow\tau \otimes \widetilde{\rho}\uparrow$, the dual of $\uparrow\tau\otimes\rho\uparrow$ is the somewhat unwieldy $\uparrow((\tau\otimes\rho) \multimap \perp)\otimes\mathbf{1}\uparrow$ rather than the $\tau \multimap \widetilde{\rho}$ for which we would have hoped. Yet we observe that the former can be transformed into the latter with a **yield** operation, an uncurrying, a partial application, and a $\mathbf{go}^i$ we combine these steps into a function send:

$$\begin{array}{lcl} \mathsf{send}^{\tau\multimap\widetilde{\rho}} & : & \widetilde{\uparrow\tau\otimes\rho\uparrow} \multimap \tau \multimap \widetilde{\rho} \\ \mathsf{send}^{\tau\multimap\widetilde{\rho}} & = & \lambda s{:}\widetilde{\uparrow\tau\otimes\rho\uparrow}. \\ & & \quad \mathbf{let}\ (f,u) = \mathbf{yield}\ s\ \mathbf{in} \\ & & \quad\ u; \lambda x{:}\tau.\ \mathbf{go}^{\rho}\ \lambda p{:}\rho.\ f\ (x,p) \end{array}$$

Similarly, the dual of $\uparrow\rho_1 \oplus \rho_2\uparrow$ is $\uparrow((\rho_1 \oplus \rho_2) \multimap \perp) \otimes \mathbf{1}\uparrow$; to coerce this to $\widetilde{\rho_1} \,\&\, \widetilde{\rho_2}$, we define select as

$$\begin{array}{lcl} \mathsf{select}^{\widetilde{\rho_1}\&\widetilde{\rho_2}} & : & \widetilde{\uparrow\rho_1 \oplus \rho_2\uparrow} \multimap \widetilde{\rho_1} \,\&\, \widetilde{\rho_2} \\ \mathsf{select}^{\widetilde{\rho_1}\&\widetilde{\rho_2}} & = & \lambda s{:}\widetilde{\uparrow\rho_1 \oplus \rho_2\uparrow}. \\ & & \quad \mathbf{let}\ (f,u) = \mathbf{yield}\ s\ \mathbf{in} \\ & & \quad\ u; \langle \mathbf{go}^{\rho_1}\ \lambda p_1{:}\rho_1.\ f\ \mathbf{in}_1^{\rho_1\oplus\rho_2}\ p_1, \\ & & \quad\quad \mathbf{go}^{\rho_2}\ \lambda p_2{:}\rho_2.\ f\ \mathbf{in}_2^{\rho_1\oplus\rho_2}\ p_2\rangle \end{array}$$

To demonstrate the first of these coercions in action, we look to the identity function echo, which spawns a child process, passes its argument to that child, then receives it back:

$$\begin{array}{lcl} \mathsf{reply}^\tau & : & \uparrow\tau \otimes (\tau \multimap \perp)\uparrow \multimap \perp \\ \mathsf{reply}^\tau & = & \lambda h{:}\uparrow\tau \otimes (\tau \multimap \perp)\uparrow.\ \mathbf{let}\ (y,g) = \mathbf{yield}\ h\ \mathbf{in}\ g\ y \end{array}$$

$$\begin{array}{lcl} \mathsf{echo}^\tau & : & \tau \multimap \tau \\ \mathsf{echo}^\tau & = & \lambda x{:}\tau.\ \mathbf{let}\ (z,u) = \mathbf{yield} \\ & & \qquad \mathsf{send}^{\tau\multimap\uparrow\tau\otimes\mathbf{1}\uparrow}\ (\mathbf{go}^{\uparrow\tau\otimes(\tau\multimap\perp)\uparrow}\ \mathsf{reply}^\tau)\ x \\ & & \quad \mathbf{in}\ u; z \end{array}$$

Here reply is the body of the child process that will receive the initial argument and send it back. (The type of reply^τ could equally well have been written as the equivalent $\uparrow\tau \otimes (\tau \multimap \perp) \multimap \perp\uparrow$—this notation better reflects how it is used with echo, while the notation given above more closely matches its definition.)

The execution of $\mathsf{echo}^\tau\ \ v$ for some v of type τ is shown in Figure 6. We can see how, while the initial spawning of the reply^τ process orients the channel a in the usual child-to-parent direction, the machinery of send spawns another process that sets up a channel b in the opposite direction; afterwards, a third channel c is established in the original direction. All this is facilitated again by our congruence rules.

It is worth noting that, while the value v cycles among several processes, at no point does a cycle exist in the communication structure—the $\rightsquigarrow$ arrows—of Figure 6. That this fact always holds is crucial to our proof of soundness in Section 4.

A larger example So far we have seen relatively small examples. As a larger demonstration of the protocols expressible in Lolliproc, we consider Diffie-Hellman key exchange, formulated as follows:

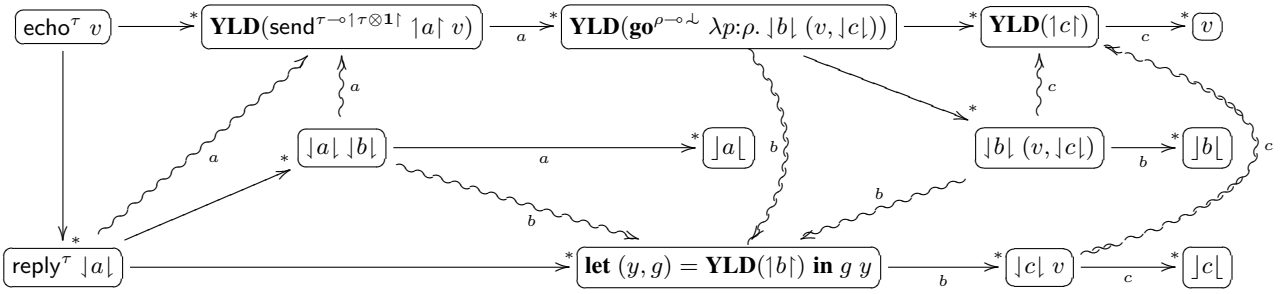


Figure 6. Evaluation of $\mathsf{echo}^\tau\ v$

1. Alice and Bob select secret integers a and b.
2. Alice and Bob exchange $g^a \bmod p$ and $g^b \bmod p$ in the clear.
3. Alice and Bob compute the shared secret $(g^b)^a = (g^a)^b \bmod p$ and use it to encrypt further communication.

Here g is a publicly known generator with certain properties, often 2 or 5, and p is a similarly known large prime number. The shared secret cannot feasibly be computed from the publicly known values g^a and g^b. For purposes of this example, we declare that further communication consists only of Alice sending an encrypted string to Bob, and we treat Alice's session as a child process spawned by Bob rather than as a process somewhere over the network that initiates contact. We augment Lolliproc with the types Int and String, as well as necessary operations over these types:

$$\begin{array}{rcl} \mathsf{bigrandom} & : & \mathbf{1} \multimap \mathsf{Int} \\ \mathsf{powmod} & : & \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int} \\ \mathsf{lessthan} & : & \mathsf{Int} \multimap \mathsf{Int} \multimap (\mathbf{1} \oplus \mathbf{1}) \\ \mathsf{encrypt} & : & \mathsf{Int} \multimap \mathsf{String} \multimap \mathsf{String} \\ \mathsf{decrypt} & : & \mathsf{Int} \multimap \mathsf{String} \multimap \mathsf{String} \end{array}$$

For clarity, we also freely use general **let** expressions rather than only those that eliminate multiplicative products, and we allow the reuse of variables of type Int.

To demonstrate the use of additive products and sums—and to add a hint of realism—we allow Alice or Bob to abort the session after receiving a value from the other party. Thus the protocol type that must be enforced in Alice's session and a sample implementation of said session are

$$\mathsf{Alice} = \mathsf{Int} \multimap \uparrow\bot \oplus \uparrow\mathsf{Int} \otimes (\bot\ \&\ (\mathsf{String} \multimap \bot))\uparrow\uparrow$$

$$\begin{array}{rcl} \mathsf{alice} & : & \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Alice} \multimap \bot \\ \mathsf{alice} & = & \lambda g{:}\mathsf{Int}.\ \lambda p{:}\mathsf{Int}.\ \lambda n{:}\mathsf{Int}.\ \lambda s{:}\mathsf{Alice}. \\ & & \quad \textbf{let}\ a = \mathsf{bigrandom}\ ()\ \textbf{in} \\ & & \quad \textbf{case yield}\ (s\ (\mathsf{powmod}\ g\ a\ p))\ \textbf{of} \\ & & \quad\quad \textbf{in}_1\ s_1 \mapsto s_1 \\ & & \quad\quad \mid \textbf{in}_2\ s_2 \mapsto \textbf{let}\ (b, s') = \textbf{yield}\ s_2\ \textbf{in} \\ & & \quad\quad\quad \textbf{case}\ \mathsf{lessthan}\ b\ n\ \textbf{of} \\ & & \quad\quad\quad\quad \textbf{in}_1\ u_1 \mapsto u_1; s'.1 \\ & & \quad\quad\quad\quad \mid \textbf{in}_2\ u_2 \mapsto u_2; \textbf{let}\ k = \mathsf{powmod}\ b\ a\ p\ \textbf{in} \\ & & \quad\quad\quad\quad\quad (s'.2)\ (\mathsf{encrypt}\ k\ \texttt{"I know secrets!"}) \end{array}$$

Since Alice's session is the child process, the point at which she must check for an abort signal from Bob appear as $\bot \oplus \rho$, while the point at which she may abort appears as $\bot\ \&\ \rho$. In this case, Alice chooses to abort whenever the public key Bob sends her is too small in comparison to some parameter n.

An implementation of Bob's side of the communication—*i.e.*, the parent process—looks very similar. While bob relies on the type Alice to specify the whole communication protocol, we do need type annotations B1 and B2 for our uses of send and select.

$$\begin{array}{rcl} \mathsf{B1} & = & \mathbf{1}\ \&\ \uparrow((\mathsf{Int} \otimes (\bot\ \&\ (\mathsf{String} \multimap \bot))) \multimap \bot) \otimes \mathbf{1}\uparrow \\ \mathsf{B2} & = & \mathsf{Int} \multimap \uparrow\mathbf{1} \oplus (\mathsf{String} \multimap \bot)\uparrow \end{array}$$

$$\begin{array}{rcl} \mathsf{bob} & : & \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{Int} \multimap \mathsf{String} \\ \mathsf{bob} & = & \lambda g{:}\mathsf{Int}.\ \lambda p{:}\mathsf{Int}.\ \lambda n{:}\mathsf{Int}. \\ & & \quad \textbf{let}\ (a, s) = \textbf{yield}\ (\mathbf{go}^{\mathsf{Alice}}\ (\mathsf{alice}\ g\ p\ n))\ \textbf{in} \\ & & \quad \textbf{case}\ \mathsf{lessthan}\ a\ n\ \textbf{of} \\ & & \quad\quad \textbf{in}_1\ u_1 \mapsto u_1; (\mathsf{select}^{\mathsf{B1}}\ s).1; \texttt{"ERROR1"} \\ & & \quad\quad \mid \textbf{in}_2\ u_2 \mapsto u_2; \textbf{let}\ s_1 = (\mathsf{select}^{\mathsf{B1}}\ s).2\ \textbf{in} \\ & & \quad\quad\quad \textbf{let}\ b = \mathsf{bigrandom}\ b\ \textbf{in} \\ & & \quad\quad\quad \textbf{let}\ s_2 = \mathsf{send}^{\mathsf{B2}}\ s_1\ (\mathsf{powmod}\ g\ b\ p)\ \textbf{in} \\ & & \quad\quad\quad \textbf{case yield}\ s'\ \textbf{of} \\ & & \quad\quad\quad\quad \textbf{in}_1\ u \mapsto u; \texttt{"ERROR2"} \\ & & \quad\quad\quad\quad \mid \textbf{in}_2\ s'' \mapsto \textbf{let}\ k = \mathsf{powmod}\ a\ b\ p\ \textbf{in} \\ & & \quad\quad\quad\quad\quad \textbf{let}\ (c, u') = \textbf{yield}\ s''\ \textbf{in} \\ & & \quad\quad\quad\quad\quad u'; \mathsf{decrypt}\ k\ c \end{array}$$

For brevity, we do not illustrate an evaluation of bob $g\ p\ n$. We observe, however, that nothing new is going on in this example as compared to echo^τ. We also observe that the definitions of alice and bob are relatively straightforward. They could be improved by standard type inference and by syntactic sugar that gave the repeated generation and consumption of linear variables the appearance of a single variable being mutated [31], but they are generally quite readable.

4. Metatheory

We now discuss the technical aspects of Lolliproc, including the formal proofs of soundness, strong normalization, and confluence.

4.1 Typing

The expression typing rules for Lolliproc can be seen in Figure 7. As we discussed in the introduction, these typing rules follow the natural-deduction presentation of intuitionistic linear calculi. Our typing judgment $\Pi; \Delta \vdash e : \tau$ depends both on a channel context Π and a term variable context Δ. Term variables x are bound to types τ in Δ, while Π contains binders $a{\cdot}\rho$ (representing the ability

$$[\text{T-Unit}]\ \cdot;\cdot \vdash () : \mathbf{1} \qquad [\text{T-Seq}]\ \frac{\Pi_1;\Delta_1 \vdash e_1 : \mathbf{1} \qquad \Pi_2;\Delta_2 \vdash e_2 : \tau}{\Pi_1 \Cup \Pi_2;\Delta_1 \Cup \Delta_2 \vdash e_1;e_2 : \tau} \qquad [\text{T-Var}]\ \cdot;x{:}\tau \vdash x : \tau$$

$$[\text{T-Lam}]\ \frac{\Pi;\Delta,x{:}\tau_1 \vdash e : \tau_2}{\Pi;\Delta \vdash \lambda x{:}\tau_1.\, e : \tau_1 \multimap \tau_2} \qquad [\text{T-App}]\ \frac{\Pi_1;\Delta_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Pi_2;\Delta_2 \vdash e_2 : \tau_1}{\Pi_1 \Cup \Pi_2;\Delta_2 \Cup \Delta_2 \vdash e_1\ e_2 : \tau_2} \qquad [\text{T-Go}]\ \frac{\Pi;\Delta \vdash e : \rho \multimap \bot}{\Pi;\Delta \vdash \mathbf{go}^{\rho}\ e : \widetilde{\rho}}$$

$$[\text{T-With}]\ \frac{\Pi;\Delta \vdash e_1 : \tau_1 \qquad \Pi;\Delta \vdash e_2 : \tau_2}{\Pi;\Delta \vdash \langle e_1,e_2\rangle : \tau_1 \,\&\, \tau_2} \qquad [\text{T-Select}]\ \frac{\Pi;\Delta \vdash e : \tau_1 \,\&\, \tau_2}{\Pi;\Delta \vdash e.i : \tau_i} \qquad [\text{T-Yield}]\ \frac{\Pi;\Delta \vdash e : \uparrow\tau\uparrow}{\Pi;\Delta \vdash \mathbf{yield}\ e : \tau}$$

$$[\text{T-Tensor}]\ \frac{\Pi_1;\Delta_1 \vdash e_1 : \tau_1 \qquad \Pi_2;\Delta_2 \vdash e_2 : \tau_2}{\Pi_1 \Cup \Pi_2;\Delta_1 \Cup \Delta_2 \vdash (e_1,e_2) : \tau_1 \otimes \tau_2} \qquad [\text{T-Let}]\ \frac{\Pi_1;\Delta_2 \vdash e' : \tau_1 \otimes \tau_2 \qquad \Pi_2;\Delta_2,x_1{:}\tau_1,x_2{:}\tau_2 \vdash e : \tau}{\Pi_1 \Cup \Pi_2;\Delta_1 \Cup \Delta_2 \vdash \mathbf{let}\ (x_1,x_2) = e'\ \mathbf{in}\ e : \tau}$$

$$[\text{T-In}]\ \frac{\Pi;\Delta \vdash e : \tau_i}{\Pi;\Delta \vdash \mathbf{in}_i^{\tau_1 \oplus \tau_2}\ e : \tau_1 \oplus \tau_2} \qquad [\text{T-Case}]\ \frac{\Pi_1;\Delta_1 \vdash e' : \tau_1 \oplus \tau_2 \qquad \Pi_2;\Delta_2,x_1{:}\tau_1 \vdash e_1 : \tau \qquad \Pi_2;\Delta_2,x_2{:}\tau_2 \vdash e_2 : \tau}{\Pi_1 \Cup \Pi_2;\Delta_1 \Cup \Delta_2 \vdash \mathbf{case}\ e'\ \mathbf{of}\ \mathbf{in}_1\ x_1 \mapsto e_1 \mid \mathbf{in}_2\ x_2 \mapsto e_2}$$

$$[\text{T-Sink}]\ a{\cdot}\rho;\cdot \vdash \lfloor a \rfloor : \rho \qquad [\text{T-Source}]\ \tilde{a{\cdot}\rho};\cdot \vdash \lceil a \rceil : \widetilde{\rho} \qquad [\text{TR-Done}]\ a{:}\bot;\cdot \vdash \rfloor a \lfloor : \bot$$

Figure 7. Expression typing rules

$$[\text{U-Empty}]\ \cdot \Cup \cdot = \cdot \qquad [\text{UT-Left}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \mathrm{dom}(\Delta)}{\Delta_1,x{:}\tau \Cup \Delta_2 = \Delta,x{:}\tau} \qquad [\text{UT-Right}]\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \mathrm{dom}(\Delta)}{\Delta_1 \Cup \Delta_2,x{:}\tau = \Delta,x{:}\tau}$$

$$\begin{array}{rcl} \S & ::= & \cdot \mid \tilde{\cdot} \mid : \\ \Ddot{\Cup} & ::= & \Cup \mid \widehat{\Cup} \end{array} \qquad [\text{UC-Left}]\ \frac{\Pi_1 \Ddot{\Cup} \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1,a\S\rho \Ddot{\Cup} \Pi_2 = \Pi,a\S\rho} \qquad [\text{UC-Right}]\ \frac{\Pi_1 \Ddot{\Cup} \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1 \Ddot{\Cup} \Pi_2,a\S\rho = \Pi,a\S\rho}$$

$$[\text{UC-None}]\ \frac{\Pi_1 \Cup \Pi_2 = \Pi}{\Pi_1 \widehat{\Cup} \Pi_2 = \Pi} \qquad [\text{UC-SrcSnk}]\ \frac{\Pi_1 \Cup \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1,\tilde{a{\cdot}\rho} \widehat{\Cup} \Pi_2,a{\cdot}\rho = \Pi,a{:}\rho} \qquad [\text{UC-SnkSrc}]\ \frac{\Pi_1 \Cup \Pi_2 = \Pi \qquad a \notin \mathrm{dom}(\Pi)}{\Pi_1,a{\cdot}\rho \widehat{\Cup} \Pi_2,\tilde{a{\cdot}\rho} = \Pi,a{:}\rho}$$

Figure 8. Context splitting rules

$$[\text{TP-Exp}]\ \frac{\Pi;\cdot \vdash e : \tau}{\Pi \vdash e : \tau} \qquad [\text{TP-New}]\ \frac{\Pi,a{:}\rho \vdash P : \tau}{\Pi \vdash \nu a{:}\rho.\, P : \tau}$$

$$[\text{TP-ParLeft}]\ \frac{\Pi_1 \vdash P_1 : \tau \qquad \Pi_2 \vdash P_2 : \bot}{\Pi_1 \widehat{\Cup} \Pi_2 \vdash P_1 \mid P_2 : \tau}$$

$$[\text{TP-ParRight}]\ \frac{\Pi_1 \vdash P_1 : \bot \qquad \Pi_2 \vdash P_2 : \tau}{\Pi_1 \widehat{\Cup} \Pi_2 \vdash P_1 \mid P_2 : \tau}$$

Figure 9. Process typing rules

to send on the channel a), $\tilde{a{\cdot}\rho}$ (representing the ability to receive on a), and $a{:}\rho$ (combining both capabilities). Both varieties of context are linear, in the sense that they permit neither weakening nor contraction.

Many of our rules are standard for a linear type system, but as linear type systems themselves are not quite standard, they still deserve some explanation. Because linear variables cannot be discarded, rules that serve as the leaves of proof trees require contexts that are either empty (as in T-Unit) or that contain exactly what is being typed (as in T-Var).

Rules with multiple premises vary depending on how many of their subterms will eventually be evaluated. If only one of several will, then all those subexpressions should share the same contexts, as in T-With. When multiple subexpressions will be evaluated, as in T-Tensor, the contexts must be divided among them. We write $\Pi_1 \Cup \Pi_2$ and $\Delta_1 \Cup \Delta_2$ to denote contexts that can be split into Π_1 and Π_2 and into Δ_1 and Δ_2 respectively; this relation is formally defined in Figure 8.

The typing rules for our new constructs are straightforward. The types for $\mathbf{go}^{\rho}\ e$ and $\mathbf{yield}\ e$ have already been discussed; channel endpoints $\lfloor a \rfloor$ and $\lceil a \rceil$ have the types ascribed to them by the channel context Π by $a{\cdot}\rho$ and $\tilde{a{\cdot}\rho}$ respectively. The closed channel $\rfloor a \lfloor$ accounts for both endpoints but must be given the type $\bot$.

We write $\Pi \vdash P : \tau$ for a well-typed process P with channels typed by Π; our process typing rules are given in Figure 9. No Δ is needed, as processes never depend on expression variables; rule TP-Exp type checks atomic processes in the empty variable context. Rule TP-New extends the channel environment at binders. As the final type of all processes but our original will always be $\bot$, rules TP-ParLeft and TP-ParRight require that one of their components always have type $\bot$.

Note that TP-ParLeft and TP-ParRight split their channel context with $\widehat{\Cup}$ rather than simply $\Cup$. As seen in Figure 8, this allows exactly one $a{:}\rho$ binding to be decomposed into an $a{\cdot}\rho$ binding and an $\tilde{a{\cdot}\rho}$ binding. This means that, in any well-typed process of the form $P_1 \mid P_2$, there can be at most one channel for which one endpoint is in P_1 and the other is in P_2. This restriction substantially cuts down the set of well-typed processes and, as will be seen shortly, proves crucial for type soundness.

4.2 Soundness

Taking the usual approach and defining soundness in terms of preservation—well-typed terms that step always step to well-typed

$$[\text{EQP-REFL}]\ P \equiv P \qquad [\text{EQP-SYM}]\ \frac{P_2 \equiv P_1}{P_1 \equiv P_2} \qquad [\text{EQP-TRANS}]\ \frac{P_1 \equiv P_2 \qquad P_2 \equiv P_3}{P_1 \equiv P_3} \qquad [\text{EQP-COMM}]\ P_1 \mid P_2 \equiv P_2 \mid P_1$$

$$[\text{EQP-PAR}]\ \frac{P_1 \equiv P_1' \qquad P_2 \equiv P_2'}{P_1 \mid P_2 \equiv P_1' \mid P_2'} \qquad [\text{EQP-NEW}]\ \frac{P \equiv P'}{\nu a{:}\rho.\ P \equiv \nu a{:}\rho.\ P'} \qquad [\text{EQP-ASSOC}]\ (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$[\text{EQP-SWAP}]\ \nu a_1{:}\rho_1.\ \nu a_2{:}\rho_2.\ P \equiv \nu a_2{:}\rho_2.\ \nu a_1{:}\rho_1.\ P \qquad [\text{EQP-EXTRUDE}]\ \frac{a \text{ not free in } P_2}{(\nu a{:}\rho.\ P_1) \mid P_2 \equiv \nu a{:}\rho.\ (P_1 \mid P_2)}$$

Figure 10. Process equivalence rules

terms—and progress—well-typed non-values can always take a step—we observe that, while preservation makes sense on both expressions and processes, progress is only a property of well-typed processes, as there are certainly well-typed expressions that require the process evaluation rules to take a step.

Preservation on expressions is straightforward, requiring the usual substitution lemma:

Lemma 1 (Substitution)**.** *If* $\Pi; \Delta_1, x{:}\tau', \Delta_2 \vdash e : \tau$ *and* $\Pi'; \Delta' \vdash e' : \tau'$*, then* $\Pi, \Pi'; \Delta_1, \Delta', \Delta_2 \vdash \{x \mapsto e'\}e : \tau$.

Lemma 2 (Expression preservation)**.** *If* $\Pi; \Delta \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\Pi; \Delta \vdash e' : \tau$.

We have proved these results in the Coq proof assistant; the proofs are fairly standard, although the linear contexts introduce complexities that can usually be avoided in other systems, *e.g.*, the need to reason about context permutation.

Preservation and progress for processes are more complex. We first define a process equivalence relation $\equiv$ as shown in Figure 10. This equivalence separates unimportant structural differences in process syntax from the evaluation rules of Figure 3, which determine how processes truly evolve. All of these equivalence rules are standard; they state that the precise position of binders, as well as the order and grouping of parallel composition, are irrelevant.

We next introduce a notion of (not necessarily unique) *canonical forms* for processes: a canonically formed process is of the form $\nu a_1{:}\rho_1.\ \ldots \nu a_m{:}\rho_m.\ e_1 \mid (e_2 \mid (\ldots \mid e_n))$ for some $m \geq 0$ and $n \geq 1$. It is easy to see that any process can be put in canonical form by using the process equivalence rules.

Property 3 (Canonization)**.** *For any process* P*, there exists some* P' *in canonical form such that* $P \equiv P'$.

We define the *communication graph* of a process P to be the undirected[6] multigraph in which the vertices are the atomic processes (that is, expressions) that make up P and an edge exists for each active channel a within the process, connecting the expressions containing $\lfloor a \rfloor$ and $\lceil a \rceil$. (No edge exists for $\lfloor a \rfloor$.) Since graphs are built out of atomic processes, it is easy to see that this graph structure is invariant under process equivalence.

Property 4 (Graph invariance)**.** *For any processes* P *and* P' *where* $P \equiv P'$*, the communication graph of* P' *is isomorphic to the communication graph of* P.

We immediately notice a correspondence between well-typedness of a process and the acyclicity of its communication graph:

Lemma 5 (Acyclicity and typing)**.** *If* $\Pi \vdash P : \tau$*, then the communication graph of* P *is acyclic.*

[6] One might imagine that the directed nature of communication in Lolliproc would suggest directed graphs, but undirected graphs both entail stronger acyclicity properties and simplify the proof of process preservation.

Proof. Recall the definition of $\widehat{\mathbb{U}}$, which allows only a channel to be split over the two halves of a parallel composition. It is not possible to partition the atomic processes in a cycle without going through at least two edges, thus making it impossible to type check a process with a cyclic communication graph. □

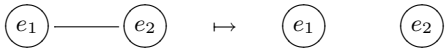
Finally, we observe that acyclicity of communication graphs is preserved under process evaluation:

Lemma 6 (Acyclicity and evaluation)**.** *If the communication graph of* P *is acyclic and* $P \longrightarrow P'$*, then the graph of* P' *is also acyclic.*

Proof. With respect to evaluation graphs, we observe that all evaluation steps amount to doing some combination of the following:

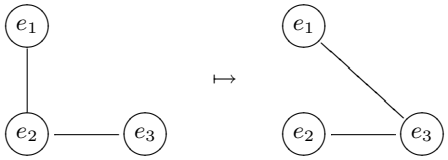
1. the creation of a new vertex and a new edge connecting it to one existing vertex, *e.g.*

2. the deletion of a single edge, *e.g.*



3. the deletion of a single unconnected vertex, *e.g.*

4. and transferring the endpoint of an edge from one vertex to another by sending it across some other edge, *e.g.*



EP-GO involves one use of (1) along with uses of (4) corresponding to the number of channel endpoints in the argument to $\mathbf{go}^{\rho}$. EP-APPSINK can similarly be seen as a repetition of (4), while EP-CLOSE and EP-DONE exactly correspond, respectively, to (2) and (3). All other evaluation rules do not impact the communication graph.

Only (4) can conceivably create a cycle. If a cycle is created, the final step in its creation must be the connection of some atomic processes e_1 and e_2. But this can only be facilitated by some e_3 that is already connected to both e_1 and e_2, in which case a cycle would already exist! Acyclic graphs can thus never become cyclic through application of these graph operations. □

We can now tackle preservation and progress; our statements of both lemmas reflects the idea that both process typing and process evaluation are performed modulo the process equivalence relation.

Lemma 7 (Process preservation)**.** *If $\Pi \vdash P_1 : \tau$ and there exists some P_1' and P_2' such that $P_1 \equiv P_1'$ and $P_1' \longrightarrow P_2'$, then there exists some P_2 such that $P_2 \equiv P_2'$ and $\Pi \vdash P_2 : \tau$.*

Proof. Mostly straightforward, given the obvious extensions of Lemma 2 to evaluation contexts and processes. The difficulty comes from the requirement of the channel context splitting relation $\widehat{\mathbb{U}}$ that at most one $a{:}\rho$ binder be split at each step. We must show that, given the canonical form of P_2', we can always rearrange the parallel compositions such that this is the case.

Observe, however, that we can always do this if the communication graph of P_2' (and thus its canonical form) is acyclic: we have our parallel compositions cut at most one edge at a time, and we will eventually reduce down to atomic processes. From Lemma 5 we already know that the communication graph of P_1 and hence also P_1' is acyclic, and thus from Lemma 6 we can conclude that the graph of P_2' is acyclic as well. From this we can appropriately rearrange its canonical form to create a well-typed P_2. □

For progress we must first define what it means for a process to be done evaluating. We use one of the simplest such definitions: a process has *finished* when it contains an atomic process that is a value and that is not $\lfloor a \rfloor$, $\lceil a \rceil$, or $\rfloor a \lfloor$. Our proofs make use of the standard canonical forms properties: all expressions of a given type eventually reduce to certain forms. Some types have more canonical forms than usual, as sources and sinks are both values.

Lemma 8 (Progress)**.** *If $\Pi \vdash P : \tau$, then either P has finished or there exists some P_1 and P_2 such that $P \equiv P_1$ and $P_1 \longrightarrow P_2$.*

Proof. We proceed by examining each of the atomic processes within P. If, in doing so, we find an appropriate value or the opportunity to take a step, then we are done, but we may encounter an expression e stuck at the elimination of a sink or a **yield** on a source. In that case, we consider the atomic process e' that contains the other endpoint of the channel in question. If e' itself can take a step, we are done. If e' is ready to communicate with e we stop searching, as we have found a matched source and sink. Otherwise, e' itself is stuck at the elimination of a sink or a **yield** on a source for some different channel, in which case we recursively continue our search using the same procedure.

Because P is well typed, it has an acyclic communication graph, so this search will eventually terminate in the identification of some matching source and sink that are ready to communicate. We then consider the canonical form of P and repeatedly push the appropriate channel binding inwards until the process matches the form of one of our communication rules. □

From progress and preservation, we can state the standard soundness theorem:[7]

Theorem 9 (Soundness)**.** *If $\cdot \vdash P{:}\tau$, then there exists no P_1 such that $P \equiv P_1$, $P_1 \longrightarrow^* P_2$, and P_2 has not completed but is not equivalent to any process that can step further.*

This soundness property guarantees freedom from deadlocks in Lolliproc, but our type system says nothing about whether an expression will evaluate to a single value or a composition of processes—both are considered acceptable final outcomes, and there is nothing preventing the programmer from, for instance, not matching each call to `future` with a corresponding call to `wait`. These concerns can be addressed in a language that also includes unrestricted types, however, which we will discuss in Section 5.

[7] We are still working to extend our Coq proofs to preservation and progress on processes; complications arise due to the relatively informal nature, by Coq's standards, of our the graph-based reasoning.

4.3 Strong normalization and confluence

Other properties common to simple, typed λ-calculi are strong normalization—the fact that all sequences of evaluations terminate—and confluence—the fact that all possible evaluations for a given term converge to the same final result. Although Lolliproc has a non-deterministic operational semantics, it still enjoys these properties.

Theorem 10 (Strong normalization)**.** *If $\Gamma \vdash P : \tau$, any reduction sequence $P \equiv P_1$, $P_1 \longrightarrow P_1'$, $P_1' \equiv P_2$, $P_2 \longrightarrow P_2'$, ... will eventually terminate in some P_n' such that there exists no P_{n+1} and P_{n+1}' for which $P_n' \equiv P_{n+1}$ and $P_{n+1} \longrightarrow P_{n+1}'$.*

Proof. Since everything in our language is linear, subterms are never duplicated; thus we can verify strong normalization by assigning non-negative weights $w(P)$ to processes P and $w(D)$ to derivations D of $\Gamma; \Delta \vdash e : \tau$—which we abbreviate as $w(e)$—and showing that these weights always decrease with evaluation.

We define $w(\nu a{:}\rho.\ P) = 1 + w(P)$ and $w(P_1 \mid P_2) = w(P_1) + w(P_2)$. For channel endpoints, we first define the length of a protocol type $\ell(\rho)$ as $\ell(\lightning) = 1$, $\ell(\tau \multimap \rho) = 1 + \ell(\rho)$, and $\ell(\rho_1 \& \rho_2) = 1 + \max(\ell(\rho_1), \ell(\rho_2))$. Whenever $\lfloor a \rfloor$ has type ρ, we define $w(\lfloor a \rfloor) = \ell(\rho)$; similarly, when $\lceil a \rceil$ has type $\widetilde{\rho}$, we define $w(\lceil a \rceil) = 2 \cdot \ell(\rho)$ (as larger terms appear on the source side after communication). Since process communication always decreases the length of the protocol type, it will consequently decrease the weight of the composite process. We define $w(\mathbf{go}^{\rho}\ e) = 2 + 3 \cdot \ell(\rho) + w(e)$, ensuring that its evaluation also decreases in weight even as it spawns a new process.

The weights of most other expression forms are fairly straightforward; for instance, $w(x) = w(()) = 0$, $w(\lambda x{:}\tau e) = 1 + w(e)$, $w((e_1, e_2)) = 1 + w(e_1) + w(e_2)$, and $w(\langle e_1, e_2 \rangle) = 1 + \max w(e_1), w(e_2)$. The cases for **yield** and application are tricky, though, since the rules E-YIELDOTHER and E-APPSOURCE appear to increase the size of terms. For **yield**, we define $w(\mathbf{yield}\ e) = 1 + w(e)$ whenever e is either $(\mathbf{go}^{\rho}\ e')$ or any source; otherwise, given that e is assigned the type $\lceil \tau \rceil$, we define

$$\begin{aligned} w(\mathbf{yield}\ e) &= 1 + w(\mathbf{let}\ (y, z) = \mathbf{yield}\ (\mathbf{go}^{\tau \multimap \lightning}\ e)\ \mathbf{in}\ z; y) \\ &= 5 + w(\mathbf{go}^{\tau \multimap \lightning} e) \\ &= 13 + w(e) \end{aligned}$$

For applications, we must conservatively estimate how many times E-APPSOURCE might be applied. For this we first define the height of a type $h(\tau)$ such that $h(\tau \multimap \lightning) = 1 + h(\tau)$ and $h(\tau) = 0$ otherwise. Assuming the derivation for $e_1\ e_2$ gives e_1 the type $\tau_1 \multimap \tau_2$ and e_2 the type τ_1, then we can define $w(e_1\ e_2) = 1 + 14 \cdot h(\tau_1) + w(e_1) + w(e_2)$, since the height of τ_1 determines the maximum number of **yield**s that could ever be introduced.

With these definitions in place, it is clear by inspection of our evaluation rules that the weight of a process decreases with each evaluation step. Since weights are never negative, this assures us that evaluation always terminates. □

With strong normalization, we can obtain confluence directly from local confluence (also known as the diamond property).

Theorem 11 (Local confluence)**.** *If $\Gamma \vdash P : \tau$, and we have that $P \equiv P_1$, $P \equiv P_2$, $P_1 \longrightarrow P_1'$, and $P_2 \longrightarrow P_2'$, then there exist some P_3, P_3', P_4, and P_4' such that $P_1' \equiv P_3$, $P_3 \longrightarrow P_3'$, $P_2' \equiv P_4$, $P_4 \longrightarrow P_4'$, and $P_3' \equiv P_4'$.*

Proof. Our expression evaluation rules are deterministic, and there is only one way to decompose an expression e into some $E\left[e'\right]$ such that some expression or process evaluation rule applies—and only one such rule will ever apply. Our only source of non-determinism, then, is the parallel composition of processes. We

must thus show that the evaluation $P_1 \longrightarrow P_1'$ does not rule out subsequently applying the same steps that produced $P_2 \longrightarrow P_2'$, and vice-versa.

We observe that, in a well-typed process, potential evaluation steps can never interfere with each other. We have only two endpoints for each process, so multiple acts of communication can never conflict, and since communication always involves values, it cannot conflict with some internal evaluation step on a non-value expression. And of course such internal steps cannot conflict with each other. It is thus easy to see that local confluence holds. □

Strong normalization and confluence show that the concurrency available in Lolliproc is particularly well behaved. Strong normalization implies that there are no livelocks, while confluence implies a lack of race conditions, which could otherwise introduce irreconcilable nondeterminism.

5. Future directions and related work

Finally, we examine a few possible future directions of this work and look briefly at related systems.

5.1 Extending Lolliproc

Lolliproc is very far from being a full-fledged programming language. Many of the extensions needed to bridge this gap—compilation and runtime system, support for processes spread over the network, useful libraries, *etc.*—are beyond the scope of this paper, but several obvious extensions do warrant more discussion here.

Unrestricted types and polymorphism Although we have defined Lolliproc such that *all* variables must be used exactly once, this is clearly an unrealistic simplification; unrestricted types must be accounted for somehow. In earlier work [31] we introduced an intuitionistic language System F$^\circ$, an extension of the fully polymorphic System F in which the distinction between the linear and the unrestricted is handled at the *kind* level: a kind $\star$ categorizes unrestricted types, while a kind $\circ$ categorizes linear types. System F$^\circ$ features a subkinding relation in which $\star \leqslant \circ$, implying that unrestricted types may safely be treated as though they were linear.

We can extend this approach to encompass Lolliproc by introducing a protocol kind $\bullet$ such that $\bullet \leqslant \circ$. We could then replace our syntactic separation of ρ types with the appropriate kinding rules. For function types—which System F$^\circ$ writes as $\xrightarrow{\kappa}$ rather than the $\multimap$ we use for Lolliproc—this gives us

$$[\text{K-ARR}]\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2 \qquad \kappa = \bullet \implies \kappa_2 = \bullet}{\Gamma \vdash \tau_1 \xrightarrow{\kappa} \tau_2 : \kappa}$$

Here Γ is an unrestricted context, binding both type variables and, although not relevant to this judgment, unrestricted term variables.

Since such a system allows quantification over type variables α of kind $\bullet$, we would also require dualized type variables $\widetilde{\alpha}$, instantiated to $\widetilde{\rho}$ whenever α is instantiated with ρ. If we also allow $\forall\alpha{:}\kappa.\ \rho$ to be a protocol type—thus permitting types to be sent between processes—we gain even greater flexibility, allowing partially specified protocols dependent on protocol type variables.

Adopting the techniques of System F$^\circ$ also allows us to address the concerns mentioned at the end of Section 4.2: we would know that, if e is a well-typed expression of type τ that does not contain any channel endpoints, e will eventually step to some isolated value v, regardless of how many processes may be spawned along the way. Here we appeal to an alternate operational semantics for System F$^\circ$ that tags values and types as they are substituted into expressions: this semantics guarantees that unrestricted values do not contain tagged linear objects, and, since channel endpoints do not appear in source programs, they would always appear tagged.

Recursion and non-determinism We have proved in Section 4.3 that Lolliproc is both strongly normalizing and confluent. However, one does not generally want to program in languages that rule out non-terminating programs, and in a concurrent setting it is common to want programs that might evaluate differently depending on which processes are available to communicate at which times, thus breaking confluence.

One natural companion to Lolliproc's existing constructs is *recursive types* $\mu\alpha[{:}\kappa].\,\tau$, where any αs appearing within τ expand to $\mu\alpha[{:}\kappa].\tau$. Such types allow for full general recursion, can be used to encode many standard datatypes (*e.g.*, lists over a given type), and, in our setting, enable looping protocols, for which there are many obvious applications. For instance, we could write a session-serving server with the type $\mu\alpha[{:}\bullet].\,\uparrow(\rho \otimes \alpha))\uparrow$, which could be used to send out any number of sessions for the protocol ρ.

For controlled non-confluence, we can imagine a family of primitive functions like the one below

$$\mathsf{receive}_2^{\tau_1,\tau_2,\tau} : \uparrow\tau_1\uparrow \multimap \uparrow\tau_2\uparrow \multimap ((\tau_1 \multimap \uparrow\tau_2\uparrow \multimap \tau) \,\&\, (\uparrow\tau_1\uparrow \multimap \tau_2 \multimap \tau)) \multimap \tau$$

A call to a receive function waits until a **yield** on one of its source arguments can succeed, then selects and applies the appropriate function from its additive product argument to handle that result and the other remaining sources. (We would, of course, want syntactic sugar for these functions.)

This closely mimics the non-deterministic operations found in many concurrent languages—*e.g.*, the join calculus [20, 21] and Erlang [3]—while still preserving our linearly typed channels. We would also likely want other constructs to handle cases for which receive is awkward: for instance, we might want non-deterministic analogs of map and fold for several sources of the same type.

Proof theory The expression typing rules in (Figure 7), when viewed as a logic, are clearly *sound* with respect to standard classical linear logic. To see why, note that we may consider only case where Π is empty, as channels do not occur in source programs. Our only nonstandard rules are then T-GO and T-YIELD, but these are both admissible in standard linear logic. We leave establishing the *completeness*—with respect to the non-exponential fragment of standard linear logic—to future work. It would also be interesting to study the relationship between our evaluation rules and proof normalization—there seems to be a strong connection between our definition of channel endpoints and "focused" proofs [46].

5.2 Related work

There is a vast literature on linear logic, its proof theory, and related type systems, ranging from applications to categorical semantics—we cannot possibly cover it all here. Thus we highlight the work most closely related to ours and suggest some connections that might be fruitful for subsequent research.

Intuitionistic linear types The intuitionistic fragment of linear logic has seen much use in programming languages [6, 9, 11, 29, 30]—particularly its connections to memory management [2, 13, 39, 42]. We recently looked at enforcing user-defined protocols in a linear variant of System F [31]. De Paiva and Ritter study an intuitionistic linear language that, like Lolliproc, is not directly involutive (*i.e.*, τ is not identified with $\tau^{\perp\perp}$); its operational semantics is reminiscent of the classical calculi described below.

Classical natural deduction and control Natural deduction presentations of classical logics [10, 14, 35–37] typically use multiple conclusions judgments of the form:

$$x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e : \tau,\ y_{n+1}{:}\tau_{n+1}, \ldots, y_m{:}\tau_m$$

By duality, such a judgment is logically equivalent to

$$x_1{:}\tau_1, \ldots, x_n{:}\tau_n,\ y_{n+1}{:}\tau_{n+1}^{\perp}, \ldots, y_m{:}\tau_m^{\perp} \vdash e : \tau$$

This approach recovers the usual shape of the typing judgment and so can be reconciled more easily with type systems for functional programming. Moreover, if we recall that $\tau \multimap \bot$ is the type of a continuation accepting τ values, it is possible to read the ys above as binding continuations. Operational semantics in this setting implement commuting conversions, which give rise to nondeterminism. The correspondence with concurrency is obscured, however, because these semantics rely on decomposing a single term (often using evaluation contexts).

The connection between classical logic and control operators has been known for some time [17, 23, 34]. As mentioned in Section 2, **control** has the type of double-negation elimination; the more familiar **callcc** can similarly be given the type of Peirce's Law. While these operations cannot be directly imported to the linear setting, they are a major part of the inspiration for our approach.

Linear continuations in otherwise unrestricted languages have also been studied, as they indicate certain predictable patterns of control flow [8, 19]. Berdine's dissertation [7], for example, shows how such continuations can be used to implement coroutines; Lolliproc goes further by allowing true concurrent evaluation.

Our process typing rules can be seen as an alternative to the multiple conclusion judgment style described above. While these systems give all auxiliary conclusions a continuation type $\tau \multimap \bot$, our helper processes simply have type $\bot$. A practical consequence of our design is that, since processes appear only at runtime, a type checker for a language based on Lolliproc would not need to implement these rules at all.

Linear sequent calculi In order to take advantage of the symmetries discussed in Section 2, languages and proof terms based on linear sequent calculi [1, 22] feature a multiplicative disjunction $\parr$ and define $\tau_1 \multimap \tau_2$ as $\tau_1^{\bot} \parr \tau_2$. It has proved difficult, however, to find intuitions for $\parr$ in a standard functional programming setting that fit as naturally as those for $\otimes$, $\&$, and $\oplus$ [44].

We can encode $\parr$ in Lolliproc by noting following the logical equivalence:

$$\tau_1 \parr \tau_2 \iff ((\tau_1 \multimap \bot) \multimap \tau_2) \,\&\, ((\tau_2 \multimap \bot) \multimap \tau_1)$$

We will not be able to construct an object of this type unless we can eliminate some $\tau \multimap \bot$ without producing a witness of type $\bot$, which requires the existence of another process and a channel over which we can send the closed channel token. Thus $\parr$ serves as a way of internalizing—and at least partially suspending—two processes within one, although it cannot exist in isolation. The choice of projections offered by $\&$ internalizes the commutativity of the '|' constructor of process terms.

Zeilberger presented an interesting sequent calculus [46] that, while not actually linear, makes use of the connectives of linear logic for their polarity and gives a term assignment in which eager positive connectives and lazy negative connectives coexist harmoniously. The dual calculus [43] and Filinski's language [18] are also tightly tied to sequent calculus while being closer to standard term languages than, *e.g.*, proof nets. All of these languages define programs as interactions between terms and co-terms, departing rather significantly from the norm in functional programming.

Process calculi Many type systems exist for the π-calculus [32], some able to guarantee sophisticated properties; Kobayashi [27] gives a good overview of this area. Many of these type systems use linearity in one form or another [4, 28], and, in particular, session types [12, 25, 38, 40] originated in this setting. The Sing♯ language, which ensures safety for its light-weight processes through its type system, takes many ideas from the world of process calculi [15].

Programming in a process calculus, however, is also rather different from programming in a traditional functional language, and it is not always clear how to best take ideas from that setting while reusing as much standard machinery as possible. Additionally, π-calculus type systems are not as tightly coupled with logics as λ-calculus type systems are, though there has been some work on using π-calculus terms to describe proof reductions [5].

5.3 Conclusion

We have presented Lolliproc, a concurrent language whose design separates source programs from the processes they spawn at runtime, while retaining a close correspondence to classical linear logic. Though simple, Lolliproc can express useful protocols whose well-behaved interactions are enforced by session types. It is our hope that Lolliproc will inspire language designers, if not to build their next language on its ideas, then at least to consider what linear types might have to offer in terms of concurrency. Whether or not this comes to pass, however, we feel that our approach offers an appealing point in the design space of concurrent calculi.

Acknowledgments

The authors thank the anonymous reviewers, the Penn PL Club, and the MSR Cambridge types wrestling group for their feedback about this work. Phil Wadler and Guillaume Munch-Maccagnoni also provided excellent suggestions about how to improve this paper. This work was supported in part by NSF Grant CCF-541040 and some of this research was conducted while the second author was a visiting researcher at Microsoft Research, Cambridge.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, 2007.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[4] Emmanuel Beffara. A concurrent model for linear logic. *Electronic Notes in Theoretical Computer Science*, 155:147–168, 2006.

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

[6] Nick Benton, G. M. Bierman, J. Martin E. Hyland, and Valeria de Paiva. A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 75–90. Springer-Verlag LNCS 664, 1993.

[7] Josh Berdine. *Linear and Affine Typing of Continuation-Passing Style*. PhD thesis, Queen Mary, University of London, 2004.

[8] Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linearly used continuations. In *Proceedings of the Continuations Workshop*, 2001.

[9] G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Fourth International Workshop on Higher Order Operational Techniques in Semantics, Montral, volume 41 of Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[10] Gavin Bierman. A classical linear lambda calculus. *Theoretical Computer Science*, 227(1–2):43–78, 1999.

[11] Gavin M. Bierman. Program equivalence in a linear functional language. *Journal of Functional Programming*, 10(2), 2000.

[12] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, Paris, France, August 2010. Springer LNCS.

[13] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 213–224, New York, NY, USA, 2008. ACM.

[14] Valeria de Paiva and Eike Ritter. A parigot-style linear lambda-calculus for full intuitionistic linear logic. *Theory and Applications of Categories*, 17(3), 2006.

[15] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[16] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 13–24, Berlin, Germany, June 2002.

[17] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[18] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, August 1989.

[19] Andrzej Filinski. Linear continuations. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, pages 27–38, 1992.

[20] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[21] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, nov 1998.

[22] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[23] Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58. ACM Press, 1990.

[24] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in Cyclone. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 73–84, New York, NY, USA, 2004. ACM.

[25] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag, 1998.

[26] W. A. Howard. The formulae-as-types notion of contstruction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[27] Naoki Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 10th Anniversary Cooloquium*, March 2002.

[28] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[29] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988. Corrections in vol. 62, pp. 327–328.

[30] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11'th International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Lousiana, – 1995.

[31] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 77–88, New York, NY, USA, 2010. ACM.

[32] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[33] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

[34] C.-H. L. Ong and C. A. Stewart. A curry-howard foundation for functional computation with control. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 215–227, Paris, France, 1997.

[35] Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

[36] Michel Parigot. Classical proofs as programs. In *Proceedings of the 3rd Kurt Gödel Colloquium*, volume 713 of *Lecture Notes in Computer Science*, pages 263–276. Springer-Verlag, 1993.

[37] Eike Ritter, David J. Pym, and Lincoln A. Wallen. Proof-terms for classical and intuitionistic resolution. *Journal of Logic and Computation*, 10(2):173–207, 2000.

[38] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, pages 398–413. Springer-Verlag, 1994. Lecture Notes in Computer Science number 817.

[39] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231–248, September 1999.

[40] Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

[41] Edsko Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 201–218, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[43] Philip Wadler. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 189–201, New York, NY, USA, 2003. ACM.

[44] Philip Wadler. Down with the bureaucracy of syntax! Pattern matching for classical linear logic. unpublished manuscript, 2004.

[45] Nobuko Yoshida, Kohei Honda, and Martin Berger. Linearity and bisimulation. *J. Log. Algebr. Program.*, 72(2):207–238, 2007.

[46] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3):66–96, 2006.

[47] Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, Long Beach, CA, January 2005. Springer-Verlag LNCS vol. 3350.