Session-Typed Concurrent Logical Specifications

Henry DeYoung
hdeyoung@cs.cmu.edu

February 5, 2015
Minor revisions on March 15, 2015

Abstract

Concurrency arises naturally in a proof-construction-as-computation interpretation of intuitionistic linear logic: following concurrent equality, if all permutations of independent proof steps in a logical specification are treated as indistinguishable, then those proof steps appear to happen concurrently. Concurrency also arises naturally in proof-reduction-as-computation: there is a Curry–Howard isomorphism due to Caires and Pfenning between sequent proofs in intuitionistic linear logic and session-typed processes in the π-calculus, between principal cut reductions and process reductions.

In this proposal, we put forward the thesis that session types form a bridge between these two apparently disparate notions of concurrency. Specifically, we propose to show that, given an assignment of process and message roles to atomic propositions, a class of concurrent linear logical specifications can be translated to session-typed processes. In addition to the practical benefits of generating well-typed implementations from logical specifications, the proposed work can be seen as giving a proof-theoretic reconstruction of work on multiparty session types; as assigning behavioral types to a class of logical specifications, thereby ensuring deadlock freedom for those specifications; and as furthering an understanding of the relationship between proof construction and proof reduction.

This document aims to establish the thesis's plausibility by defending it in the restricted setting of intuitionistic ordered logic. The primary area of proposed research will then be to relax that restriction, extending the ideas in this document to linear logic.

Keywords: substructural logics, proof reduction, proof construction, concurrency, session types

Contents

1 Introduction

With the increasingly complex, distributed nature of today's software systems, concurrency is ubiquitous. Concurrency facilitates distributed computation by structuring systems as nondeterministic compositions of simpler subsystems. But, concomitant with nondeterminism, concurrent systems are notoriously tricky to get right: subtle races and deadlocks can occur even in the most rigorously tested of systems.

At the same time, decades of research into connections between proof theory and programming languages have firmly established the principle of *computation as deduction* as the gold standard framework for clear, expressive, and provably correct programs. Examples abound: lax logic for effectful computation (Benton et al. 1998), temporal logic for functional reactive programming (Jeffrey 2012), and linear logic for graph-based algorithms (Cruz et al. 2014), to name just a few.

Can a computation-as-deduction approach make it similarly easier to clearly and concisely specify, as well as correctly implement, concurrent programs?

✦

Computation-as-deduction comes in two flavors: *proof-construction-as-computation* and *proof-reduction-as-computation*. Proof-construction-as-computation views the search for a proof, according to a fixed strategy, as the basis of computation; it is the foundation for logic programming (Miller et al. 1991; Andreoli 1992). Proof-reduction-as-computation, on the other hand, revolves around a correspondence, known as the Curry–Howard Isomorphism (Howard 1980), between propositions and types, proofs and programs, and proof simplification, or reduction, and program evaluation; it is the foundation for typed functional programming (Martin-Löf 1980).

Both the proof-construction and proof-reduction approaches have been applied to concurrent programming, stemming from Girard's (1987) suggestion of connections between linear logic and concurrency. In the proof-construction vein, the Concurrent Logical Framework (CLF; Watkins et al. 2002) treats the permutability of inference rules as a source of concurrency. CLF has been used to specify a variety of concurrent systems, ranging from the π-calculus to security protocols and even emergent story narratives (Cervesato and Scedrov 2009; Martens et al. 2013). Although these same concurrent systems can be simulated according to their CLF specifications by the Lollimon (López et al. 2005) and Celf (Schack-Nielsen 2011) logic programming engines, the programs ultimately remain specifications, not actual *decentralized* implementations.

Taking the other, proof-reduction tack, Abramsky (1993), Bellin and Scott (1994), and later Caires and Pfenning (2010) with Toninho (2012, 2013), among others, have given correspondences between sequent calculus proofs or proof nets in linear logic and concurrent processes, between cut elimination and concurrent process execution. Moreover, in Caires et al.'s work, the correspondence is a true Curry–Howard isomorphism in that intuitionistic linear propositions are also types—*session types* (Honda 1993) that describe the interaction protocol to which a process adheres. Unlike proof construction, the proof-reduction approach more naturally yields actual decentralized implementations (Toninho et al. 2013; Griffith and Pfenning 2014).

In spite of their common basis in linear logic, the proof-construction and proof-

reduction approaches to concurrent computation appear at first glance to be strikingly disparate. They have different dynamics; they offer different guarantees (session fidelity, behavioral type preservation, and deadlock freedom for the proof-reduction approach, but only non-behavioral type preservation for the proof-construction approach); and, perhaps most importantly, they serve very different roles in programming practice. Proof construction is better suited to system specification and reasoning, whereas proof reduction is better suited to implementation.

To reduce the possibility of error when building an implementation from a specification, we'd like to minimize the gap between the two. Despite the apparent disparity between proof construction and proof reduction, is there a class of concurrent specifications from which distributed concurrent implementations can be automatically extracted? Stated differently, is there perhaps some fragment of linear logic in which the computational nature of proof construction and proof reduction correspond?

✦

The thesis is that, yes, thanks to session types, we can have our cake and eat it too:

> Thesis statement. *Session types form a bridge between distinct notions of concurrency in computational interpretations of intuitionistic linear logic based on proof construction, on one hand, and proof reduction, on the other hand.*

The remainder of this proposal document aims to establish this thesis as a plausible one. To do so, we turn our attention from intuitionistic linear logic to propositional intuitionistic ordered logic (Lambek 1958; Polakow and Pfenning 1999)—a restriction of linear logic in which the context of hypotheses forms a list rather than a multiset or bag—and defend the thesis in this restricted setting. The primary area of proposed thesis research will then be to relax this restriction, extending the ideas in this document to intuitionistic linear logic.

Specifically, in defending the thesis for ordered logic, this document breaks down the problem into several pieces, as depicted in Fig. 1. First, Section 2 reviews a string rewriting interpretation of proof construction in a non-modal fragment of intuitionistic ordered logic (Simmons 2012). These ordered logical specifications are equipped with a natural notion of concurrency based on treating as equivalent the different interleavings of independent rewriting steps—essentially CLF's *concurrent equality* (Watkins et al. 2002; Cervesato et al. 2012) adapted to the ordered setting.

Despite being concurrent, ordered logical specifications lack an immediate notion of *process* or *process identity*. Toward this end, Section 3 introduces *choreographies*, a further restriction of ordered logical specifications in which atomic propositions are assigned roles as either process-like atoms or message-like atoms. (By convention, message-like atoms, such as $\underleftarrow{\mathsf{inc}}$ in Fig. 1, are indicated with an arrow decoration.) A specification may admit several choreographies, but, as described in Section 3.2, a well-formed choreography must be lock-step equivalent with the specification once the role annotations are erased.

However, even with process- and message-like atoms and the notion of process that they confer, choreographies remain specifications rather than full-fledged process implementations. Choreographies are nevertheless the crucial stepping-stone.

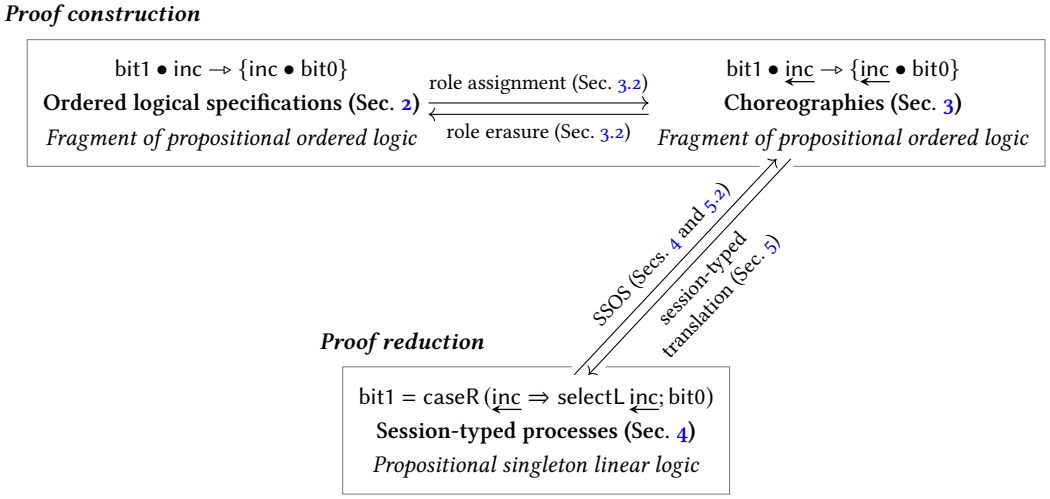


Figure 1: Proof construction to proof reduction

Section 4 presents a variant of linear logic in which sequents are restricted to use at most one linear hypothesis[1]; we dub it *singleton linear logic*. When viewed through a Curry–Howard lens, singleton linear logic becomes a session-typed process calculus in which well-typed process networks are "linear" chains of processes: following the example of Caires et al.'s SILL (2013), propositions are session types, proofs are process chains, and proof reductions are process chain reductions.

Process chains prove to be just the right match for the ordered contexts used by ordered logical specifications and their choreographies. By giving process chains a substructural operational semantics (SSOS; Pfenning 2004) using ordered logical specifications, it's fairly straightforward to relate proof reduction to proof construction. What's surprising, however, is that the converse is also possible—as we show in Section 5, the choreographies of Section 3 (which are ordered logical specifications) may be translated to process chains. And this translation is correct, in the sense that it is a weak bisimulation: a choreography's transitions and its corresponding process's reductions (as defined by the SSOS) match quite closely (Section 5.2).

✦

To summarize, the proposed thesis is that, for a class of concurrent linear logical specifications, we can establish a correspondence with session-typed processes and thereby relate the proof-construction approach to concurrent computation to the proof-reduction approach.

The contributions of this proposed thesis can be viewed from several perspectives.

- This work can be seen as a proof-theoretic reconstruction of multiparty session types (Honda et al. 2008). In multiparty session types, binary sessions are generalized to *conversations* among several parties. Programmers specify

[1]Note that such contexts are trivially ordered, meaning that the logic is also a variant of ordered logic.

conversations in their entirety using *global session types*. However, rather than building implementations of the participants directly from the global session type, a collection of *local, binary session types*—one for each pair of participants—is automatically projected from the global type. The programmer then builds a local implementation of each participant from its much simpler local type.

Intuitively, global types for multiparty sessions serve the same purpose as our choreographies: both describe the conversation as a whole. And, because both extract local information from a global description, the projection of local types from global types is related to our translation of well-typed processes from choreographies. Moreover, our framework has the advantage of generating implementations directly from choreographies, whereas the multiparty session type discipline generates only local types that programmers must then implement.

- Unlike those based on proof reduction, computations based on proof construction may fail, essentially because the goal may not be provable. Computations based on proof construction therefore do not generally enjoy the same progress and (strong) type preservation properties as those based on proof reduction. For concurrent logical specifications, this means that computations may deadlock—the notion of a computational state is simply too permissive.

  This work can be seen as assigning behavioral types to a class of specifications: the type of a specification is the type of the process to which it corresponds under the translation described in Section 5. In this way, those specifications do enjoy strong safety properties, such as type preservation and deadlock freedom.

- Finally, this work can be seen as furthering an understanding of the relationship between proof construction and proof reduction. To the best of our knowledge, there has been relatively little work on relating these two proof-theoretic approaches to computation.

  In the functional logic programming paradigm, languages such as Curry (Hanus 2013) and Mercury (Somogyi et al. 1996) combine the functional and top-down logic programming paradigms (which derive from proof reduction and top-down proof construction), but the combination is arguably more of an amalgamation than a connection between the paradigms. Felleisen (1985) and Spivey and Seres (1999) give shallow embeddings of the Prolog logic programming language into the Scheme and Haskell functional languages, respectively.

  None of these works deals with concurrency or bottom-up proof construction, in contrast with our proposal; even more importantly, none of these works treat proof-theoretic aspects.

2 Background: Concurrent ordered logical specifications

Viewed through a computational lens, proof construction in a fragment of ordered logic becomes a bottom-up logic programming language (Pfenning and Simmons 2009). It can be seen as a logically motivated generalization of string rewriting (see, e.g., Book

and Otto 1993), an analogy which we will exploit to provide some intuition for this form of ordered logic programming.

From the perspective of string rewriting, an ordered logical specification's atomic propositions are letters; ordered conjunctions (or ordered contexts) of these atoms are strings; and, under a focused proof construction strategy (Andreoli 1992), the ordered implications that serve as specification clauses are string rewriting rules. An example will help to clarify.

2.1 Example: Binary counter

Using ordered logic, we can specify the behavior of an incrementable binary counter. The counter is represented as a string of bit0 and bit1 atoms terminated at the most significant end by an eps. For instance, the ordered conjunction eps • bit1 • bit0 is a string that represents a counter with value 2. Increment instructions are represented by inc atoms at the counter's least significant end. Thus, eps • bit1 • inc represents a counter with value 1 that has been instructed to increment once.

Operationally, increments are described by three clauses, which together constitute the specification's *signature*, Σ_{inc}. The first of these clauses is

$$\mathsf{bit1} \bullet \mathsf{inc} \multimap \{\mathsf{inc} \bullet \mathsf{bit0}\} \,.$$

From a string rewriting perspective, this implication is a rule for rewriting the (sub)string bit1 • inc as inc • bit0, an interpretation which is justified logically because implications are transformations.[2] By rewriting bit1 • inc as inc • bit0, this clause serves to carry the inc up past any bit1s that may exist at the counter's least significant end.

Whenever the carried inc reaches the eps or right-most bit0, the carry is resolved by one of the other two clauses:

$$\mathsf{eps} \bullet \mathsf{inc} \multimap \{\mathsf{eps} \bullet \mathsf{bit1}\}$$
$$\mathsf{bit0} \bullet \mathsf{inc} \multimap \{\mathsf{bit1}\} \,.$$

By rewriting eps • inc as eps • bit1, this second clause ensures that in the eps case the carry becomes a new most significant bit1; similarly, by rewriting bit0 • inc as bit1, the third clause ensures that in the bit0 case the carry flips the bit0 to bit1.

For example, under the Σ_{inc} signature,[3] the counter eps • bit1 • inc can be maximally rewritten as in the trace

$$\mathsf{eps} \bullet \underline{\mathsf{bit1} \bullet \mathsf{inc}} \longrightarrow_{\Sigma_{\mathsf{inc}}} \underline{\mathsf{eps} \bullet \mathsf{inc}} \bullet \mathsf{bit0} \longrightarrow_{\Sigma_{\mathsf{inc}}} \mathsf{eps} \bullet \mathsf{bit1} \bullet \mathsf{bit0} \not\longrightarrow_{\Sigma_{\mathsf{inc}}} ,$$

where at each step the sites amenable to rewriting have been underlined. This trace computes $1 + 1 = 2$ in binary representation. More generally, the above clauses adequately specify the increment operation on binary numbers.

[2] The braces around inc•bit0 are a monad (or lax modality), a technical device borrowed from CLF (Watkins et al. 2002) to separate top-down proof construction from bottom-up proof construction. The reader who is unfamiliar with CLF can safely gloss over the monad when reading the specifications in this section.

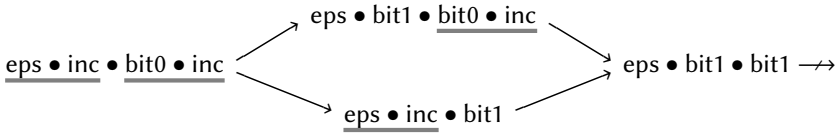
[3] For precision, the rewriting relation, $\longrightarrow$, is indexed by the signature of clauses that may contribute to the rewriting, such as Σ_{inc} in $\longrightarrow_{\Sigma_{\mathsf{inc}}}$ here. We frequently omit the index when it is clear from the context.

2.2 Concurrency

Some strings contain more than one site that is amenable to rewriting. For instance, the binary counter eps • bit1 • inc • inc has two incs in flight, which, after one step, give rise to two disjoint rewrite sites:

$$\underline{\mathsf{eps} \bullet \mathsf{inc}} \bullet \underline{\mathsf{bit0} \bullet \mathsf{inc}}\,.$$

The rewritings at these sites can be interleaved in two ways: either according to the upper path or the lower path in the following diagram.



However, because these two rewritings are independent, they should be considered concurrent. Rather than giving a semantics for string rewriting based on true concurrency, we use interleaved concurrency. We treat different interleavings of independent steps as indistinguishable[4], and then, because we can't observe which rewriting occurred first, the two rewritings appear to happen concurrently. This is the idea of *concurrent equality* (Watkins et al. 2002; Cervesato et al. 2012) from the CLF framework for linear logical specifications, later adapted by Simmons (2012) for ordered logical specifications.

2.3 Example: Binary counter with decrements

As a further example, it's possible to extend the binary counter specification with support for decrements. Like increments, a decrement instruction is represented by a dec atom at the counter's least significant end. To perform the decrement, a dec begins propagating up the counter. As it passes over any bit0s at the least significant end, they are marked as bit0′ atoms to indicate that they are waiting to borrow from their more significant neighbors:

$$\mathsf{bit0} \bullet \mathsf{dec} \multimap \{\mathsf{dec} \bullet \mathsf{bit0}'\}\,.$$

Whenever the dec reaches the eps or right-most bit1, it is replaced with either fail or ok, respectively, to show whether the borrow was possible; in the case of bit1, the borrow is also effected:

$$\mathsf{eps} \bullet \mathsf{dec} \multimap \{\mathsf{eps} \bullet \mathsf{fail}\}$$
$$\mathsf{bit1} \bullet \mathsf{dec} \multimap \{\mathsf{bit0} \bullet \mathsf{ok}\}\,.$$

Then the fail or ok travels back over all of the bit0′ atoms that were waiting to borrow. In the case of fail, the bits are returned to their original bit0 state because no borrow was possible; in the case of ok, a borrow was performed and so the bits are set to bit1:

$$\mathsf{fail} \bullet \mathsf{bit0}' \multimap \{\mathsf{bit0} \bullet \mathsf{fail}\}$$
$$\mathsf{ok} \bullet \mathsf{bit0}' \multimap \{\mathsf{bit1} \bullet \mathsf{ok}\}\,.$$

[4] In the above example, indistinguishability of the two different interleavings is tantamount to commutativity of the diagram.

$$\Sigma_{\mathsf{inc}} = \begin{array}{l} \mathsf{bit1} \bullet \mathsf{inc} \rightarrowtriangle \{\mathsf{inc} \bullet \mathsf{bit0}\}, \\ \mathsf{eps} \bullet \mathsf{inc} \rightarrowtriangle \{\mathsf{eps} \bullet \mathsf{bit1}\}, \\ \mathsf{bit0} \bullet \mathsf{inc} \rightarrowtriangle \{\mathsf{bit1}\} \end{array}$$

(a) Increments

$$\Sigma_{\mathsf{dec}} = \begin{array}{l} \mathsf{bit0} \bullet \mathsf{dec} \rightarrowtriangle \{\mathsf{dec} \bullet \mathsf{bit0}'\}, \\ \mathsf{eps} \bullet \mathsf{dec} \rightarrowtriangle \{\mathsf{eps} \bullet \mathsf{fail}\}, \\ \mathsf{bit1} \bullet \mathsf{dec} \rightarrowtriangle \{\mathsf{bit0} \bullet \mathsf{ok}\}, \\ \mathsf{fail} \bullet \mathsf{bit0}' \rightarrowtriangle \{\mathsf{bit0} \bullet \mathsf{fail}\}, \\ \mathsf{ok} \bullet \mathsf{bit0}' \rightarrowtriangle \{\mathsf{bit1} \bullet \mathsf{ok}\} \end{array}$$

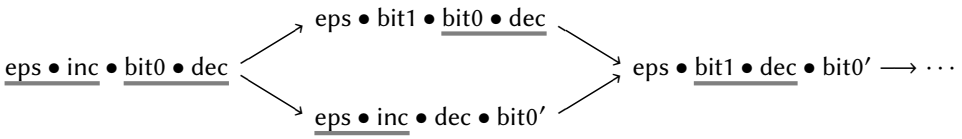
(b) Decrements

Figure 2: Summary of binary counter example

For example, the counter $\mathsf{eps} \bullet \mathsf{bit1} \bullet \mathsf{bit0} \bullet \mathsf{dec}$ can be maximally rewritten as

$$\begin{array}{l} \mathsf{eps} \bullet \mathsf{bit1} \bullet \underline{\mathsf{bit0} \bullet \mathsf{dec}} \\ \quad \longrightarrow \mathsf{eps} \bullet \underline{\mathsf{bit1} \bullet \mathsf{dec}} \bullet \mathsf{bit0}' \\ \quad \longrightarrow \mathsf{eps} \bullet \mathsf{bit0} \bullet \underline{\mathsf{ok} \bullet \mathsf{bit0}'} \\ \quad \longrightarrow \mathsf{eps} \bullet \mathsf{bit0} \bullet \mathsf{bit1} \bullet \mathsf{ok} \\ \quad \not\longrightarrow \end{array}$$

Once again, there are possibilities for concurrency. For example, the following two traces are indistinguishable because they differ only in the order of independent rewritings:



This justifies treating the two rewritings as concurrent.

Figure 2 summarizes the increment and decrement programs.

2.4 Infinite traces

Thus far, all traces have been finite, but this is not necessarily so. Consider adding an atom, incs, that generates a stream of inc atoms:

$$\mathsf{incs} \rightarrowtriangle \{\mathsf{inc} \bullet \mathsf{incs}\}.$$

Among the infinite traces now possible is

$$\mathsf{eps} \bullet \underline{\mathsf{incs}} \longrightarrow \underline{\mathsf{eps} \bullet \mathsf{inc}} \bullet \underline{\mathsf{incs}} \longrightarrow \cdots \longrightarrow \underline{\mathsf{eps} \bullet \mathsf{inc}} \bullet \mathsf{inc} \bullet \cdots \bullet \mathsf{inc} \bullet \underline{\mathsf{incs}} \longrightarrow \cdots.$$

Notice that this trace never rewrites the infinitely available $\mathsf{eps} \bullet \mathsf{inc}$ substring, instead always choosing to rewrite incs. Because of its scheduling bias against rewriting $\mathsf{eps} \bullet \mathsf{inc}$, we say that the trace is *(weakly) transition-unfair*.

Transition unfairness is fundamentally at odds with concurrency because it admits the possibility that one event precludes another, independent event—just as the persistent rewriting of incs precludes the rewriting of the independent $\mathsf{eps} \bullet \mathsf{inc}$ here. Therefore, we will implicitly assume that all traces are weakly transition-fair.

2.5 Technical details

The previous sections have hopefully provided an intuition for ordered logical specifications. In this section, we review the technical details, generally following the lead of Simmons's SLS framework (2012), but confining ourselves to a propositional fragment and using a weakly focused proof-construction strategy (Simmons and Pfenning 2011b) instead. The reader should feel free to skim or skip this section (especially if he is familiar with ordered logic), since the technical details are not critical to an understanding of the rest of this proposal.

2.5.1 Propositions, terms, and traces

Propositions. Propositions are polarized into *positive* and *negative* classes:

$$\begin{array}{ll} \text{Positive propositions} & A^+ ::= p^+ \mid A^- \mid A^+ \bullet B^+ \mid 1 \\ \text{Negative propositions} & A^- ::= A^+ \twoheadrightarrow B^- \mid A^+ \rightarrowtail B^- \mid A^- \mathbin{\&} B^- \mid \{A^+\} \end{array}$$

The negative propositions, A^-, are those whose right rules are invertible, whereas the strictly positive propositions, A^+ (but not A^-), are those whose left rules are invertible.

Positive atomic propositions, p^+, stand in for arbitrary positive propositions. Negative propositions, A^-, are implicitly included in positive ones. The lax modality, or monad, $\{A^+\}$, will be responsible for typing traces; it gives logical force to the separation of the two classes of propositions, yet still allows positive propositions to be explicitly included in negative ones.

Contexts. The set of all ordered contexts, Ω, forms the free monoid over the alphabet of hypotheses $x{:}A^+$ and $x{:}A^-$. Concatenation is written as Ω_1, Ω_2 and its unit is the empty context, $\cdot$.

$$\text{Ordered contexts} \quad \Omega ::= \cdot \mid x{:}A^+ \mid x{:}A^- \mid \Omega_1, \Omega_2$$

Notice that because the proof-construction strategy is weakly focused, non-atomic positive propositions, such as $A^+ \bullet B^+$, are indeed permissible hypotheses.

In addition to ordered contexts, we include *frames*. Frames, Θ, can be thought of as ordered contexts with one hole; i.e., think of Θ as $\Omega_L, \square, \Omega_R$, for some Ω_L and Ω_R. The hole may be filled with an ordered context, so that $\Theta\{\Omega\}$ is the ordered context $\Omega_L, \Omega, \Omega_R$. To ease the notational clutter in typing rules, we also allow contexts to be matched against filled frames. (We must be careful to distinguish the syntax for the monad, $\{A^+\}$, from filled frames, $\Theta\{\Omega\}$.)

The reader familiar with substructural logics will note that we have chosen not to include unrestricted contexts or persistent hypotheses. Specification clauses, such as $\mathsf{bit1} \bullet \mathsf{inc} \twoheadrightarrow \{\mathsf{inc} \bullet \mathsf{bit0}\}$, should indeed be persistent, but they are handled by signatures.

Signatures. Signatures Σ are collections of clauses. Each clause is a constant c with type of the form $p^+ \twoheadrightarrow A^-$.

$$\text{Signatures} \quad \Sigma ::= \cdot \mid \Sigma, c{:}p^+ \twoheadrightarrow A^-$$

$\boxed{T :: \Omega \longrightarrow^* \Omega'}$

$$\text{Traces} \quad T ::= \diamond \mid T_1; T_2 \mid S$$

$$\frac{}{\diamond :: \Omega \longrightarrow^* \Omega} \qquad \frac{T_1 :: \Omega \longrightarrow^* \Omega' \quad T_2 :: \Omega' \longrightarrow^* \Omega''}{T_1; T_2 :: \Omega \longrightarrow^* \Omega''} \qquad \frac{S :: \Omega \longrightarrow \Omega'}{S :: \Omega \longrightarrow^* \Omega'}$$

Figure 3: Traces

$\boxed{S :: \Omega \longrightarrow \Omega'}$

$$\text{Steps} \quad S ::= \{x\} \leftarrow R \mid y_1 \bullet y_2 \leftarrow x \mid 1 \leftarrow x$$

$$\frac{\Omega \vdash R : \langle \{A^+\} \rangle}{\{x\} \leftarrow R :: \Theta\{\Omega\} \longrightarrow \Theta\{x{:}A^+\}}$$

$$\frac{}{y_1 \bullet y_2 \leftarrow x :: \Theta\{x{:}A_1^+ \bullet A_2^+\} \longrightarrow \Theta\{y_1{:}A_1^+, y_2{:}A_2^+\}} \qquad \frac{}{1 \leftarrow x :: \Theta\{x{:}1\} \longrightarrow \Theta\{\cdot\}}$$

Figure 4: Steps

This is a slight departure from the example clauses in Sections 2.1 and 2.3 that were of the more general form $B^+ \multimap\!\!\triangleright A^-$. However, clauses of the form $p^+ \multimap\!\!\triangleright A^-$ can express the previous examples (in curried form) and will turn out to be a better fit for the choreographies to be introduced in Section 3.

Although it looks like a proposition, $p^+ \multimap\!\!\triangleright A^-$ is treated as a judgment that represents a persistent implication. Because the implication is persistent and therefore can be copied to any point in the context, $\multimap\!\!\triangleright$ is simultaneously both a left implication and a right implication.

Another way to think of $p^+ \multimap\!\!\triangleright A^-$ is as syntactic sugar for either $!(p^+ \twoheadrightarrow A^-)$ or $!(p^+ \rightarrowtail A^-)$, since both have the same derived rules in ordered logic.

Terms. As previously mentioned, a focused proof-construction strategy (Andreoli 1992) forms the basis of ordered logical specifications. Specifically, we choose a weakly focused strategy (Simmons and Pfenning 2011b). Each form of sequent in the weakly focused calculus corresponds to a syntactic class of terms.

We begin in Fig. 3 with traces, T, which are typed with a judgment $T :: \Omega \longrightarrow^* \Omega'$ that describes the change of state that the trace effects. Traces are either empty, $\diamond$, or the (nominally) sequential composition of two traces, $T_1; T_2$, or a single step, S. The empty trace effects no change to the state; the sequential composition $T_1; T_2$ first carries out trace T_1, and then continues with the remainder of the trace, T_2. A single step is also typed by the change of state that it effects: $S :: \Omega \longrightarrow \Omega'$ (Fig. 4). Each step is either a left-focusing phase, with term $\{x\} \leftarrow R$, or one of two inversion steps, with terms $y_1 \bullet y_2 \leftarrow x$ and $1 \leftarrow x$. In a left-focusing phase, if some piece of state Ω within

$\boxed{\Omega \vdash R : \langle C^- \rangle}$

$$\text{Atomic terms} \quad R ::= c \cdot (x; Sp) \mid x \cdot Sp$$

$$\frac{c{:}p^+ \multimap A^- \in \Sigma \quad \Theta\{[A^-]\} \vdash Sp : \langle C^- \rangle}{\Theta\{x{:}p^+\} \vdash c \cdot (x; Sp) : \langle C^- \rangle} \qquad \frac{\Theta\{[A^-]\} \vdash Sp : \langle C^- \rangle}{\Theta\{x{:}A^-\} \vdash x \cdot Sp : \langle C^- \rangle}$$

Figure 5: Atomic terms

$\boxed{\Theta\{[A^-]\} \vdash Sp : \langle C^- \rangle}$

$$\text{Spines} \quad Sp ::= \mathsf{nil} \mid V; Sp \mid \pi_1; Sp \mid \pi_2; Sp$$

$$\frac{}{[A^-] \vdash \mathsf{nil} : \langle A^- \rangle}$$

$$\frac{\Omega \vdash V : [A^+] \quad \Theta\{[B^-]\} \vdash Sp : \langle C^- \rangle}{\Theta\{[A^+ \twoheadrightarrow B^-], \Omega\} \vdash V; Sp : \langle C^- \rangle} \qquad \frac{\Omega \vdash V : [A^+] \quad \Theta\{[B^-]\} \vdash Sp : \langle C^- \rangle}{\Theta\{\Omega, [A^+ \rightarrowtail B^-]\} \vdash V; Sp : \langle C^- \rangle}$$

$$\frac{\Theta\{[A_1^-]\} \vdash Sp : \langle C^- \rangle}{\Theta\{[A_1^- \mathbin{\&} A_2^-]\} \vdash \pi_1; Sp : \langle C^- \rangle} \qquad \frac{\Theta\{[A_2^-]\} \vdash Sp : \langle C^- \rangle}{\Theta\{[A_1^- \mathbin{\&} A_2^-]\} \vdash \pi_2; Sp : \langle C^- \rangle}$$

Figure 6: Spines

$\Theta\{\Omega\}$ satisfies the requirements imposed by atomic term R of type $\{A^+\}$, then Ω is replaced with $x{:}A^+$—no inversion is performed because the proof-construction strategy is weakly focused. The inversion occurs as discrete steps that decompose non-atomic positive propositions.

The typing rules for atomic terms, R, according to the judgment $\Omega \vdash R : \langle C^- \rangle$ are shown in Fig. 5. These rules correspond to those for beginning a left-focusing phase from a stable sequent. Atomic terms consist of a head—either a constant c or variable x—followed by a spine, Sp, that completes the focusing phase.

Spines, as shown in Fig. 6, are either empty (nil), a value application followed by a spine ($V; Sp$), or a projection followed by a spine ($\pi_1; Sp$ or $\pi_2; Sp$). The spine typing judgment $\Theta\{[A^-]\} \vdash Sp : \langle C^- \rangle$ corresponds to the left-focused sequent form. Notice that spine typing can succeed even at non-atomic propositions, provided that the proposition under focus matches the consequent. Also, value applications require that values, V, be typed by positive propositions under focus.

The value typing judgment, $\Omega \vdash V : [A^+]$, is shown in Fig. 7. Of particular note is that normal terms, N, are included as values that type negative propositions that are included as positive ones.

Normal terms, N, are typed by negative propositions, A^-, under the judgment $\Omega \vdash N : A^-$ shown in Fig. 8. Each normal term is either an abstraction ($\lambda x.N$), a pair ($\langle N_1, N_2 \rangle$), or a trace capped by a value ($\{\text{let } T \text{ in } V\}$). The typing judgment for normal

$\boxed{\Omega \vdash V : [A^+]}$

$$\text{Values} \quad V ::= x \mid N \mid V_1 \bullet V_2 \mid 1$$

$$\frac{}{x{:}p^+ \vdash x : [p^+]} \qquad \frac{\Omega \vdash N : A^-}{\Omega \vdash N : [A^-]} \qquad \frac{\Omega_1 \vdash V_1 : [A_1^+] \quad \Omega_2 \vdash V_2 : [A_2^+]}{\Omega_1, \Omega_2 \vdash V_1 \bullet V_2 : [A_1^+ \bullet A_2^+]} \qquad \frac{}{\cdot \vdash 1 : [1]}$$

Figure 7: Values

$\boxed{\Omega \vdash N : A^-}$

$$\text{Normal terms} \quad N ::= \lambda x.N \mid \langle N_1, N_2 \rangle \mid \{\text{let } T \text{ in } V\}$$

$$\frac{\Omega, x{:}A^+ \vdash N : B^-}{\Omega \vdash \lambda x.N : A^+ \twoheadrightarrow B^-} \qquad \frac{x{:}A^+, \Omega \vdash N : B^-}{\Omega \vdash \lambda x.N : A^+ \rightarrowtail B^-} \qquad \frac{\Omega \vdash N_1 : A_1^- \quad \Omega \vdash N_2 : A_2^-}{\Omega \vdash \langle N_1, N_2 \rangle : A_1^- \mathbin{\&} A_2^-}$$

$$\frac{T :: \Omega \longrightarrow^* \Omega' \quad \Omega' \vdash V : [A^+]}{\Omega \vdash \{\text{let } T \text{ in } V\} : \{A^+\}}$$

Figure 8: Normal terms

terms corresponds to eager inversion on the right.

2.5.2 Concurrent equality

As described in Section 2.2, *concurrent equality* formalizes the idea that different interleavings of independent rewritings should be indistinguishable. Traces T thus form a trace monoid in which independent rewriting steps commute. Following Cervesato et al. (2012), this independence relation is defined on the sets of input and output variables, ${}^\bullet S$ and $S^\bullet$, of a step S.

Specifically, ${}^\bullet S$ and $S^\bullet$ are given by:

$$\begin{aligned} {}^\bullet(\{x\} \leftarrow R) &= \mathrm{FV}(R) & (\{x\} \leftarrow R)^\bullet &= \{x\} \\ {}^\bullet(y_1 \bullet y_2 \leftarrow x) &= \{x\} & (y_1 \bullet y_2 \leftarrow x)^\bullet &= \{y_1, y_2\} \\ {}^\bullet(1 \leftarrow x) &= \{x\} & (1 \leftarrow x)^\bullet &= \emptyset \end{aligned}$$

Consider the trace $S_1; S_2$. These two neighboring steps are independent and commute if ${S_1}^\bullet \cap {}^\bullet S_2 = \emptyset$. In other words, if ${S_1}^\bullet \cap {}^\bullet S_2 = \emptyset$, then $S_1; S_2 :: \Omega \longrightarrow^* \Omega''$ if and only if $S_2; S_1 :: \Omega \longrightarrow^* \Omega''$. (Notice that bound variables can always be renamed to be distinct from the input and output variables of previous steps in the trace.) Concurrent equality is the congruence relation on traces that is obtained by associativity and unit axioms and this partial commutativity.

3 Choreographies

As the binary counter from Sections 2.1 and 2.2 exemplifies, a notion of concurrency, based on indistinguishable interleavings of independent rewritings, arises naturally in ordered logical specifications. And, under a forward-chaining logic programming interpretation, these specifications can be executed by a central, omniscient "puppeteer" that rewrites the state globally.

In contrast, concurrency is traditionally phrased in formal calculi as the composition of *communicating processes*: processes are not omniscient but instead execute independently, with interaction between them limited to the exchange of messages. How are these two seemingly distinct notions of concurrency related? Are there processes hidden within ordered logical specifications—processes that would allow us to reconcile these two views?

Inspired by the process-as-formula view of linear logic (Miller 1992; Cervesato and Scedrov 2009), we propose that each atomic proposition in an ordered logical specification be assigned one (and only one) of two roles: an atom is either process-like or message-like. These role assignments, or *choreographies*, refine the original specification: whereas the original specification describes globally *what* are the valid interactions among atoms, a choreography[5] of that specification[6] describes *how* those interactions are achieved locally by asynchronous message passing.

For example, we might choose to choreograph increments in the binary counter specification by treating eps, bit0, and bit1 as process-like atoms and inc as a message-like atom. Using an arrow decoration to indicate the message-like atoms, the choreography clause

$$\mathsf{bit1} \bullet \underleftarrow{\mathsf{inc}} \multimap \{\underleftarrow{\mathsf{inc}} \bullet \mathsf{bit0}\}$$

specifies a bit1 process that, upon receiving an inc message at its right side, asynchronously sends an inc message to the left and then continues as a bit0 process.

Ideally, choreographies would be mechanically generated from specifications. But this appears to be quite difficult: Different specifications often require different patterns of communication, and therefore different choreographies. And some specifications even admit several choreographies, among which the programmer should choose the one that best fits the situation. Therefore, we'll assume that the programmer himself supplies the choreography.

It's also important to point out that not all role assignments are valid choreographies. We'll now describe what counts as a choreography, first with informal examples and then with formal definitions.

3.1 Choreographies by example

3.1.1 The binary counter

In giving the intuition behind the binary counter specification (Section 2.1), we described the inc atoms as moving up the counter. This hints at a choreography in which

[5] We borrow the term 'choreography' from the literature on session-based concurrency. The analogy is intended only as a loose one, however, and should not be taken to imply a precise, technical correspondence.

[6] Notice that a choreography is always relative to a given specification.

inc atoms act as messages that trigger the increment action: Whenever an inc message arrives at an eps, bit0, or bit1 process, that process takes responsibility for completing the increment action.

Expressed as an annotation of the original ordered logical specification, this choreography is:

$$\begin{aligned} \mathsf{eps} \bullet \underleftarrow{\mathsf{inc}} &\multimap \{\mathsf{eps} \bullet \mathsf{bit1}\} \\ \mathsf{bit0} \bullet \underleftarrow{\mathsf{inc}} &\multimap \{\mathsf{bit1}\} \\ \mathsf{bit1} \bullet \underleftarrow{\mathsf{inc}} &\multimap \{\underleftarrow{\mathsf{inc}} \bullet \mathsf{bit0}\}\,, \end{aligned}$$

where the eps, bit0, and bit1 atoms are viewed as processes, but the $\underleftarrow{\mathsf{inc}}$ atoms are viewed as messages.[7] We'll call this choreography our $\underleftarrow{\mathsf{inc}}$-choreography. Two properties are crucial:

Locality. Each clause's premise depends on exactly one process-like atom and (at most) one message-like atom. Consequently, each process's decisions are entirely local and deterministic: the eps, bit0, and bit1 processes act (independently) only after receiving an $\underleftarrow{\mathsf{inc}}$ message.[8]

Locality serves to ensure that the choreography describes sensible message-passing behaviors. A clause such as $\underleftarrow{\mathsf{inc}} \multimap \{\ldots\}$, whose premise does not contain a process-like atom, is not message-passing because no process is there to receive the $\underleftarrow{\mathsf{inc}}$ message.

Specification-preserving. The choreography exposes the same behaviors for eps, bit0, bit1, and inc as in the original specification. Its clauses are exactly those of the specification, except that each inc atom in the specification has been annotated as an $\underleftarrow{\mathsf{inc}}$ message-like atom in the choreography.

In this sense, there is a very strong, lock-step equivalence between the choreography and its specification. The choreography does not fundamentally alter the specification—it only refines that specification by making the communication patterns explicit.

3.1.2 Messages can flow in one of two directions

In our binary counter specification with decrements (Section 2.3), dec atoms propagate up the counter similarly to incs, with the difference that each dec atom eventually gives rise to either a fail or ok atom that travels back down the counter. Once again, this hints at a choreography in which dec, fail, and ok atoms are message-like:

- Whenever a $\underleftarrow{\mathsf{dec}}$ message arrives at an eps, bit0, or bit1 process's right-hand side, that process completes the local decrement action: the eps and bit1 processes send a $\underrightarrow{\mathsf{fail}}$ or $\underrightarrow{\mathsf{ok}}$ message, respectively, to their right; the bit0 process forwards the $\underleftarrow{\mathsf{dec}}$ message to its left and then continues as a bit0′ process.

[7] It's convenient to think of the programmer as supplying this choreography in full, but in practice the programmer might only give the assignment of roles to atoms, e.g. $\underleftarrow{\mathsf{inc}}$ for inc.

[8] In SSOS terminology, processes that wait to receive a message, like eps, bit0, and bit1 here, would be termed *latent* propositions; and messages, like $\underleftarrow{\mathsf{inc}}$ here, would be termed *passive* propositions (Pfenning and Simmons 2009).

- Whenever a $\underrightarrow{\text{fail}}$ or $\underrightarrow{\text{ok}}$ message arrives at a bit0′ process's left-hand side, that process forwards the message to its right-hand neighbor and then continues as a bit0 or bit1 process, respectively.

To account for decrements, the binary counter's choreography is therefore extended with the following clauses:

$$\begin{aligned}
&\text{eps} \bullet \underleftarrow{\text{dec}} \multimap \{\text{eps} \bullet \underrightarrow{\text{fail}}\} \\
&\text{bit0} \bullet \underleftarrow{\text{dec}} \multimap \{\underleftarrow{\text{dec}} \bullet \text{bit0}'\} \\
&\text{bit1} \bullet \underleftarrow{\text{dec}} \multimap \{\text{bit0} \bullet \underrightarrow{\text{ok}}\} \\
&\underrightarrow{\text{fail}} \bullet \text{bit0}' \multimap \{\text{bit0} \bullet \underrightarrow{\text{fail}}\} \\
&\underrightarrow{\text{ok}} \bullet \text{bit0}' \multimap \{\text{bit1} \bullet \underrightarrow{\text{ok}}\} .
\end{aligned}$$

Once again, these clauses are just an annotation of the original specification's clauses, with dec, ok, and fail annotated as $\underleftarrow{\text{dec}}$, $\underrightarrow{\text{ok}}$, and $\underrightarrow{\text{fail}}$. The extended choreography thus continues to be specification-preserving.

This extended choreography illustrates that each message atom has a unique direction, which is either left-directed, like $\underleftarrow{\text{inc}}$ and $\underleftarrow{\text{dec}}$, or right-directed, like $\underrightarrow{\text{fail}}$ and $\underrightarrow{\text{ok}}$. Because a left-directed message travels from right to left, it must always arrive at the right-hand side of its recipient; dually, a right-directed message must always arrive at the left-hand side of its recipient. This directionality is another aspect of locality, and it further constrains the structure of a choreography's premises. For example, this choreography's premises satisfy locality because each message flows toward its recipient, whereas premises of the forms $\underleftarrow{m_1} \bullet p$ or $p \bullet \underrightarrow{m_2}$ do not satisfy locality because the direction is wrong for process p to receive the $\underleftarrow{m_1}$ or $\underrightarrow{m_2}$ message.

3.1.3 Choreographies are not always unique

As alluded to previously, multiple choreographies are possible for some specifications. This is true of our binary counter specification, for instance. (To simplify the example, we'll ignore decrements.) In the $\underleftarrow{\text{inc}}$-choreography (Section 3.1.1), the counter's value is represented by a chain of eps, bit0, and bit1 processes that are acted upon by $\underleftarrow{\text{inc}}$ messages. Alternatively, the counter's value could be represented by a sequence of $\underrightarrow{\text{eps}}$, $\underrightarrow{\text{bit0}}$, and $\underrightarrow{\text{bit1}}$ messages that are fed to an inc process; the inc process would then emit a sequence that represents the result:

$$\begin{aligned}
&\underrightarrow{\text{eps}} \bullet \text{inc} \multimap \{\underrightarrow{\text{eps}} \bullet \underrightarrow{\text{bit1}}\} \\
&\underrightarrow{\text{bit0}} \bullet \text{inc} \multimap \{\underrightarrow{\text{bit1}}\} \\
&\underrightarrow{\text{bit1}} \bullet \text{inc} \multimap \{\text{inc} \bullet \underrightarrow{\text{bit0}}\} .
\end{aligned}$$

As required, this $\underrightarrow{\text{bit}}$-choreography possesses the locality and specification-preserving properties.

These two choreographies have distinct flavors, owing to the different process and message roles that they assign to the inc and eps, bit0, and bit1 atoms. The $\underleftarrow{\text{inc}}$-choreography has an object-oriented character: by sending an $\underleftarrow{\text{inc}}$ message, the increment method dispatches on the receiving object's class—either eps, bit0, or bit1.

In contrast, this new $\underrightarrow{\text{bit}}$-choreography has a functional character: inc is a function that receives its argument as a sequence of messages—either $\underrightarrow{\text{eps}}$, $\underrightarrow{\text{bit0}}$, or $\underrightarrow{\text{bit1}}$.

3.1.4 Two non-choreographies

Another, slightly more complex reformulation of the binary counter specification chooses to treat the inc atom as a simple process, not a message:

$$\begin{aligned}
&\text{inc} \multimap \{\underleftarrow{\text{incm}}\} \\
&\text{eps} \bullet \underleftarrow{\text{incm}} \multimap \{\text{eps} \bullet \text{bit1}\} \\
&\text{bit0} \bullet \underleftarrow{\text{incm}} \multimap \{\text{bit1}\} \\
&\text{bit1} \bullet \underleftarrow{\text{incm}} \multimap \{\text{inc} \bullet \text{bit0}\}\,.
\end{aligned}$$

In fact, the inc process does nothing but send an $\underleftarrow{\text{incm}}$ message.[9]

This signature is equivalent to the binary counter specification in that it ultimately exposes the same eps, bit, and inc behaviors. However, in contrast with the choreographies, this formulation does more than simply refine the specification by making the communication explicit: instead, it introduces $\underleftarrow{\text{incm}}$ as a new message-like atom, it introduces the clause $\text{inc} \multimap \{\underleftarrow{\text{incm}}\}$, and it modifies the premises of existing clauses to use $\underleftarrow{\text{incm}}$. So this signature is not specification-preserving—and therefore not a choreography—for the binary counter specification.

But that doesn't mean that the programmer cannot achieve the same behavior anyway: the programmer is free to rewrite the *specification* to incorporate that behavior at the specification level. If the specification is changed to be

$$\begin{aligned}
&\text{inc} \multimap \{\text{incm}\} \\
&\text{eps} \bullet \text{incm} \multimap \{\text{eps} \bullet \text{bit1}\} \\
&\text{bit0} \bullet \text{incm} \multimap \{\text{bit1}\} \\
&\text{bit1} \bullet \text{incm} \multimap \{\text{inc} \bullet \text{bit0}\}\,,
\end{aligned}$$

then the above signature is indeed a choreography for *this* specification.

Another signature that is equivalent to the binary counter specification, in the sense that the two track the same value, is the first-order signature

$$\text{num}\,N \bullet \underleftarrow{\text{inc}} \multimap \{\text{num}\,(N{+}1)\}\,.$$

Nevertheless, we wouldn't consider this to be a choreography of the binary counter specification because, by using a single number held by num instead of a string of bits, it fundamentally alters the specification. Once again, however, we would consider this signature to be a choreography of a different, simple counter specification, namely $\text{num}\,N \bullet \text{inc} \multimap \{\text{num}\,(N{+}1)\}$.

[9] Because it eagerly becomes $\underleftarrow{\text{incm}}$, the inc atom here would be termed an *active* proposition in SSOS terminology (Pfenning and Simmons 2009).

3.2 Choreographies, formally

Hopefully the preceding examples have given some intuition for what counts as a choreography. To make the definition precise, we need only formalize the locality and specification-preserving properties.

3.2.1 Locality

As discussed in Section 3.1.1, clauses such as $\mathsf{bit1} \bullet \underleftarrow{\mathsf{inc}} \multimap \{\underleftarrow{\mathsf{inc}} \bullet \mathsf{bit0}\}$ satisfy locality because the premise consists of a process that receives a message. Although it is possible to admit clauses like this one, they come at the expense of complicating the statement of locality slightly. Instead, it is more convenient to require that clauses be given in curried form, such as $\mathsf{bit1} \multimap (\underleftarrow{\mathsf{inc}} \twoheadrightarrow \{\underleftarrow{\mathsf{inc}} \bullet \mathsf{bit0}\})$.

For curried clauses, locality holds if there is exactly one process-like atom, p^+, in the premises and if any message-like atoms, $\underleftarrow{m}$ and $\underrightarrow{m}$, are used only with right- and left-ordered implications, respectively.

Definition 1 (Locality). A clause, which must have form $p^+ \multimap A^-$, satisfies *locality* if it adheres to the following refined grammar.[10]

$$\begin{array}{ll} \text{Negative propositions} & A^-, B^- ::= \{A^+\} \mid \&_{i \in I}(\underleftarrow{m}_i^+ \twoheadrightarrow \{A_i^+\}) \mid \&_{i \in I}(\underrightarrow{m}_i^+ \rightarrowtail \{A_i^+\}) \\ \text{Positive propositions} & A^+, B^+ ::= A^+ \bullet B^+ \mid 1 \mid \underleftarrow{m}^+ \bullet A^+ \mid A^+ \bullet \underrightarrow{m}^+ \mid p^+ \mid A^- \end{array}$$

A signature Σ satisfies *locality* if each of its clauses, $p^+ \multimap A^-$, satisfies locality and if there is a unique clause $p^+ \multimap A^-$ for each process-like atom p^+ that appears in Σ.

Notice that, for technical reasons related to the translation to session-typed processes (Section 5), this definition also requires that message-like atoms $\underleftarrow{m}_1$ and $\underrightarrow{m}_2$ never appear in clause heads except as part of a conjunction—either $\underleftarrow{m}_1 \bullet A^+$ or $A^+ \bullet \underrightarrow{m}_2$. Because 1 is available, this requirement does not restrict expressiveness however. Also notice that clause heads can emit several messages in both directions, such as $(\underleftarrow{m}_1^+ \bullet p^+) \bullet \underrightarrow{m}_2^+$; clause heads also allow clauses to be higher-order, since positive propositions can include A^-.

As a consequence of this document's focus on *well-typed* processes, also notice the more fundamental omission of propositions like $(\underrightarrow{m}_1^+ \rightarrowtail \{A_1^+\}) \,\&\, (\underleftarrow{m}_2^+ \twoheadrightarrow \{A_2^+\})$. Intuitively, such a proposition corresponds to an input-guarded nondeterministic choice—the process chooses either to receive $\underrightarrow{m}_1^+$ from the left and continue as process A_1^+, or to receive $\underleftarrow{m}_2^+$ from the right and continue as process A_2^+. Nondeterministic choice is not typable in the current SILL typing scheme of Caires et al. (2013). These propositions are ruled out by requiring all arms of an additive conjunction to be consistently left- or right-implications.

Likewise, notice that $p^+ \multimap (\underrightarrow{m}_1^+ \rightarrowtail (\underleftarrow{m}_2^+ \twoheadrightarrow \{A^+\}))$ is not included in the above grammar because it would correspond to a demand that both messages be received (in some nondeterministic order) before continuing with A^+. The above grammar instead only allows $p^+ \multimap (\underrightarrow{m}_1^+ \rightarrowtail \{\underleftarrow{m}_2^+ \twoheadrightarrow \{A^+\}\})$ with the monad introducing a pause that requires $\underrightarrow{m}_1^+$ to be received before continuing.

[10] An alternative notation for $\&_{i \in I}(\underleftarrow{m}_i^+ \twoheadrightarrow \{A_i^+\})$ could be the Lambek-inspired $\&_{i \in I}(\{A_i\} \twoheadleftarrow \underleftarrow{m}_i^+)$, which would better emphasize that left-directed messages, $\underleftarrow{m}_i^+$, must arrive from the right.

With some simple transformations, the choreographies presented earlier do indeed satisfy locality. For example, the $\underleftarrow{\mathsf{inc}}$-choreography can be put into the curried form

$$\begin{aligned}&\mathsf{eps} \multimap (\underleftarrow{\mathsf{inc}} \twoheadrightarrow \{\mathsf{eps} \bullet \mathsf{bit1}\})\\ &\mathsf{bit0} \multimap (\underleftarrow{\mathsf{inc}} \twoheadrightarrow \{\mathsf{bit1}\})\\ &\mathsf{bit1} \multimap (\underleftarrow{\mathsf{inc}} \twoheadrightarrow \{\underleftarrow{\mathsf{inc}} \bullet \mathsf{bit0}\})\,.\end{aligned}$$

Likewise, after some transformations, the $\underrightarrow{\mathsf{bit}}$-choreography also satisfies locality:

$$\mathsf{inc} \multimap \begin{pmatrix} (\underrightarrow{\mathsf{eps}} \rightarrowtail \{(1 \bullet \underrightarrow{\mathsf{eps}}) \bullet \underrightarrow{\mathsf{bit1}}\}) \\ \& (\underrightarrow{\mathsf{bit0}} \rightarrowtail \{1 \bullet \underrightarrow{\mathsf{bit1}}\}) \\ \& (\underrightarrow{\mathsf{bit1}} \rightarrowtail \{\mathsf{inc} \bullet \underrightarrow{\mathsf{bit0}}\}) \end{pmatrix}$$

However, notice that a clause $\underrightarrow{m_1}^+ \bullet p^+ \bullet \underleftarrow{m_2}^+ \multimap \{A^+\}$ *cannot* be put into a form that satisfies locality. Its closest curried form is $p^+ \multimap (\underrightarrow{m_1}^+ \rightarrowtail (\underleftarrow{m_2}^+ \twoheadrightarrow \{A^+\}))$, but even that form does not adhere to the grammar for local clauses because $\rightarrowtail$ is not followed by a $\{\cdots\}$.

3.2.2 Specification-preserving

To judge that a signature is specification-preserving, we rely on a notion of erasure that removes the assigned roles, translating message- and process-like atoms to ordinary atoms.

Definition 2 (Role erasure). For atomic propositions, the *role erasure* $(-)^e$ is given by

$$\begin{aligned}(\underleftarrow{m}^+)^e &= m^+ = (\underrightarrow{m}^+)^e\\ (p^+)^e &= p^+\,.\end{aligned}$$

Role erasures for propositions and contexts, $(A^+)^e$, $(A^-)^e$, and $(\Omega)^e$, are defined compositionally, lifting role erasure for atoms.

For instance, the role erasure of $\underleftarrow{\mathsf{inc}}$ is $(\underleftarrow{\mathsf{inc}})^e = \mathsf{inc}$, matching the intuition that the message-like atom $\underleftarrow{\mathsf{inc}}$ in the $\underleftarrow{\mathsf{inc}}$-choreography (Section 3.1.1) serves to implement the specification's inc atom.

Using role erasure, we can define the specification-preserving property as follows:

Definition 3 (Specification-preserving). A signature X is *specification-preserving* for the specification Σ if: $\Omega \longrightarrow_X \Omega'$ under the signature X if and only if $(\Omega)^e \longrightarrow_\Sigma (\Omega')^e$ under the specification Σ. In other words, X is specification-preserving for Σ if $(-)^e$ is a bisimulation.

Thus, to be specification-preserving, a choreography must be lock-step equivalent with its specification. For example, the $\underleftarrow{\mathsf{inc}}$-choreography for the binary counter specification is indeed specification-preserving because the two are lock-step equivalent. Figure 9 shows how the steps correspond using bisimulation diagrams.

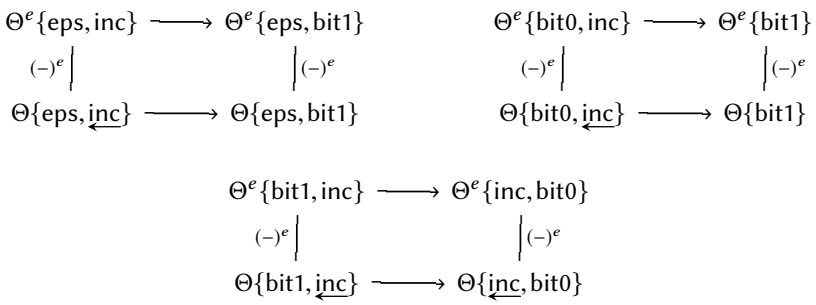


Figure 9: The $\underleftarrow{\mathsf{inc}}$-choreography of Section 3.1.1 is specification-preserving.

4 Session-typed processes from singleton linear logic

Thus far, we have followed a proof-construction approach to computation, having in Section 2 reviewed an interpretation of ordered logical specifications as concurrent string rewriting and in Section 3 identified a fragment in which those specifications have a message-passing character. In this section, we turn to a proof-reduction view of computation.

Recently, Caires and Pfenning (2010) with Toninho (2013) have established a Curry–Howard isomorphism, dubbed SILL, between the sequent calculus for intuitionistic linear logic and a session-typed π-calculus, in which propositions are session types, proofs are session-typed processes, and cut reductions are process reductions.[11] This gives a proof-reduction view of concurrency that differs, apparently substantially, from the proof-construction perspective. But, by the end of this proposal document, we will have shown that the differences are not as substantial as they first appear.

In this section, we present a reformulation of Caires et al.'s SILL for a restriction of intuitionistic linear logic, which we call singleton linear logic, that is a better fit for comparisons with ordered logical specifications.

4.1 Toward singleton linear logic

In a session-based model of concurrency, pairs of processes interact in well-defined sessions, with one process offering a service that its session partner uses. Session types, pioneered by Honda (1993), describe the interaction protocol to which a process adheres when offering its service. When processes interact, the session type changes: one process now offers, and the other uses, the continuation of the initial service. As shown by Caires et al. (2013), the logical reading of session-based concurrency is linear logic exactly because it can express this change of state.

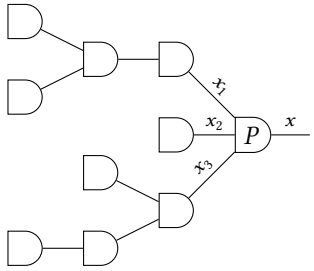
[11] Wadler (2014) later developed a correspondence between classical linear logic and a session-typed π-calculus, but Caires et al.'s intuitionistic correspondence turns out to be better suited to our goals here, for reasons we will explain shortly.

Because a process offers its service along a distinguished channel, the basic session-typing judgment of Caires et al.'s SILL is $P :: x{:}A$, meaning "process P offers a service of session type A along channel x". However, P itself may rely on services offered by yet other processes, and so, more generally, the SILL session-typing judgment is a linear sequent annotated as

$$\underbrace{x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n}_{\Delta} \vdash P :: x{:}A \quad (n \geq 0)\,,$$

meaning "Using services A_i offered along channels x_i, the process P offers service A along channel x." (The channels x_i and x must all be distinct and are binding occurrences with scope over the process P.)

In SILL, the linear sequent calculus's inference rules thus become session-typing rules for processes. Just as the inference rules arrange sequents into a proof tree, so do the SILL session-typing rules arrange processes into a tree-shaped network in which some processes are clients of more than one process (i.e., some nodes have more than one child). The following is a snapshot taken during the execution of one such tree-shaped process network.

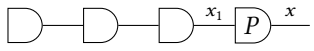


In this thesis proposal, we are interested in a restriction of SILL that will match concurrent ordered logical specifications. To match the ordered context of these specifications, we cannot directly use SILL—at least not in its full generality. Instead, we need a restriction of SILL in which process networks are chains, not arbitrary trees.

One might expect that process chains should arise from *ordered* logic. But, taking into account the way that the structure of the SILL session-typing judgment induces a tree-shape for process networks, it becomes apparent that process chains, in fact, arise when processes are restricted to use at most one service each—that is, when contexts Δ are restricted to be either empty or singletons. The session-typing judgment becomes

$$\cdot \vdash P :: x{:}A \quad \text{or} \quad x_1{:}A_1 \vdash P :: x{:}A$$

and the process networks are necessarily chains:



Having made this restriction, we can simplify the judgments: if there are at most two channels, they can always be unambiguously named "left" and "right", rather than

bothering with fresh names like x_1 and x. Moreover, since the channel names are now fixed by position (rather like de Bruijn indices), we may as well omit them altogether from the session-typing judgments:

$$\cdot \vdash P :: A \quad \text{or} \quad A_1 \vdash P :: A\,.$$

The following sections describe this restriction of SILL. It's worth emphasizing that although we choose to present the logical rules and process assignment simultaneously, singleton linear logic is indeed a well-defined logic in its own right: even in the presence of the restriction to singleton antecedents, we still have a Curry–Howard isomorphism.

4.2 Cut as composition

The cut rule of intuitionistic linear logic composes a plan for obtaining resource A with another plan that uses resource A:

$$\frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\,.$$

When antecedents are restricted to be either empty or singletons, that rule becomes the cut rule for singleton linear logic; it retains the character of a composition:

$$\frac{\Delta \vdash A \quad A \vdash C}{\Delta \vdash C}\ \textsc{cut}_A\,.$$

Because proofs are to be processes, this suggests that the process interpretation of the cut rule should compose a process that offers service A with another process that uses service A. The $\textsc{cut}_A$ rule thus becomes a typing rule for process composition:

$$\frac{\Delta \vdash P :: A \quad A \vdash Q :: C}{\Delta \vdash \mathsf{spawn}\, P; Q :: C}\ \textsc{cut}_A\,,$$

where spawn $P; Q$ means "Spawn process P to the left, and then continue as process Q."

To complete the description of spawn, we must make its operational semantics precise. Rather than using a structural operational semantics, it is convenient to describe the semantics as an ordered logical specification (Pfenning 2004; Pfenning and Simmons 2009), in the style known as a substructural operational semantics (SSOS). In our SSOS, we will use the atomic proposition exec P to represent an executing process P. The rule for executing a spawn is

$$\mathsf{exec}\,(\mathsf{spawn}\, P; Q) \multimap \{\mathsf{exec}\, P \bullet \mathsf{exec}\, Q\}\,.$$

Thus, to execute the spawn, we execute the processes P and Q side-by-side, with process P offering a service that Q uses.

4.3 Additive conjunction as branching

So far we have discussed only cut, a judgmental principle that applies to all services[12]; specific services are defined by the right and left rules of the logical connectives.

The singleton linear sequent calculus rules for the additive conjunction $A_1 \mathbin{\&} A_2$ are as follows. Other than the restriction to singleton or empty antecedents, these rules are the same as those from linear logic.

$$\frac{\Delta \vdash A_1 \quad \Delta \vdash A_2}{\Delta \vdash A_1 \mathbin{\&} A_2}\ \&\mathrm{R} \qquad \frac{A_1 \vdash C}{A_1 \mathbin{\&} A_2 \vdash C}\ \&\mathrm{L}_1 \qquad \frac{A_2 \vdash C}{A_1 \mathbin{\&} A_2 \vdash C}\ \&\mathrm{L}_2$$

The right rule, $\&\mathrm{R}$, says that to prove $A_1 \mathbin{\&} A_2$ we must prove both A_1 and A_2 (using the same resource Δ) so that we are prepared for whichever of the two resources, A_1 or A_2, is eventually chosen by a $\&\mathrm{L}_1$ or $\&\mathrm{L}_2$ left rule.

Correspondingly, a process that offers service $A_1 \mathbin{\&} A_2$ gives its client a choice of services A_1 and A_2; the process must be prepared to offer whichever service the client chooses. Based on this intuition, we interpret the $\&\mathrm{R}$ rule as typing a binary guarded choice:

$$\frac{\Delta \vdash P_1 :: A_1 \quad \Delta \vdash P_2 :: A_2}{\Delta \vdash \mathsf{caseR}\,(\mathsf{in}_1 \Rightarrow P_1 \parallel \mathsf{in}_2 \Rightarrow P_2) :: A_1 \mathbin{\&} A_2}\ \&\mathrm{R}$$

where $\mathsf{caseR}\,(\mathsf{in}_1 \Rightarrow P_1 \parallel \mathsf{in}_2 \Rightarrow P_2)$ means "Input either in_1 or in_2 along the right-hand channel, and then continue as process P_1 or P_2, respectively."

Conversely, the client sitting to the right that uses service $A_1 \mathbin{\&} A_2$ must behave in a complementary way: the client should select either service A_1 or service A_2 and then, having notified the offering process of its choice (as in_1 or in_2), continue the session by using that service. The left rules for type $A_1 \mathbin{\&} A_2$ are thus:

$$\frac{A_1 \vdash Q :: C}{A_1 \mathbin{\&} A_2 \vdash \mathsf{selectL}\,\mathsf{in}_1; Q :: C}\ \&\mathrm{L}_1 \qquad \frac{A_2 \vdash Q :: C}{A_1 \mathbin{\&} A_2 \vdash \mathsf{selectL}\,\mathsf{in}_2; Q :: C}\ \&\mathrm{L}_2$$

where $\mathsf{selectL}\,\mathsf{in}_{1/2}; Q$ means "Send label $\mathsf{in}_{1/2}$ along the left-hand channel and then continue as process Q."

Our intuition about the behavior of the guarded choice processes is made precise by their asynchronous operational semantics. First, the in_1 branch:

$$\begin{aligned}&\mathsf{exec}\,(\mathsf{selectL}\,\mathsf{in}_1; Q) \multimap \{\mathsf{msgL}\,\mathsf{in}_1 \bullet \mathsf{exec}\,Q\}\\ &\mathsf{exec}\,(\mathsf{caseR}\,(\mathsf{in}_1 \Rightarrow P_1 \parallel \mathsf{in}_2 \Rightarrow P_2)) \bullet \mathsf{msgL}\,\mathsf{in}_1 \multimap \{\mathsf{exec}\,P_1\}\,.\end{aligned}$$

To execute the selection process $\mathsf{selectL}\,\mathsf{in}_1; Q$, we asynchronously send to the left a message containing the label in_1, which is represented in the SSOS as the proposition $\mathsf{msgL}\,\mathsf{in}_1$, and then immediately continue the session by executing process Q. When this message arrives, the destination process $\mathsf{caseR}\,(\mathsf{in}_1 \Rightarrow P_1 \parallel \mathsf{in}_2 \Rightarrow P_2)$ resumes execution as P_1.

The operational semantics of the in_2 branch is symmetric to that of the in_1 branch:

$$\begin{aligned}&\mathsf{exec}\,(\mathsf{selectL}\,\mathsf{in}_2; Q) \multimap \{\mathsf{msgL}\,\mathsf{in}_2 \bullet \mathsf{exec}\,Q\}\\ &\mathsf{exec}\,(\mathsf{caseR}\,(\mathsf{in}_1 \Rightarrow P_1 \parallel \mathsf{in}_2 \Rightarrow P_2)) \bullet \mathsf{msgL}\,\mathsf{in}_2 \multimap \{\mathsf{exec}\,P_2\}\,.\end{aligned}$$

[12]The other judgmental principle, identity, is postponed to Section 4.8 in the interest of presenting only what is necessary for a first, simple example.

Practical considerations. To make the language more palatable for the programmer, we diverge slightly from a pure propositions-as-types interpretation of singleton linear logic by including n-ary labeled additive conjunctions, $\&\{\ell : A_\ell\}_{\ell \in L}$, as a primitive.

By analogy with the binary conjunction, the typing rules and operational semantics for $\&\{\ell : A_\ell\}_{\ell \in L}$ are as follows. Notice that the $\&\mathrm{R}$ has a set of premises, one for each label $\ell \in L$.

$$\frac{\forall \ell \in L\colon\ \ \Delta \vdash P_\ell :: A_\ell}{\Delta \vdash \mathsf{caseR}_{\ell \in L}\,(\ell \Rightarrow P_\ell) :: \&\{\ell : A_\ell\}_{\ell \in L}}\ \&\mathrm{R} \qquad \frac{A_k \vdash Q :: C \quad (k \in L)}{\&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{selectL}\,k; Q :: C}\ \&\mathrm{L}$$

$$\begin{aligned} &\mathsf{exec}\,(\mathsf{selectL}\,k; Q) \multimap \{\mathsf{msgL}\,k \bullet \mathsf{exec}\,Q\} \\ &\mathsf{exec}\,(\mathsf{caseR}_{\ell \in L}\,(\ell \Rightarrow P_\ell)) \bullet \mathsf{msgL}\,k \multimap \{\mathsf{exec}\,P_k\} \qquad (k \in L) \end{aligned}$$

Another possibility would be to simply treat n-ary labeled conjunctions as syntactic sugar for nested binary conjunctions, but this would introduce a communication overhead because we would be sending multiple $\mathsf{in}_{1/2}$s separately rather than a single label k.

4.4 Recursive session types and process definitions

Concurrent processes frequently exhibit unbounded or infinite, yet well-defined, behavior; for instance, we may wish to have a counter that offers an increment service indefinitely. Toninho et al. (2014) have proposed an extension of their SILL type theory that incorporates inductive and coinductive session types. However, to keep matters simpler, we instead rely on general recursion here and choose to be content with the rather ad hoc departure from a pure Curry–Howard isomorphism.

Session types thus include general recursive types, $\mu t.A$, and type variables, t. The type $\mu t.A$ is interpreted equi-recursively, being identified with the unfolding $[(\mu t.A)/t]A$. Processes correspondingly include mutually recursive process definitions, via `letrec`, and process variables, X.

We extend the session-typing judgment with a context, Ψ, of process-variable typings. Because a process is typed according with a sequent of services that it uses and offers, process variables are typed as $X : \{\Delta \vdash A\}$ if process X can offer service A by using services Δ. When channels of appropriate types are available, the process X can be called:

$$\frac{}{\Psi, X{:}\{\Delta \vdash A\}\ ;\ \Delta \vdash \mathsf{call}\,X :: A}\ \textsc{call}\,.$$

A common idiom is spawn $\mathsf{call}\,X; Q$, which spawns a call to X that is run in parallel with some process Q. We will sometimes abbreviate this with the syntactic sugar $\mathsf{spawn}\,X; Q$ or even $X; Q$.

In mutually recursive process definitions, the process bodies may refer to any of the mutually recursive processes via process variables. The typing rule is

$$\frac{(\Psi' = X_1{:}\{\Delta_1 \vdash A_1\}, \ldots, X_n{:}\{\Delta_n \vdash A_n\} = P_n) \quad \forall i\colon\ \ \Psi, \Psi'\ ;\ \Delta_i \vdash P_i :: A_i \quad \Psi, \Psi'\ ;\ \Delta \vdash Q :: C}{\Psi\ ;\ \Delta \vdash \mathsf{letrec}\,X_1{:}\{\Delta_1 \vdash A_1\} = P_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ X_n{:}\{\Delta_n \vdash A_n\} = P_n\ \mathsf{in}\,Q :: C}\ \textsc{letrec}\,.$$

Listing 1: A simple binary counter supporting an increment operation

```
stype Cntr = &{ inc: Cntr }

eps : { |- Cntr } =
{ caseR of
    inc => eps; bit1 }

bit0 : { Cntr |- Cntr } =
{ caseR of
    inc => bit1 }

bit1 : { Cntr |- Cntr } =
{ caseR of
    inc => selectL inc; bit0 }
```

$$\frac{(\Psi' = \{X{:}\{\Delta_X \vdash A_X\}\}_{X\in\mathcal{X}}) \quad \forall X \in \mathcal{X}\colon\ \Psi, \Psi' \,;\, \Delta_X \vdash P_X :: A_X \quad \Psi, \Psi' \,;\, \Delta \vdash Q :: C}{\Psi \,;\, \Delta \vdash \mathsf{letrec}_{X\in\mathcal{X}}\,(X{:}\{\Delta_X \vdash A_X\} = P_X)\,\mathsf{in}\,Q :: C}\ \textsc{letrec}\,.$$

The operational semantics of these constructs are as follows:

$$\begin{aligned}&\mathsf{exec}\,(\mathsf{letrec}\,X_1{:}\{\Delta_1 \vdash A_1\} = P_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ X_n{:}\{\Delta_n \vdash A_n\} = P_n\ \mathsf{in}\,Q)\\ &\quad \multimap \{!\mathsf{bnd}\,X_1\,P_1 \bullet \ldots \bullet\, !\mathsf{bnd}\,X_n\,P_n \bullet \mathsf{exec}\,Q\}\\ &!\mathsf{bnd}\,X\,P \bullet \mathsf{exec}\,(\mathsf{call}\,X) \multimap \{\mathsf{exec}\,P\}\end{aligned}$$

The environment of bindings of process variables to process expressions is represented in the SSOS as a collection of $\mathsf{bnd}\,X\,P$ hypotheses. To execute a group of mutually recursive process definitions, bindings are introduced for each of the process variables and then the body of the letrec is executed. To execute a free process variable X, instead execute the process expression to which X is bound.

Having presented recursion, we can finally give a simple example program.

4.5 Example: Binary counter

We can implement a simple session-typed counter on natural numbers as shown in listing 1.[13] The counter is a chain of `bit0` and `bit1` processes, one for each bit in the binary representation of the counter's value, and is terminated at the most significant end with an `eps` process. For instance, the process chain `eps; bit1; bit0` represents a counter with value 2.

The counter offers a very simple service: the client may only choose to increment the counter, with the same service being offered recursively after the increment. This

[13] This example is adapted from one by Toninho et al. (2013).

service, `Cntr`, is therefore a recursive (unary) additive conjunction, declared in the concrete syntax as `stype Cntr = &{ inc: Cntr }`. The `eps` process offers this service outright, and thus has type `{ |- Cntr }`. The `bit0` and `bit1` processes, on the other hand, use the service offered by their more significant neighbors, and thus have type `{ Cntr |- Cntr }`.

The process definitions of `eps`, `bit0`, and `bit1` are mutually recursive. When an `eps` process receives an `inc` message, it creates a new most significant bit by spawning a new `eps` process and then making a recursive call to a `bit1` process. When a `bit0` process receives an `inc`, the bit is flipped by virtue of a recursive call to a `bit1` process. Lastly, when a `bit1` process receives an `inc`, the bit is flipped and a carry is propagated; this is accomplished by first sending `inc` along the left to the `Cntr` offered by the next more significant bit and then making a recursive call to a `bit0` process.

Informally, we can see that, as implemented, the `inc` operation respects a counter's denotation: whenever a counter representing natural number N is incremented, the resulting counter represents $N + 1$. Note, however, that this adequacy property is not enforced by the type `Cntr`. An appropriate dependent session type could enforce increment adequacy, but, for simplicity of exposition, we prefer the simple type here.

4.6 Additive disjunction as choice

In the singleton linear sequent calculus, additive disjunction, $A \oplus B$, is dual to additive conjunction, $A \mathbin{\&} B$.

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ \oplus\mathrm{R}_1 \qquad \frac{\Delta \vdash A_2}{\Delta \vdash A_1 \oplus A_2}\ \oplus\mathrm{R}_2 \qquad \frac{A_1 \vdash C \quad A_2 \vdash C}{A_1 \oplus A_2 \vdash C}\ \oplus\mathrm{L}$$

We should expect this duality to also appear in the process assignment. Whereas a process of type $A \mathbin{\&} B$ offers its client at the right a choice of services A and B, a process of type $A \oplus B$ chooses between offering service A or service B to its client at the right. The client waits to be notified of the offering process's choice and then uses that service.

$$\frac{\Delta \vdash P :: A_1}{\Delta \vdash \mathsf{selectR}\,\mathsf{in}_1; P :: A_1 \oplus A_2}\ \oplus\mathrm{R}_1 \qquad \frac{\Delta \vdash P :: A_2}{\Delta \vdash \mathsf{selectR}\,\mathsf{in}_2; P :: A_1 \oplus A_2}\ \oplus\mathrm{R}_2$$

$$\frac{A_1 \vdash Q_1 :: C \quad A_2 \vdash Q_2 :: C}{A_1 \oplus A_2 \vdash \mathsf{caseL}\,(\mathsf{in}_1 \Rightarrow Q_1 \,[]\, \mathsf{in}_2 \Rightarrow Q_2) :: C}\ \oplus\mathrm{L}$$

Confirming the intuition that $\mathsf{selectR}\,\mathsf{in}_{1/2}; P$ sends along the right-hand channel and $\mathsf{caseL}\,(\mathsf{in}_1 \Rightarrow Q_1 \,[]\, \mathsf{in}_2 \Rightarrow Q_2)$ receives along the left-hand channel are the SSOS rules:

$$\begin{aligned}
&\mathsf{exec}\,(\mathsf{selectR}\,\mathsf{in}_1; P) \multimap \{\mathsf{exec}\,P \bullet \mathsf{msgR}\,\mathsf{in}_1\}\\
&\mathsf{msgR}\,\mathsf{in}_1 \bullet \mathsf{exec}\,(\mathsf{caseL}\,(\mathsf{in}_1 \Rightarrow Q_1 \,[]\, \mathsf{in}_2 \Rightarrow Q_2)) \multimap \{\mathsf{exec}\,Q_1\}\\
&\mathsf{exec}\,(\mathsf{selectR}\,\mathsf{in}_2; P) \multimap \{\mathsf{exec}\,P \bullet \mathsf{msgR}\,\mathsf{in}_2\}\\
&\mathsf{msgR}\,\mathsf{in}_2 \bullet \mathsf{exec}\,(\mathsf{caseL}\,(\mathsf{in}_1 \Rightarrow Q_1 \,[]\, \mathsf{in}_2 \Rightarrow Q_2)) \multimap \{\mathsf{exec}\,Q_2\}\,.
\end{aligned}$$

Because the operational semantics is asynchronous, it's important to distinguish the msgR predicate, which represents messages that flow to the right, from the msgL

Listing 2: A binary counter supporting increments and decrements

```
stype Cntr = &{ inc: Cntr , dec: Cntr' }
  and Cntr' = +{ ok: Cntr , fail: Cntr }

eps : { |- Cntr } =
{ caseR of
    inc => eps; bit1
  | dec => selectR fail; eps }

bit0 : { Cntr |- Cntr } =
{ caseR of
    inc => bit1
  | dec => selectL dec; bit0' }

bit0' : { Cntr' |- Cntr' } =
{ caseL of
    ok => selectR ok; bit1
  | fail => selectR fail; bit0 }

bit1 : { Cntr |- Cntr } =
{ caseR of
    inc => selectL inc; bit0
  | dec => selectR ok; bit1 }
```

predicate, which represents messages that flow to the left. Otherwise, a selection process's continuation could mistakenly capture the message that was just sent, as might happen in executing $\mathsf{selectR}\,\mathsf{in}_1; \mathsf{caseR}\,(\mathsf{in}_1 \Rightarrow P_1 \,[]\, \mathsf{in}_2 \Rightarrow P_2)$, for example.

Once again, to make the language more convenient for the programmer, we include n-ary labeled additive disjunctions $\oplus\{\ell : A_\ell\}_{\ell \in L}$. The typing rules and operational semantics are thus more generally

$$\frac{\Delta \vdash P :: A_k \quad (k \in L)}{\Delta \vdash \mathsf{selectR}\,k; P :: \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus\mathrm{R} \qquad \frac{\forall \ell \in L\colon\ A_\ell \vdash Q_\ell :: C}{\oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{caseL}_{\ell \in L}\,(\ell \Rightarrow Q_\ell) :: C}\ \oplus\mathrm{L}$$

$$\mathsf{exec}\,(\mathsf{selectR}\,k; P) \rightarrowtriangle \{\mathsf{exec}\,P \bullet \mathsf{msgR}\,k\}$$
$$\mathsf{msgR}\,k \bullet \mathsf{exec}\,(\mathsf{caseL}_{\ell \in L}\,(\ell \Rightarrow Q_\ell)) \rightarrowtriangle \{\mathsf{exec}\,Q_k\} \qquad (k \in L)$$

4.7 Example: Binary counter with decrements

Listing 2 shows a counter that takes advantage of additive disjunction to support a truncated decrement operation. According to the type declaration, a process offering the `Cntr` service gives its client a choice of increment or decrement services. If the

client chooses to decrement, the offering process will choose to reply with either `fail` or `ok` and then recursively offer the `Cntr` service.

As implemented, decrementing the counter gives `fail` and leaves the process network unchanged if the counter represents 0; if it represents some $N > 0$, then decrementing the counter gives `ok` after decrementing to $N - 1$. Once again, these adequacy properties are not enforced by the type `Cntr`, although they could be with an appropriate dependent session type.

4.8 Identity as forwarding

In singleton linear logic, in addition to the cut principle, there is an identity principle that states that one way to obtain a resource is to directly use an existing resource:

$$\frac{}{A \vdash A}\ \textsc{id}_A\,.$$

Under the process interpretation, a process can offer service A by acting as a forwarding intermediary between its clients and another process that offers service A. The $\textsc{id}_A$ rule thus types a forwarding process between two channels:

$$\frac{}{A \vdash \leftrightarrow :: A}\ \textsc{id}_A\,.$$

Rather than making the forwarding explicit in the operational semantics, we can simply eliminate the middleman, adjoining the neighboring processes:

$$\mathsf{exec}\,(\leftrightarrow) \multimap \{1\}\,.$$

4.9 Other session types

In addition to the those already mentioned, the other connectives of singleton linear logic correspond to session types.

Multiplicative unit. Like its SILL cousin, the multiplicative unit 1 is the service that terminates without any interaction. Its right rule, 1R, types a process that immediately terminates; its left rule, 1L, types a process that waits for the left-hand side to terminate:

$$\frac{}{\cdot \vdash \mathsf{closeR} :: 1}\ 1\textsc{r} \qquad\qquad \frac{\cdot \vdash Q :: C}{1 \vdash \mathsf{waitL}; Q :: C}\ 1\textsc{l}$$

The operational semantics is asynchronous, with the closeR process sending a quit message, msgQ:

$$\mathsf{exec}\ \mathsf{closeR} \multimap \{\mathsf{msgQ}\}$$
$$\mathsf{msgQ} \bullet \mathsf{exec}\,(\mathsf{waitL}; Q) \multimap \{\mathsf{exec}\,Q\}$$

First-order universal and existential quantification. The first-order quantifiers type processes that exchange functional values. A process offering service $\forall x{:}\tau.A_x$ (or using service $\exists x{:}\tau.A_x$) first inputs a value x of functional type τ and then offers (resp., uses) service A_x. Dually, a process offering service $\exists x{:}\tau.A_x$ (or using service $\forall x{:}\tau.A_x$)

asynchronously outputs the value of some functional term M of type τ and then offers (resp., uses) service $[M/x]A_x$. The first-order quantifiers are thus dependent session types; in the non-dependent case, we write the types as $\tau \supset A$ and $A \wedge \tau$.

Since value inputs and outputs are not critical to the remainder of this proposal, the reader who is interested in further details of their static and dynamic semantics in SILL should refer to the papers by Toninho et al. (2013, 2011); we leave the extrapolation to singleton linear logic as an exercise for the reader.

Multiplicative conjunction and linear implication. The restriction to sequents with singleton antecedents proves fatal to attempts to include multiplicative conjunction ($A \otimes B$) and linear implication ($A \multimap B$) as connectives in singleton linear logic. For multiplicative conjunction, the left rule is problematic because it breaks down one hypothesis into two; for linear implication, the right rule is problematic because it introduces a new hypothesis even if one is already there. Fortunately, for the examples in which we are interested, the absence of $\otimes$ and $\multimap$ is not an issue.

4.10 Concurrency

Having described the typing rules and SSOS rules for process chains, we need to say a word about concurrency. The SSOS rules correspond to cut reductions. Typically, from a proof-theoretic standpoint, we would only be concerned with taking cut reductions sequentially, starting with those reductions nearest the derivation's leaves. This way, we need only deal with eliminating cuts whose subproofs are cut-free.

However, another strategy would be to allow cut reductions only at the top level — allowing cut reductions to occur only underneath other cuts. This strategy, used by Caires et al. (2013), corresponds to that of the π-calculus, in that reductions occur inside parallel compositions but not underneath input or output prefixes. In general, several cut reductions may be applicable; if the different interleavings of these reductions are treated as indistinguishable, then the reductions appear to happen concurrently.

We do not separately prove a correspondence between the cut reductions of singleton SILL and a calculus of process chains, mainly because we know of no independently existing process calculus that deals exclusively with chains. However, the reader may refer to Caires et al.'s work (2013) for a correspondence between the cut reductions of intuitionistic linear logic and the process reductions of the π-calculus. Because singleton SILL is a fragment of SILL, their result carries over to singleton SILL and π-calculus chains.

5 From ordered logical specifications to processes

The goal of this document is to relate the two notions of concurrency that arise in ordered logical specifications and session-typed process chains, respectively. Thus far, we have reviewed propositional ordered logical specifications (Section 2), identified a class of message-passing choreographies (Section 3), and presented a well-defined restriction of linear logic that serves as a session-typing discipline for process chains (Section 4). We are finally ready to tackle this document's main goal.

First, in Section 5.1, we show how to translate choreographies to process chains. Section 5.2 demonstrates correctness of the translation, in the sense that the translation is a weak bisimulation. Finally, in Section 5.3, we derive a typing discipline for choreographies and show that well-typed choreographies become well-typed processes under translation.

5.1 Translation of choreographies to process chains

Recall the grammar of *local* propositions given in Section 3.2.1:

$$\begin{array}{ll} \text{Negative propositions} & A^-, B^- ::= \{A^+\} \mid \&_{i\in I}(\underset{\leftarrow}{m}_i^+ \twoheadrightarrow \{A_i^+\}) \mid \&_{i\in I}(\underset{\rightarrow}{m}_i^+ \rightarrowtail \{A_i^+\}) \\ \text{Positive propositions} & A^+, B^+ ::= A^+ \bullet B^+ \mid 1 \mid \underset{\leftarrow}{m}^+ \bullet A^+ \mid A^+ \bullet \underset{\rightarrow}{m}^+ \mid p^+ \mid A^- \end{array}$$

The translation $[\![-]\!]$ from choreographies to process chains is syntax-directed, according to the proposition's polarity. The principle that guides the design of this translation is that each choreography transition from a context Ω should be matched by a process-chain transition from the chain to which Ω translates, and vice versa.

Translating negative propositions. The translation for negative propositions is inductively defined by:

$$\begin{aligned} [\![\{A^+\}]\!] &= [\![A^+]\!] \\ [\![\&_{i\in I}(\underset{\leftarrow}{m}_i \twoheadrightarrow \{A_i^+\})]\!] &= \mathsf{caseR}_{i\in I}\,(\underset{\leftarrow}{m}_i \Rightarrow [\![A_i^+]\!]) \\ [\![\&_{i\in I}(\underset{\rightarrow}{m}_i \rightarrowtail \{A_i^+\})]\!] &= \mathsf{caseL}_{i\in I}\,(\underset{\rightarrow}{m}_i \Rightarrow [\![A_i^+]\!]) \end{aligned}$$

The lax modality $\{A^+\}$ translates just as A^+ does: the modality is silent. The proposition $\&_{i\in I}(\underset{\leftarrow}{m}_i \twoheadrightarrow \{A_i^+\})$ is a caseR process that waits to receive one of the labels $\underset{\leftarrow}{m}_i$ from the right side; the arms of the case are exactly the translations of each A_i^+. Likewise, $\&_{i\in I}(\underset{\rightarrow}{m}_i \rightarrowtail \{A_i^+\})$ is a caseL process.

Intuitively, this translation is the correct one because the caseR receives a message $\underset{\leftarrow}{m}_k$ and continues as $[\![A_k^+]\!]$ exactly when the proposition $\&_{i\in I}(\underset{\leftarrow}{m}_i \twoheadrightarrow \{A_i^+\})$ transitions to A_k^+. A similar intuition applies to the caseL process.

Translating positive propositions. Positive propositions, which appear only in monadic heads in this grammar, are also translated to processes.

$$\begin{aligned} [\![\underset{\leftarrow}{m} \bullet A^+]\!] &= \mathsf{selectL}\ \underset{\leftarrow}{m}; [\![A^+]\!] \\ [\![A^+ \bullet \underset{\rightarrow}{m}]\!] &= \mathsf{selectR}\ \underset{\rightarrow}{m}; [\![A^+]\!] \\ [\![p^+]\!] &= \mathsf{call}\ p^+ \\ [\![A_1^+ \bullet A_2^+]\!] &= \mathsf{spawn}\ [\![A_1^+]\!]; [\![A_2^+]\!] \\ [\![1]\!] &= \leftrightarrow \end{aligned}$$

The message conjunctions $\underset{\leftarrow}{m} \bullet A^+$ and $A^+ \bullet \underset{\rightarrow}{m}$ translate to left and right selection processes $\mathsf{selectL}\ \underset{\leftarrow}{m}; [\![A^+]\!]$ and $\mathsf{selectR}\ \underset{\rightarrow}{m}; [\![A^+]\!]$. Process-like atoms translate to calls to the corresponding process variable. A general ordered conjunction $A_1^+ \bullet A_2^+$ translates as

the composition of the processes to which A_1^+ and A_2^+ translate. Lastly, the proposition 1 translates to the process $\leftrightarrow$, which forwards between its left and right endpoints.[14]

The idea is that each inversion step is matched by an asynchronous transition from the process, such as the outermost inversion step for $\underleftarrow{m} \bullet A^+$ into $\underleftarrow{m}, A^+$ being matched by the asynchronous decomposition of $\mathsf{selectL}\ \underleftarrow{m}; [\![A^+]\!]$ into message $\underleftarrow{m}$ and executing process $[\![A^+]\!]$.

Translating contexts. Finally, ordered contexts, Ω, translate to chains of executing processes, which are represented as ordered contexts in the SSOS. The translation proceeds homomorphically over contexts, with each hypothesis becoming an SSOS hypothesis in the translation.

$$
\begin{aligned}
[\![\Omega_1, \Omega_2]\!]_c &= [\![\Omega_1]\!]_c, [\![\Omega_2]\!]_c \\
[\![\cdot]\!]_c &= \cdot \\
[\![A^+]\!]_c &= \mathsf{exec}\ [\![A^+]\!] \\
[\![A^-]\!]_c &= \mathsf{exec}\ [\![A^-]\!] \\
[\![\underrightarrow{m}]\!]_c &= \mathsf{msgR}\ \underrightarrow{m} \\
[\![\underleftarrow{m}]\!]_c &= \mathsf{msgL}\ \underleftarrow{m} \\
[\![p^+]\!]_c &= \mathsf{exec}\ [\![A^-]\!] \quad \text{where } p^+ \multimap A^- \in \Sigma
\end{aligned}
$$

The left- and right-directed message-like atoms $\underleftarrow{m}$ and $\underrightarrow{m}$ become the left- and right-directed messages $\mathsf{msgL}\ \underleftarrow{m}$ and $\mathsf{msgR}\ \underrightarrow{m}$, respectively, in the SSOS. The process-like atom p^+ becomes $\mathsf{exec}\ P$, where P is the process body of p^+'s unique definition in the choreography Σ. Once again, the translation is defined in such a way that the behavior of contexts and their SSOS translations match.

5.2 Correctness of the translation

As stressed in the preceding discussion, the principle that guided the translation's design was that the choreography's transitions should be matched by those of the process to which that choreography translates. In other words, the translation, $[\![-]\!]$, should be a strong bisimulation.

Unfortunately, that property does not quite hold: $[\![-]\!]$ is not even a strong simulation. Consider the choreography transition $\{A^+\} \longrightarrow A^+$, for example. For $[\![-]\!]$ to be a strong (bi)simulation, there would have to be an SSOS transition $\mathsf{exec}\ P \longrightarrow \mathsf{exec}\ P$, where $P = [\![A^+]\!]$. Such a transition simply does not exist in the SSOS of our processes.

However, if we treat that choreography transition (and, more generally, $\Theta\{\{A^+\}\} \longrightarrow \Theta\{A^+\}$) as silent, then it is true that $[\![-]\!]$ is a *weak simulation*: together $\{A^+\} \xrightarrow{\tau} A^+$ and $P = [\![A^+]\!]$ indeed (trivially) imply $\mathsf{exec}\ P \xrightarrow{\tau}{}^* \mathsf{exec}\ P$. More generally:

Theorem 1 (Completeness).

- *If* $\Omega \longrightarrow \Omega'$, *then* $[\![\Omega]\!]_c \longrightarrow [\![\Omega']\!]_c$.

[14] Interestingly, for this translation, the identity principle ($\leftrightarrow$ process) acts like a nullary cut principle (spawn $P_1; P_2$ process).

- *If* $\Omega \xrightarrow{\tau} \Omega'$, *then* $[\![\Omega]\!]_c \xrightarrow{\tau}{}^* [\![\Omega']\!]_c$.

Proof. By analyzing the structure of the given transition and translation. Notice that the given silent transition is necessarily $\Theta\{\{A^+\}\} \xrightarrow{\tau} \Theta\{A^+\}$ because no other transitions are defined to be silent. □

Neither is the inverse of $[\![-]\!]$ a strong simulation. Consider the SSOS transition $\mathsf{exec}\,(\mathsf{call}\,p^+) \longrightarrow \mathsf{exec}\,P$, where the body of the process definition of p^+ is P. By examining the definition of $[\![-]\!]$, we see that there would have to be a choreography transition $p^+ \longrightarrow p^+$ if the inverse of $[\![-]\!]$ is to be a strong simulation. Once again, such a transition simply does not exist.

However, if we treat that SSOS transition (and, more generally, $\Delta_1, \mathsf{exec}\,(\mathsf{call}\,p^+), \Delta_2 \longrightarrow \Delta_1, \mathsf{exec}\,P, \Delta_2$) as silent, then it is true that the inverse of $[\![-]\!]$ is a *weak simulation*: together $\mathsf{exec}\,(\mathsf{call}\,p^+) \xrightarrow{\tau} \mathsf{exec}\,P$ and $[\![p^+]\!]_c = \mathsf{exec}\,P$ indeed (trivially) imply $p^+ \xrightarrow{\tau}{}^* p^+$. More generally:

Theorem 2 (Soundness).

- *If* $[\![\Omega]\!]_c \longrightarrow \Delta'$, *then* $\Omega \longrightarrow \Omega'$ *for some* Ω' *such that* $\Delta' = [\![\Omega']\!]_c$.
- *If* $[\![\Omega_1]\!]_c, \mathsf{exec}\,(\mathsf{call}\,p^+), [\![\Omega_2]\!]_c \xrightarrow{\tau} [\![\Omega_1]\!]_c, \mathsf{exec}\,P, [\![\Omega_2]\!]_c$, *then* $p^+ \multimap A^- \in \Sigma$ *for some* A^- *such that* $[\![A^-]\!] = P$.

Proof. By analyzing the structure of the given transition and translation. □

Putting these two pieces together, $[\![-]\!]$ is a weak bisimulation and $[\![-]\!]$ is therefore correctly defined.

5.3 Well-typed choreographies translate to well-typed processes

At this point, we can translate choreographies to syntactically well-formed processes. However, not all choreographies translate to *well-typed* processes. For example, $(A_1^+ \bullet \underrightarrow{m}_1) \bullet (\underleftarrow{m}_2 \bullet A_2^+)$ translates to the process spawn $(\mathsf{selectR}\,\underrightarrow{m}_1; [\![A_1^+]\!]); (\mathsf{selectL}\,\underleftarrow{m}_2; [\![A_2^+]\!])$, which is not typable in singleton SILL because the messages have no matching recipients and therefore collide.

To identify a class of choreographies that enjoy the same session fidelity and, especially, deadlock freedom (i.e., progress) properties as well-typed processes, we need to introduce a typing discipline for choreographies, $\vdash^{\mathrm{ch}}$, that is preserved by the translation. The following theorem is our goal:

Theorem (Translation preserves typing).

- $\Delta \vdash^{\mathrm{ch}} A^- :: B$ *if and only if* $\Delta \vdash [\![A^-]\!] :: B$.
- $\Delta \vdash^{\mathrm{ch}} A^+ :: B$ *if and only if* $\Delta \vdash [\![A^+]\!] :: B$.

Because $[\![-]\!]$ describes a fairly tight correspondence between choreographies and processes, the rules for $\vdash^{\mathrm{ch}}$ are "just" the rules for typing processes ($\vdash$): simply replace the process terms with the corresponding ordered logic proposition. For instance,

$\underleftarrow{m}_k \bullet A^+$ translates to $\mathsf{selectL}\ \underleftarrow{m}_k; [\![A^+]\!]$ and so, from the $\&\mathrm{L}$ rule that types $\mathsf{selectL}$ processes, we obtain the rule for typing $\underleftarrow{m}_k \bullet A^+$:

$$\frac{A_k \vdash P :: C \quad (k \in L)}{\&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{selectL}\ k; P :: C}\ \&\mathrm{L} \qquad \leftrightsquigarrow \qquad \frac{A_k \vdash^{\mathrm{ch}} A^+ :: C \quad (k \in L)}{\&\{\ell : A_\ell\}_{\ell \in L} \vdash^{\mathrm{ch}} \underleftarrow{m}_k \bullet A^+ :: C}\ \&\mathrm{L}$$

Notice how these new rules use two varieties of proposition in two different ways: polarized ordered propositions are choreographies; unpolarized (singleton) linear propositions serve as their types. For this reason, the rules in isolation are admittedly a bit mind-bending, but they are in fact rather straightforward to derive if we keep the replacement pattern in mind. As another example of the pattern, 1 translates to $\leftrightarrow$, the forwarding process, and so, from the ID_A rule that types $\leftrightarrow$ processes, we obtain the rule for typing 1 as a choreography:

$$\frac{}{A \vdash \leftrightarrow :: A}\ \mathrm{ID}_A \qquad \leftrightsquigarrow \qquad \frac{}{A \vdash^{\mathrm{ch}} 1 :: A}\ \mathrm{ID}_A$$

Of the rules for typing choreographies, only two do not mimic process-typing rules. First is the rule for typing $\{A^+\}$. But, here again, the goal theorem shows the way; the $\{-\}$ modality is silent in the translation, so it should also be silent in the typing:

$$\frac{\Delta \vdash^{\mathrm{ch}} A^+ :: A}{\Delta \vdash^{\mathrm{ch}} \{A^+\} :: A}$$

Second is the rule for typing process-like atoms p. Similarly to process variables, we assume that the programmer has specified the type of each process-like atom. With this signature, we just check that each use of a process-like atom adheres to its specified type:

$$\frac{p^+ : \{\Delta \vdash^{\mathrm{ch}} A\} \in \Sigma}{\Delta \vdash^{\mathrm{ch}} p^+ :: A}$$

The complete set of rules for typing choreographies is shown in Fig. 10. It is easy to check that the goal indeed holds: the translation preserves typing.

Theorem 3 (Translation preserves typing).

- $\Delta \vdash^{\mathrm{ch}} A^- :: B$ *if and only if* $\Delta \vdash [\![A^-]\!] :: B$.
- $\Delta \vdash^{\mathrm{ch}} A^+ :: B$ *if and only if* $\Delta \vdash [\![A^+]\!] :: B$.

Proof. By induction on the structure of the given typing derivation. □

Instead of using this theorem to justify the rules for typing choreographies, it's also possible to justify them from first principles if desired. Consider the CUT_A rule, for example:

$$\frac{\Delta \vdash^{\mathrm{ch}} A_1^+ :: A \quad A \vdash^{\mathrm{ch}} A_2^+ :: C}{\Delta \vdash^{\mathrm{ch}} A_1^+ \bullet A_2^+ :: C}\ \mathrm{CUT}_A$$

This rule says that if the choreography A_1^+ has, from left to right, the interfaces Δ and A (i.e., $\Delta \vdash^{\mathrm{ch}} A_1^+ :: A$) and the choreography A_2^+ has the interfaces A and C

$$\frac{\forall \ell \in L:\ \ \Delta \vdash^{\text{ch}} A^+_\ell :: A_\ell}{\Delta \vdash^{\text{ch}} \&_{\ell \in L}(\underleftarrow{m}_\ell \twoheadrightarrow \{A^+_\ell\}) :: \&\{\ell : A_\ell\}_{\ell \in L}}\ \&\text{R} \qquad \frac{A_k \vdash^{\text{ch}} A^+ :: C \quad (k \in L)}{\&\{\ell : A_\ell\}_{\ell \in L} \vdash^{\text{ch}} \underleftarrow{m}_k \bullet A^+ :: C}\ \&\text{L}$$

$$\frac{\Delta \vdash^{\text{ch}} A^+ :: A_k \quad (k \in L)}{\Delta \vdash^{\text{ch}} A^+ \bullet \underrightarrow{m}_k :: \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus\text{R} \qquad \frac{\forall \ell \in L:\ \ A_\ell \vdash^{\text{ch}} A^+_\ell :: C}{\oplus\{\ell : A_\ell\}_{\ell \in L} \vdash^{\text{ch}} \&_{\ell \in L}(\underrightarrow{m}_\ell \rightarrowtail \{A^+_\ell\}) :: C}\ \oplus\text{L}$$

$$\frac{\Delta \vdash^{\text{ch}} A^+ :: A}{\Delta \vdash^{\text{ch}} \{A^+\} :: A} \qquad \frac{p^+ : \{\Delta \vdash^{\text{ch}} A\} \in \Sigma}{\Delta \vdash^{\text{ch}} p^+ :: A}$$

$$\frac{\Delta \vdash^{\text{ch}} A^+_1 :: A \quad A \vdash^{\text{ch}} A^+_2 :: C}{\Delta \vdash^{\text{ch}} A^+_1 \bullet A^+_2 :: C}\ \text{CUT}_A \qquad \frac{}{A \vdash^{\text{ch}} 1 :: A}\ \text{ID}_A$$

Figure 10: Choreography typing rules. The context Δ is either empty or a singleton.

(i.e., $A \vdash^{\text{ch}} A^+_2 :: C$), then their composition, $A^+_1 \bullet A^+_2$, has: *i)* Δ as its left-hand interface, which is reasonable because the left-hand side of A^+_1 is exposed; and *ii)* C as its right-hand interface, which is also reasonable because the right-hand side of A^+_2 is exposed. The other rules can be given similar justifications.

6 Proposed work

In this document, we have shown how the session types that arise from singleton linear logic form a bridge between a class of ordered logical specifications and well-typed processes—between proof-construction-as-computation and proof-reduction-as-computation. Most of the proposed work involves generalizing this connection along several dimensions: *i*) a more expressive logic for specifications; *ii*) a more expansive translation that covers generative invariants; and *iii*) a more permissive session-type system. We now outline that proposed work.

6.1 From ordered logical to linear logical specifications

The primary area of proposed work is to generalize the logic used for specifications from ordered logic to the more expressive linear logic. The process chains used in this proposal will be correspondingly generalized to Caires et al.'s (2013) SILL process trees. We'll motivate this generalization with an example: addition of binary representations.

Logical specification. By adapting ideas from Turing machines, it is possible—though undoubtedly awkward—to give an ordered logical specification for adding two binary numbers. First, the numbers are arranged end-to-end, separated by a plus atom and terminated by an equals atom. For instance, the string

$$\mathsf{eps} \bullet \mathsf{bit1} \bullet \mathsf{bit0} \bullet \mathsf{plus} \bullet \mathsf{bit1} \bullet \mathsf{bit0} \bullet \mathsf{equals}$$

$$\text{equals} \multimap \{\text{dec} \bullet \text{equals}'\}$$

$$\begin{array}{ll}
\text{bit0} \bullet \text{dec} \multimap \{\text{dec} \bullet \text{bit0}'\} & \text{bit0} \bullet \text{skip} \multimap \{\text{skip} \bullet \text{bit0}\} \\
\text{bit1} \bullet \text{dec} \multimap \{\text{skip} \bullet \text{bit0} \bullet \text{ok}\} & \text{bit1} \bullet \text{skip} \multimap \{\text{skip} \bullet \text{bit1}\} \\
\text{plus} \bullet \text{dec} \multimap \{\text{fail}\} & \text{plus} \bullet \text{skip} \multimap \{\text{inc} \bullet \text{plus}\} \\
\text{ok} \bullet \text{bit0}' \multimap \{\text{bit1} \bullet \text{ok}\} & \text{fail} \bullet \text{bit0}' \multimap \{\text{fail}\} \\
\text{ok} \bullet \text{equals}' \multimap \{\text{equals}\} & \text{fail} \bullet \text{equals}' \multimap \{1\}
\end{array}$$

Figure 11: An ordered logical specification of Turing-machine–like binary addition

represents a request to evaluate 2 + 2. Next, repeatedly decrement the second number and increment the first number. When the second number reaches 0, the first number holds the desired sum. The ordered logical specification of this addition algorithm is shown in Fig. 11.

Unfortunately, this algorithm is not especially efficient: it takes $\Omega(N \log N)$ work to compute $M + N$. It would be better to add the two binary representations bit-by-bit using the usual grade-school algorithm. However, bit-by-bit addition demands that we can *locally* access the least significant bit of each number and, separately, produce output bits—which is not possible in an ordered logical specification.

It is possible, however, in a *destination-passing* linear logical specification (Cervesato et al. 2002). Even without the ordering constraint, a tree structure can be recovered via destinations that thread the bit atoms together with a plus parent atom. Pictorially, the request to compute 2 + 2 would be expressed as the state

$$\begin{array}{r}
\text{eps}(c_2) \otimes \text{bit1}(c_2, c_1) \otimes \text{bit0}(c_1, c_0) \\
\text{eps}(d_2) \otimes \text{bit1}(d_2, d_1) \otimes \text{bit0}(d_1, d_0)
\end{array} \otimes \text{plus}(c_0, d_0, c)$$

where c and the c_is and d_js are all destinations and where the sum will be output at destination c. Thus, the destination-passing rule for adding two numbers that both end in bit0 is

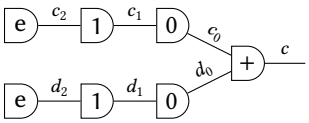
$$\text{bit0}(C_1, C_0) \otimes \text{bit0}(D_1, D_0) \otimes \text{plus}(C_0, D_0, C) \multimap \{\exists c_0'.\ \text{plus}(C_1, D_1, c_0') \otimes \text{bit0}(c_0', C)\}\,.$$

It says that if both inputs end in bit0, then their sum also ends in bit0, with the more significant bits obtained by inductively adding the more significant bits of the two inputs. When this rule is applied to the above state, the state changes and the first bit of output is produced:

$$\begin{array}{r}
\text{eps}(c_2) \otimes \text{bit1}(c_2, c_1) \\
\text{eps}(d_2) \otimes \text{bit1}(d_2, d_1)
\end{array} \otimes \text{plus}(c_1, d_1, c_0') \otimes \text{bit0}(c_0', c)\,.$$

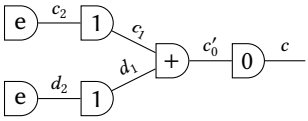
Concurrent processes. Now, let's consider how we might add two binary numbers using SILL process trees. Suppose that we represent each number as before—each number is a chain of bit processes (a degenerate process subtree if you will)—and that

we include a plus parent process that uses the two numbers to offer the sum. For example, the following process network represents a request to compute 2 + 2.[15]



where the c_is and d_js are channels. Notice the remarkable similarity of this network with the initial linear logical state shown above: destinations become channels and atoms become processes.

We would expect the plus process to be implemented in such a way that the above process network eventually transforms to the following network.



Once again, there is a remarkable similarity between this network and the linear logical state after producing one output bit.

The proposed work is to make these similarities precise. Just as in this document, the overall goal will be to identify a class of linear logical specifications that can be translated to SILL process trees. This item of proposed work is of primary importance.

Specific goals. This proposed work involves several components.

- *Identify the class of linear logical specifications that act as choreographies.* Not all linear logical specifications will describe process-like behaviors. As in the ordered case, choreographies will need to be both local and specification-preserving. Now, however, locality depends not on adjacency in the ordered context, but on sharing a destination.

  The key challenge here, therefore, will be to ensure that destinations are used in a channel-like way within the choreography. Each atom should "offer" along one destination and "use" possibly several distinct destinations, and each destination should have one occurrence as an "offer" and one occurrence as a "use". The machinery of destination uniqueness and index sets (Simmons 2012) will likely be useful here.

- *Develop a translation of choreographies to SILL processes.* In addition to translating destinations to channels, the main challenges here will be expanding the class of choreographies to allow translation to processes of $\otimes$, $\multimap$, and ! type. The $\otimes$ and $\multimap$ types are not possible in singleton linear logic (as mentioned in Section 4.9) and so nothing similar was considered in the translation of ordered choreographies to process chains. The ! type was, by choice, not considered in the ordered translation to keep the initial development simple.

[15] The process names have been abbreviated to e, 0, 1, and + for this picture.

- *Give a type system for choreographies.* I expect to follow the same pattern as in the ordered case: derive the choreography typing rules from the process typing rules by looking at the process to which a choreography translates. While everything will be notationally more complex, I do not expect many surprises here.
- *Relate the results for ordered logic to those for linear logic.* Simmons and Pfenning (2011a) show how to encode ordered logical specifications in linear logic using destinations. For instance, under their destination-adding translation, the clause for incrementing bit1 from our $\underleftarrow{\mathsf{inc}}$-choreography becomes the following linear logical clause.

$$\mathsf{bit1} \bullet \underleftarrow{\mathsf{inc}} \multimap \{\underleftarrow{\mathsf{inc}} \bullet \mathsf{bit0}\} \qquad \leftrightsquigarrow \qquad \mathsf{bit1}(C_1, C_0) \otimes \underleftarrow{\mathsf{inc}}(C_0, C) \multimap \{\exists c_0'.\ \underleftarrow{\mathsf{inc}}(C_1, c_0') \otimes \mathsf{bit0}(c_0', C)\}$$

  There should be a similar "channel-adding" translation from singleton SILL process chains to SILL processes.

$$\mathsf{bit1} = \mathsf{caseR}\,(\underleftarrow{\mathsf{inc}} \Rightarrow \mathsf{selectL}\,\underleftarrow{\mathsf{inc}}; \mathsf{bit0}) \qquad \leftrightsquigarrow \qquad c \leftarrow \mathsf{bit1} \leftarrow d = \mathsf{case}\ c\ \mathsf{of}\ \underleftarrow{\mathsf{inc}} \Rightarrow \mathsf{select}\ d\ \underleftarrow{\mathsf{inc}};\ c \leftarrow \mathsf{bit0} \leftarrow d$$

  Moreover, the translation from choreography to process should respect the destination-adding translation: adding destinations to an ordered choreography and then translating it to a process should give the same result as first translating the choreography to a process chain and then adding channels. This will serve as a sanity check on our design of the translation from linear choreographies to SILL processes.

6.2 Generative invariants as session types

In this proposal, we have used the non-modal fragment of ordered logic to specify concurrent systems, whereas that fragment of ordered logic was originally developed by Lambek (1958) to describe sentence structure. However, these two modes of use of ordered logic are not as different as they might first appear.

Recall that, in our running example of an incrementable binary counter, the counter is represented as a string of bit0, bit1, and inc atoms terminated at the most significant end by an eps. More precisely, a string is a well-formed binary counter if it can be generated from the *Cntr* nonterminal by the context-free grammar

$$\mathit{Cntr} ::= \mathsf{eps} \mid \mathit{Cntr} \bullet \mathsf{bit0} \mid \mathit{Cntr} \bullet \mathsf{bit1} \mid \mathit{Cntr} \bullet \mathsf{inc}\,,$$

which is notation for four distinct productions.

Building on Lambek's work, the same context-free grammar can be described in ordered logic using *generative invariants* (Simmons 2012). Each production in the grammar (below, left) becomes a clause (below, right), with the atomic proposition

Cntr acting as the nonterminal:

$$\begin{array}{l|l} \mathit{Cntr} \to \mathsf{eps} & \mathsf{Cntr} \multimap \{\mathsf{eps}\} \\ \mathit{Cntr} \to \mathit{Cntr} \bullet \mathsf{bit0} & \mathsf{Cntr} \multimap \{\mathsf{Cntr} \bullet \mathsf{bit0}\} \\ \mathit{Cntr} \to \mathit{Cntr} \bullet \mathsf{bit1} & \mathsf{Cntr} \multimap \{\mathsf{Cntr} \bullet \mathsf{bit1}\} \\ \mathit{Cntr} \to \mathit{Cntr} \bullet \mathsf{inc} & \mathsf{Cntr} \multimap \{\mathsf{Cntr} \bullet \mathsf{inc}\}\,. \end{array}$$

Just as all binary counters are generated from the *Cntr* nonterminal according to the above productions, so too are all binary counters generated as maximal rewritings of the Cntr atom according to these clauses:

Definition 4 (Counter well-formedness). String S is a well-formed counter if S is a maximal rewriting of Cntr under the signature Σ_{Cntr}, that is, if $\mathsf{Cntr} \longrightarrow^{+}_{\Sigma_{\mathsf{Cntr}}} S \not\longrightarrow_{\Sigma_{\mathsf{Cntr}}}$.

For example, the maximal trace

$$\underline{\mathsf{Cntr}} \longrightarrow_{\Sigma_{\mathsf{Cntr}}} \underline{\mathsf{Cntr}} \bullet \mathsf{inc} \longrightarrow_{\Sigma_{\mathsf{Cntr}}} \underline{\mathsf{Cntr}} \bullet \mathsf{bit1} \bullet \mathsf{inc} \longrightarrow_{\Sigma_{\mathsf{Cntr}}} \mathsf{eps} \bullet \mathsf{bit1} \bullet \mathsf{inc} \not\longrightarrow_{\Sigma_{\mathsf{Cntr}}}$$

witnesses that eps • bit1 • inc is a well-formed binary counter.

As observed by Simmons (2012), generative invariants like Σ_{Cntr} serve a similar purpose for ordered logical specifications as types do for functional programs: both describe the valid states and enable preservation and progress properties for their respective notions of computation.

Given the translation from choreographies (i.e., ordered logical specifications) to well-typed processes that was presented in Section 5, it's thus natural to ask how that translation interacts with a generative invariant. Being the choreography's "type", does the generative invariant become the process's session type?

It appears that the answer is likely yes. Compare, for example, the generative invariant for the $\underleftarrow{\text{inc}}$-choreography with the types of the `eps`, `bit0`, and `bit1` processes and the recursive type `Cntr` from Section 4.5:

$$\begin{array}{l|l} \mathsf{Cntr} \multimap \{\mathsf{eps}\} & \texttt{eps : \{ |- Cntr \}} \\ \mathsf{Cntr} \multimap \{\mathsf{Cntr} \bullet \mathsf{bit0}\} & \texttt{bit0 : \{ Cntr |- Cntr \}} \\ \mathsf{Cntr} \multimap \{\mathsf{Cntr} \bullet \mathsf{bit1}\} & \texttt{bit1 : \{ Cntr |- Cntr \}} \\ \mathsf{Cntr} \multimap \{\mathsf{Cntr} \bullet \underleftarrow{\mathsf{inc}}\} & \texttt{stype Cntr = \&\{ inc: Cntr \}.} \end{array}$$

Similar correspondences between generative invariants and session types exist for the $\underleftarrow{\text{dec}}$-choreography, the $\underleftrightarrow{\text{bit}}$-choreography, and all other examples that we've considered. It seems much too tantalizing to be pure coincidence.

Therefore, if all else goes smoothly, I propose to develop a translation of generative invariants to session types and prove that it is respected by the translation from choreographies to processes. I plan to follow the pattern of this thesis proposal, first developing the translation for the special case of ordered generative invariants before extending the results to linear generative invariants. This item of proposed work is of somewhat lesser importance than the generalization from ordered logic to linear logic for specifications, but has appeal in giving a more compelling explanation of the choreography types presented in Section 5.3.

6.3 Translating untyped choreographies to untyped processes

In this proposal, we have been concerned only with *well-typed* processes and a corresponding class of well-typed choreographies. The logically grounded session-type discipline ensures that well-typed processes (and, consequently, well-typed choreographies) enjoy communication safety, session fidelity, and deadlock freedom (i.e., global progress). However, by demanding such a strong form of progress, the current session-type discipline forbids *all* racy processes, even if the races are benign or non-critical.

For example, consider the following process:

$$\mathsf{caseL}\,(\mathsf{okL} \Rightarrow \mathsf{caseR}\,(\mathsf{okR} \Rightarrow P)) + \mathsf{caseR}\,(\mathsf{okR} \Rightarrow \mathsf{caseL}\,(\mathsf{okL} \Rightarrow P))\,,$$

which waits to receive—in either order—okL and okR labels from both its left- and right-hand neighbors, respectively. (The process constructor + denotes nondeterministic choice.) This process is certainly racy because it's impossible, in general, to predict the order in which the okL and okR labels will arrive. But, even so, this race is benign: execution continues with the process P once, and only once, both labels arrive in either order.

Choreographies may serve as a stepping-stone toward a more permissive, yet still logically grounded, session-type discipline that allows this and other benign races. The above example can be cast as the choreography

$$(\underrightarrow{\mathsf{okL}} \rightarrowtail \{\underleftarrow{\mathsf{okR}} \twoheadrightarrow \{A^+\}\}) \mathbin{\&} (\underleftarrow{\mathsf{okR}} \twoheadrightarrow \{\underrightarrow{\mathsf{okL}} \rightarrowtail \{A^+\}\})\,.$$

By considering how the proposed translation from generative invariants to session types might apply to a generative invariant for this choreography, we may gain insight into a session-type discipline that allows benign races. I also propose to develop a translation of a broader class of choreographies to untyped processes, which may provide different insight than just looking at the existing session-type discipline.

6.4 Session-typed Turing machines

Finally, as the example in Section 6.1 shows, some Turing machines can be session-typed: by translating the ordered logical specification from Fig. 11, we get a well-typed, Turing-machine–like process for adding two binary representations. In particular, the chain structure of singleton linear logic suggests a fit with the one-way infinite tapes of Turing machines.

Although not directly related to my proposed thesis statement, if time permits, I would like to explore further the possible connections between singleton linear logic and Turing machines. This is the most open-ended item of proposed work and the least related to my proposed thesis statement, but, if successful, may have some interest to researchers outside the programming languages community, e.g., those working in the theory of computation.

A Turing-machine–like addition process

In this appendix, we present the code for a well-typed, Turing-machine–like addition process. It corresponds to the ordered logical specification shown in Fig. 11.

```
stype Cntr = &{ inc: Cntr }

eps : { |- Cntr } =
{ caseR of
    inc => eps; bit1 }

bit0 : { Cntr |- Cntr } =
{ caseR of
    inc => bit1 }

bit1 : { Cntr |- Cntr } =
{ caseR of
    inc => selectL inc; bit0 }

stype Cntr_D = &{ dec: Cntr_D' , skip: Cntr_D }
  and Cntr_D' = +{ ok: Cntr_D , fail: Cntr }

bit0_d : { Cntr_D |- Cntr_D } =
{ caseR of
    dec => selectL dec; bit0_d'
  | skip => selectL skip; bit0_d }

bit1_d : { Cntr_D |- Cntr_D } =
{ caseR of
    dec => selectL skip; selectR ok; bit0_d
  | skip => selectL skip; bit1_d }

bit0_d' : { Cntr_D' |- Cntr_D' } =
{ caseL of
    ok => selectR ok; bit1_d
  | fail => selectR fail; <-> }

plus : { Cntr |- Cntr_D } =
{ caseR of
    dec => selectR fail; <->
  | skip => selectL inc; plus }

equals : { Cntr_D |- Cntr } =
{ selectL dec; equals' }

equals' : { Cntr_D' |- Cntr } =
```

```
{ caseL of
    ok => equals
  | fail => <-> }
```

References

Abramsky, Samson (1993). "Computational Interpretations of Linear Logic". In: *Theoretical Computer Science* 111.1–2: *Sixth Workshop on the Mathematical Foundations of Programming Semantics*. Ed. by Michael Mislove and R. D. Tennent, pp. 3–57. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90181-R (cit. on p. 3).

Andreoli, Jean-Marc (1992). "Logic Programming with Focusing Proofs in Linear Logic". In: *Journal of Logic and Computation* 2.3. Ed. by Dov M. Gabbay, pp. 297–347. ISSN: 0955-792X. DOI: 10.1093/logcom/2.3.297 (cit. on pp. 3, 7, 11).

Bellin, Gianluigi and Philip Scott (1994). "On the π-Calculus and Linear Logic". In: *Theoretical Computer Science* 135.1: *Eighth Workshop on the Mathematical Foundations of Programming Semantics*. Ed. by M.W. Mislove et al., pp. 11–65. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)00104-9 (cit. on p. 3).

Benton, P. Nick, Gavin M. Bierman, and Valeria C. V. de Paiva (1998). "Computational Types from a Logical Perspective". In: *Journal of Functional Programming* 8.2, pp. 177–193. DOI: 10.1017/S0956796898002998 (cit. on p. 3).

Book, Ronald V. and Friedrich Otto (1993). *String-Rewriting Systems*. Text and Monographs in Computer Science. Heidelberg and Berlin: Springer. DOI: 10.1007/978-1-4613-9771-7_3 (cit. on p. 6).

Caires, Luís and Frank Pfenning (2010). "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, Proceedings*. (Paris, Aug. 31–Sept. 3, 2010). Ed. by Paul Gastin and François Laroussinie. Vol. 6269. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, pp. 222–236. DOI: 10.1007/978-3-642-15375-4_16 (cit. on pp. 1, 3, 20).

Caires, Luís, Frank Pfenning, and Bernardo Toninho (2012). "Towards Concurrent Type Theory". In: *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation*. (Philadelphia, Jan. 28, 2012). Ed. by Benjamin C. Pierce. ACM Press, pp. 1–12. ISBN: 978-1-4503-1120-5. DOI: 10.1145/2103786.2103788 (cit. on p. 3).

— (2013). "Linear Logic Propositions as Session Types". In: *Mathematical Structures in Computer Science: Special Issue on Behavioral Types* (cit. on pp. 3, 5, 18, 20, 21, 29, 34).

Cervesato, Iliano and Andre Scedrov (2009). "Relating State-Based and Process-Based Concurrency through Linear Logic". In: *Information and Computation* 207.10: *Special Issue: 13th Workshop on Logic, Language, Information, and Computation (WoLLIC 2006)*. Ed. by Grigori Mints, Valéria de Paiva, and Ruy de Queiroz, pp. 1044–1077. ISSN: 0890-5401. DOI: 10.1016/j.ic.2008.11.006 (cit. on pp. 3, 14).

Cervesato, Iliano et al. (2002). *A Concurrent Logical Framework II: Examples and Applications*. Tech. rep. CMU-CS-2002-102. Revised May 2003. Pittsburgh: Department of Computer Science, Carnegie Mellon University (cit. on p. 35).

Cervesato, Iliano et al. (2012). "Trace Matching in a Concurrent Logical Framework". In: *7th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. (Copenhagen, Denmark, Sept. 9, 2012). Ed. by Adam Chlipala and Carsten Schürmann. New York: ACM Press, pp. 1–12. ISBN: 978-1-4503-1578-4. DOI: 10.1145/2364406.2364408 (cit. on pp. 4, 8, 13).

Cruz, Flávio et al. (2014). "A Linear Logic Programming Language for Concurrent Programming over Graph Structures". In: *Theory and Practice of Logic Programming* 14.4–5: *30th International Conference on Logic Programming Special Issue*. Ed. by Michael Leuschel and Tom Schrijvers, pp. 493–507. DOI: 10.1017/S1471068414000167 (cit. on p. 3).

Felleisen, Matthias (1985). *Transliterating Prolog into Scheme*. Tech. rep. 85-182. Bloomington: Computer Science Department, Indiana University (cit. on p. 6).

Girard, Jean-Yves (1987). "Linear Logic". In: *Theoretical Computer Science* 50.1. Ed. by Maurice Nivat and Matthew S. Paterson, pp. 1–102. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90045-4 (cit. on p. 3).

Griffith, Dennis and Frank Pfenning (2014). *OCaml Implementation of SILL*. Online. URL: https://github.com/ISANobody/sill (cit. on p. 3).

Hanus, Michael (2013). "Functional Logic Programming: From Theory to Curry". In: *Programming Logics: Essays in Memory of Harald Ganzinger*. Vol. 7797. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, pp. 123–168 (cit. on p. 6).

Honda, Kohei (1993). "Types for Dyadic Interaction". In: *4th International Conference on Concurrency Theory*. (Hildesheim, Germany, Aug. 23–26, 1993). Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, pp. 509–523. DOI: 10.1007/3-540-57208-2_35 (cit. on pp. 3, 20).

Honda, Kohei, Nobuko Yoshida, and Marco Carbone (2008). "Multiparty asynchronous session types". In: *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*. (San Francisco, California, Jan. 7–12, 2008). Ed. by George C. Necula and Philip Wadler. New York: ACM Press, pp. 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472 (cit. on p. 5).

Howard, William A. (1980). "The Formulae-as-Types Notion of Construction". In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Boston: Academic Press, pp. 479–490 (cit. on p. 3).

Jeffrey, Alan (2012). "LTL Types FRP: Linear-time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs". In: *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*. (Philadelphia, Pennsylvania, Jan. 24, 2012). Ed. by Koen Claessen and Nikhil Swamy. New York: ACM Press, pp. 49–60. ISBN: 978-1-4503-1125-0. DOI: 10.1145/2103776.2103783 (cit. on p. 3).

Lambek, Joachim (1958). "The Mathematics of Sentence Structure". In: *The American Mathematical Monthly* 65.3, pp. 154–170. ISSN: 0002-9890. DOI: 10.2307/2310058 (cit. on pp. 4, 37).

López, Pablo et al. (2005). "Monadic Concurrent Linear Logic Programming". In: *Principles and Practice of Declarative Programming*. Proceedings of the 7th International

ACM SIGPLAN Conference. (Lisbon, Portugal, July 11–13, 2005). Ed. by Pedro Barahona and Amy P. Felty. New York: ACM Press, pp. 35–46 (cit. on p. 3).

Martens, Chris et al. (2013). "Linear Logic Programming for Narrative Generation". In: *Logic Programming and Nonmonotonic Reasoning, 12th International Conference*. (Corunna, Spain, Sept. 15–19, 2013). Ed. by Pedro Cabalar and Tran Cao Son. Vol. 8148. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, pp. 427–432. DOI: 10.1007/978-3-642-40564-8_42 (cit. on p. 3).

Martin-Löf, Per (1980). "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI*. Amsterdam: North-Holland, pp. 153–175 (cit. on p. 3).

Miller, Dale (1992). "The π-Calculus as a Theory in Linear Logic: Preliminary Results". In: *Extensions of Logic Programming, Third International Workshop, ELP'92, Proceedings*. (Bologna, Italy, Feb. 26–28, 1992). Ed. by Evelina Lamma and Paola Mello. Vol. 660. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, pp. 242–264. DOI: 10.1007/3-540-56454-3_13 (cit. on p. 14).

Miller, Dale et al. (1991). "Uniform Proofs as a Foundation for Logic Programming". In: *Annals of Pure and Applied Logic* 51.1–2: *Selected papers from the Second Annual IEEE Symposium on Logic in Computer Science*. Ed. by Dexter Kozen, pp. 125–157. ISSN: 0168-0072. DOI: 10.1016/0168-0072(91)90068-W (cit. on p. 3).

Pfenning, Frank (2004). "Substructural Operational Semantics and Linear Destination-Passing Style (Invited Talk)". In: *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*. (Taipei, Taiwan, Nov. 4–6, 2004). Ed. by Wei-Ngan Chin. Vol. 3302. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, p. 196. ISBN: 3-540-23724-0. DOI: 10.1007/978-3-540-30477-7_13 (cit. on pp. 5, 22).

Pfenning, Frank and Robert J. Simmons (2009). "Substructural Operational Semantics as Ordered Logic Programming". In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009*. (Los Angeles, Aug. 11–14, 2009). Ed. by Andrew M. Pitts. IEEE Computer Society, pp. 101–110. ISBN: 978-0-7695-3746-7. DOI: 10.1109/LICS.2009.8 (cit. on pp. 6, 15, 17, 22).

Polakow, Jeff and Frank Pfenning (1999). "Relating Natural Deduction and Sequent Calculus for Intuitionistic Non-Commutative Linear Logic". In: *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*. (New Orleans, Louisiana, Apr. 1999). Ed. by Andre Scedrov and Achim Jung. Vol. 20. Electronic Notes in Theoretical Computer Science, pp. 449–466. DOI: 10.1016/S1571-0661(04)80088-4 (cit. on p. 4).

Schack-Nielsen, Anders (2011). "Implementing Substructural Logical Frameworks". PhD thesis. IT University of Copenhagen (cit. on p. 3).

Simmons, Robert J. (2012). "Substructural Logical Specifications". PhD thesis. Carnegie Mellon University (cit. on pp. 4, 8, 10, 36–38).

Simmons, Robert J. and Frank Pfenning (2011a). "Logical Approximation for Program Analysis". In: *Higher-Order and Symbolic Computation* 24.1–2, pp. 41–80. ISSN: 1388-3690. DOI: 10.1007/s10990-011-9071-2 (cit. on p. 37).

— (2011b). *Weak Focusing for Ordered Linear Logic*. Tech. rep. CMU-10-147. Carnegie Mellon University (cit. on pp. 10, 11).

Somogyi, Zoltan, Fergus Henderson, and Thomas Conway (1996). "The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language". In: *The Journal of Logic Programming* 29.1–3: *High-Performance Implementations of Logic Programming Systems*. Ed. by Gopal Gupta and Mats Carlsson, pp. 17–64. ISSN: 0743-1066. DOI: 10.1016/S0743-1066(96)00068-4 (cit. on p. 6).

Spivey, J. Michael and Silvija Seres (1999). "Embedding Prolog in Haskell". In: *Proceedings of the 1999 Haskell Workshop*. (Paris, France, Oct. 9, 1999). Ed. by Erik Meijer (cit. on p. 6).

Toninho, Bernardo, Luís Caires, and Frank Pfenning (2011). "Dependent Session Types via Intuitionistic Linear Type Theory". In: *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. (Odense, Denmark, July 20–22, 2011). Ed. by Peter Schneider-Kamp and Michael Hanus. New York: ACM Press, pp. 161–172. ISBN: 978-1-4503-0776-5. DOI: 10.1145/2003476.2003499 (cit. on p. 29).

— (2013). "Higher-Order Processes, Functions, and Sessions: A Monadic Integration". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013. Proceedings*. (Rome, Mar. 16–24, 2013). Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Heidelberg and Berlin: Springer, pp. 350–369. DOI: 10.1007/978-3-642-37036-6_20 (cit. on pp. 3, 25, 29).

— (2014). "Corecursion and Non-Divergence in Session-Typed Processes". In: *9th International Symposium on Trustworthy Global Computing*. (Rome, Italy, Sept. 5–6, 2014). Ed. by Matteo Maffei and Emilio Tuotso (cit. on p. 24).

Wadler, Philip (2014). "Propositions as Sessions". In: *Journal of Functional Programming* 24.2–3: *Special Issue Dedicated to ICFP 2012*, pp. 384–418. ISSN: 1469-7653. DOI: 10.1017/S095679681400001X (cit. on p. 20).

Watkins, Kevin et al. (2002). *A Concurrent Logical Framework I: Judgments and Properties*. Tech. rep. CMU-CS-2002-101. Revised May 2003. Pittsburgh: Department of Computer Science, Carnegie Mellon University (cit. on pp. 3, 4, 7, 8).