Asynchronous Multistructural Session Types

FRANK PFENNING, Carnegie Mellon University
KLAAS PRUIKSMA, Carnegie Mellon University

We present a system of multistructural session types that uniformly integrates layers with linear and non-linear types based on Benton's LNL. The underlying computational mechanism derives from traditional principal cut reductions and therefore is, by nature, synchronous. We then reformulate the sequent calculus in order to enforce *asynchronous* communication in the sense that a sender can proceed before a message is received, but the messages must still be received in order.

In addition to the usual point-to-point message passing primitives supported by binary session types, we also obtain logical explanations of multicast (sending a message to multiple recipients) and copy-on-receive (instantiating a new copy of a service upon receipt of a message). Our language also allows a write-once shared memory interpretation suitable for implementation. Surprisingly, this shared memory interpretation does *not* map sends to writes and receives to reads, but cuts to allocation, right rules to writes, and left rules to reads.

Additional Key Words and Phrases: Linear logic, session types, concurrency

1 INTRODUCTION

Binary session types [Honda 1993; Honda et al. 1998] arise in a natural way from a Curry-Howard interpretation of linear sequent calculus [Caires and Pfenning 2010; Wadler 2012], where a principal cut reduction is interpreted as an act of communication. This leads to a *synchronous* model of communication since a cut reduction simultaneously affects both premises of the cut. While theoretically elegant, synchronous message passing is not immediately implementable, so *asynchronous* models of message exchange have also been investigated [Carbone et al. 2008; Gay and Vasconcelos 2010]. Unlike in the π-calculus, in the setting of linear session types, synchronous and asynchronous models of computation can simulate each other [Balzer and Pfenning 2017; DeYoung et al. 2012; Pfenning and Griffith 2015].

A first contribution of this paper is the design of a variant of the sequent calculus where the cut reduction steps correspond directly to *asynchronous* communication. This calculus does not satisfy traditional cut elimination, but proofs have an analytic normal form that can be obtained via translations to and from the traditional sequent calculus. In addition, computation satisfies the expected session fidelity and global progress properties even when extended with recursive types and processes.

Our second contribution is a generalization LNL† of the modified sequent calculus to encompass LNL [Benton 1994], which uniformly integrates linear and intuitionistic layers. While it is difficult to see a feasible synchronous concurrent semantics for the (traditional) sequent calculus, its asynchronous formulation generalizes rather easily, modeling multicast (simultaneously sending one message to multiple recipients) and copy-on-receive (instantiating a new copy of a service upon message receipt).

As our third contribution we observe that the asynchronous sequent calculus also admits a *shared memory semantics*. One might expect that send actions become memory writes and receive actions become memory reads, but instead all right rules write and all left rules read. On the negative connectives this means that instead of receiving a message, a process will write a continuation

Authors' addresses: Frank Pfenning, Computer Science Department, Carnegie Mellon University, fp@cs.cmu.edu; Klaas Pruiksma, Computer Science Department, Carnegie Mellon University, kpruiksm@andrew.cmu.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

to memory and instead of sending a message, a process will jump to a stored continuation. This comes close to being a typed version of futures [Halstead 1985] except that we still have too much parallelism: we synchronize in the style of futures, but other language constructs are also concurrent. The notion of shared memory, however, remains pure in the sense that we allocate uninitialized memory but it becomes immutable once it has been written to.

Both operational interpretations leverage the expressive power of *substructural operational semantics* [Pfenning 2004; Pfenning and Simmons 2009; Simmons 2012] in an interesting way: The structural properties of semantic objects under their multiset rewriting interpretation [Cervesato and Scedrov 2009] mirror the structural properties of the types they interpret.

The remainder of the paper is organized as follows. In Sec. 2 we present a minor variant of Benton's sequent calculus for LNL, followed by an analysis of cut reduction as communication for the purely linear fragment in Sec. 3. We then formalize our operational intuition in the style of substructural operational semantics in Sec. 4, still for the linear fragment. This is generalized to encompass non-linear types in Sec. 5. We analyze the metatheory, proving session fidelity and freedom from deadlock in Sec. 6. A shared memory semantics is then presented in Sec. 7. We conclude with some additional related and future work.

2 A SEQUENT CALCULUS FOR LNL

In this section we provide a slightly streamlined formulation of LNL [Benton 1994] with an eye towards future generalization to Adjoint Logic [Pruiksma et al. 2018a; Reed 2009]. We posit two modes of truth, *linear* and *non-linear*, where linear proposition satisfy only exchange while non-linear propositions also satisfy weakening and contraction. We write L for the linear mode and U for the non-linear mode and index every proposition and connective with a mode.

$$\begin{array}{rcll} A_m, B_m, C_m & ::= & \oplus_m\{\ell : A^\ell_m\}_{\ell\in L} \mid A_m \otimes_m B_m \mid \mathbf{1}_m & \text{positives} \\ & \mid & \&_m\{\ell : A^\ell_m\}_{\ell\in L} \mid A_m \multimap_m B_m & \text{negatives} \\ & \mid & (\uparrow A_{\mathsf{L}})_{\mathsf{U}} \mid \downarrow(A_{\mathsf{U}})_{\mathsf{L}} & \text{shifts} \end{array}$$

All connectives except for the shift modalities have linear as well as non-linear versions. For example, the usual intuitionistic implication $A \longrightarrow B$ would be written as $A_{\mathsf{U}} \multimap_{\mathsf{U}} B_{\mathsf{U}}$ for non-linear propositions A and B. The shift modalities have the restriction that $\uparrow A_{\mathsf{L}}$ is of mode U and $\downarrow A_{\mathsf{U}}$ is of mode L. In Benton's notation, $\downarrow A_{\mathsf{U}} = F\, A_{\mathsf{U}}$ and $\uparrow A_{\mathsf{L}} = G\, A_{\mathsf{L}}$. It is possible to polarize the proposition via the shift operators [Pfenning and Griffith 2015] but this does not appear important for the results in the paper. We generally omit the mode subscript on the connectives since it follows from the mode of the constituent propositions. We also permit propositional variables a_m and proofs can be parametric in such variables.

We write Ψ for antecedents with implicit exchange that consist of mixed linear and non-linear propositions. We write $\Psi \geq m$ if all propositions in Ψ have a mode greater or equal to m, where $\mathsf{U} > \mathsf{L}$. Ψ_{U} stands for a collection of antecedents of mode U. All our sequents $\Psi \vdash A_m$ must syntactically obey the *independence principle* stating that $\Psi \geq m$. We enforce this in the rules when read bottom-up with appropriate premises. Just as in Benton's formulation, we think of this is being a syntactic presupposition necessary for the well-formedness of a sequent and never consider sequents that violate independence. LNL is a generalization of intuitionistic linear logic in that we can recover the exponential $!A$ as $\downarrow\uparrow A_{\mathsf{L}}$.

The inference rules for LNL can be found in Figure 1. The independence principle entails, for example, that there are three versions of the rule of cut ($m = \mathsf{U}$ and k either U or L or $m = k = \mathsf{L}$). This calculus satisfies the usual expected property of cut and identity elimination.

Theorem 2.1 (Cut and Identity [Benton 1994]). *If $\Psi \vdash A_k$ then there is a proof of the same sequent without the rule of cut and using identity only on propositional variables.*

$$\frac{}{A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Psi_1 \geq m \geq k \quad \Psi_1 \vdash A_m \quad \Psi_2, A_m \vdash C_k}{\Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}$$

$$\frac{\Psi \vdash C}{\Psi, A_{\mathsf{U}} \vdash C_k}\ \mathsf{weaken} \qquad \frac{\Psi, A_{\mathsf{U}}, A_{\mathsf{U}} \vdash C_k}{\Psi, A_{\mathsf{U}} \vdash C_k}\ \mathsf{contract}$$

$$\frac{i \in L \quad \Psi \vdash A^i_m}{\Psi \vdash \oplus\{\ell : A^\ell_m\}_{\ell \in L}}\ \oplus R_i \qquad \frac{(\text{for all } \ell \in L) \quad \Psi, A^\ell_m \vdash C_k}{\Psi, \oplus\{\ell : A^\ell_m\}_{\ell \in L} \vdash C_k}\ \oplus L$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Psi \vdash C_k}{\Psi, \mathbf{1} \vdash C_k}\ \mathbf{1}L$$

$$\frac{\Psi_1 \vdash A_m \quad \Psi_2 \vdash B_m}{\Psi_1, \Psi_2 \vdash A_m \otimes B_m}\ \otimes R \qquad \frac{\Psi, A_m, B_m \vdash C_k}{\Psi, A_m \otimes B_m \vdash C_k}\ \otimes L$$

$$\frac{(\text{for all } \ell \in L) \quad \Psi \vdash A^\ell_m}{\Psi \vdash \&\{\ell : A^\ell_m\}_{\ell \in L}}\ \&R \qquad \frac{i \in L \quad \Psi, A^i_m \vdash C_k}{\Psi, \&\{\ell : A^\ell_m\}_{\ell \in L} \vdash C_k}\ \&L_i$$

$$\frac{\Psi, A_m \vdash B_m}{\Psi \vdash A_m \multimap B_m}\ \multimap R \qquad \frac{\Psi_1 \geq m \quad \Psi_1 \vdash A_m \quad \Psi_2, B_m \vdash C_k}{\Psi_1, \Psi_2, A_m \multimap B_m \vdash C_k}\ \multimap L$$

$$\frac{\Psi_{\mathsf{U}} \vdash A_{\mathsf{U}}}{\Psi_{\mathsf{U}} \vdash \downarrow A_{\mathsf{U}}}\ \downarrow R \qquad \frac{\Psi, A_{\mathsf{U}} \vdash C_{\mathsf{L}}}{\Psi, \downarrow A_{\mathsf{U}} \vdash C_{\mathsf{L}}}\ \downarrow L$$

$$\frac{\Psi \vdash A_{\mathsf{L}}}{\Psi \vdash \uparrow A_{\mathsf{L}}}\ \uparrow R \qquad \frac{\Psi, \uparrow A_{\mathsf{L}}, A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Psi, \uparrow A_{\mathsf{L}} \vdash C_{\mathsf{L}}}\ \uparrow L$$

Fig. 1. LNL with explicit structural rules

3 CUT REDUCTION AS COMMUNICATION

In this section we consider only the operational interpretation of the linear fragment of LNL, to be generalized in Sec. 5. Under the Curry-Howard isomorphism, linear propositions correspond to types and proofs to processes. Equally significant is that in the sequent calculus, the rules of the operational semantics are derived from cut reduction [Caires and Pfenning 2010].

Fundamentally, we interpret the proof of a sequent $A_1, \ldots, A_n \vdash C$ as a process P and write

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash P :: (z : C)$$

where x_i and z are distinct *channels*. We say the P is the *client* to x_i and *provides* z. The propositions A_i and C are interpreted as *session types* prescribing the form of interactions of P along these channels. Before we get to the interpretation of specific types, we review the linear form of cut, annotated with process expressions. Here we write Δ instead of Ψ to emphasize the linearity of all propositions. Where bound variables are present we will generally indicate a dependence of P on one or more variables by writing $P[x_1, \ldots, x_n]$.

$$\frac{\Delta \vdash P :: (x : A) \quad \Delta', x : A \vdash Q :: (z : C)}{\Delta, \Delta' \vdash (x \leftarrow P[x]\ ;\ Q[x]) :: (z : C)}\ \mathsf{cut}$$

We see that cut corresponds to the parallel composition of P and Q with a private channel x provided by P and used by Q.

As an example of a logical connective we now consider the binary version of internal choice, $A \oplus B = \oplus\{\pi_1 : A, \pi_2 : B\}$. The following cut reduction (plus the symmetric one for $\oplus R_2$) should be the guide:

$$\dfrac{\dfrac{\overset{P}{\Delta \vdash A}}{\Delta \vdash A \oplus B}\,\oplus R_1 \quad \dfrac{\overset{Q_1}{\Delta', A \vdash C} \quad \overset{Q_2}{\Delta', B \vdash C}}{\Delta', A \oplus B \vdash C}\,\oplus L}{\Delta, \Delta' \vdash C}\,\mathrm{cut}_{A \oplus B} \quad \Longrightarrow \quad \dfrac{\overset{P}{\Delta \vdash A} \quad \overset{Q_1}{\Delta', A \vdash C}}{\Delta, \Delta' \vdash C}\,\mathrm{cut}_A$$

We see that the proof of the first premise has some information (namely: which disjunct is actually proved) while the proof of the second premise branches on this information. In other words, the process implementing the first premise has to communicate either π_1 or π_2 to the process implementing the second premise. We can also see that communication according to this rule is *synchronous*: the processes in both premises step forward in unison during the reduction.

Extrapolating from this example, we can see that, from the provider's perspective, positive connectives ($\oplus$, $\mathbf{1}$, $\otimes$) send information while negative connectives ($\&$, $\multimap$) receive. Of course, the client performs complementary actions.

3.1 Enforcing Asynchronous Communication

As we have seen in the preceding section, cut reduction in a standard sequent calculus implies a synchronous model of communication under the Curry-Howard correspondence. While synchronous communication is not directly implementable, it has previously been observed that we can *encode* asynchronous communication [Balzer and Pfenning 2017; DeYoung et al. 2012]. Nevertheless, from a foundational perspective this is somewhat unsatisfactory because either (a) the implementer of a session-typed language must somehow code synchronous communication at a lower level of abstraction, for example, with acknowledgments [Pfenning and Griffith 2015], or (b) the implementer uses an asynchronous semantics, departing from the Curry-Howard foundation of the calculus.

The *asynchronous π-calculus* replaces the usual action prefix for output $x\langle y\rangle.P$ by a process expression $x\langle y\rangle$ *without a continuation*, thereby ensuring that communication is asynchronous. Such a process represents the message y sent along channel x. Under our interpretation, the continuation process corresponds to the proof of the premise of a rule. Therefore, if we can restructure the sequent calculus so that the rules that send ($\oplus R$, $\mathbf{1}R$, $\otimes R$, $\downarrow R$, $\&L$, $\multimap L$, $\uparrow L$) have zero premises, then we may achieve a corresponding effect.

As an example, we consider the two right rules for $\oplus$. Reformulated as axioms, they become

$$\dfrac{}{A \vdash A \oplus B}\,\oplus R_1^0 \qquad \dfrac{}{B \vdash A \oplus B}\,\oplus R_2^0$$

In the presence of cut, these two rules together produce the same theorems as the usual two right rules. In one direction, we use cut

$$\dfrac{\Delta \vdash A \quad \dfrac{}{A \vdash A \oplus B}\,\oplus R_1^0}{\Delta \vdash A \oplus B}\,\mathrm{cut}_A \qquad \dfrac{\Delta \vdash B \quad \dfrac{}{B \vdash A \oplus B}\,\oplus R_2^0}{\Delta \vdash A \oplus B}\,\mathrm{cut}_B$$

and in the other direction we use identity

$$\dfrac{\dfrac{}{A \vdash A}\,\mathrm{id}_A}{A \vdash A \oplus B}\,\oplus R_1 \qquad \dfrac{\dfrac{}{B \vdash B}\,\mathrm{id}_B}{B \vdash A \oplus B}\,\oplus R_2$$

to derive the other rules.

Returning to the π-calculus, instead of explicitly *sending* a message $a\langle b\rangle.\, P$ we *spawn* a new process in parallel $a\langle b\rangle \mid P$. This use of parallel composition corresponds to a cut; receiving a message is achieved by cut reduction:

$$\dfrac{\dfrac{}{A \vdash A \oplus B}\,\oplus R_1^0 \quad \dfrac{\begin{array}{cc} Q_1 & Q_2 \\ \Delta', A \vdash C & \Delta', B \vdash C \end{array}}{\Delta', A \oplus B \vdash C}\,\oplus L}{\Delta', A \vdash C}\,\mathsf{cut}_{A\oplus B} \quad \Longrightarrow \quad \begin{array}{c} Q_1 \\ \Delta', A \vdash C \end{array}$$

We see the cut reduction completely eliminates the cut in one step, which corresponds precisely to receiving a message. In this example the message would be π_1 since the axiom $\oplus R_1^0$ was used; for $\oplus R_1^0$ it would be π_2.

In summary, if we restructure the sequent calculus so that the non-invertible rules (those that send) have zero premises, then (1) messages are proofs of axioms, (2) message sends are modeled by cut, and (3) message receives are a new form of cut reduction with a single continuation.

In the process we give something up, namely the traditional cut elimination theorem. For example, the sequent $\cdot \vdash \mathbf{1} \oplus \mathbf{1}$ has no cut-free proof since no rule matches this conclusion. The saving grace is that we can reach a normal form where each cut just simulates the usual rules of the sequent calculus. This can be shown by translation to the ordinary sequent calculus, applying cut elimination, and translating the result back. Proofs in this normal form have a strong subformula property. Perhaps more importantly, we have session fidelity and deadlock freedom for the corresponding process calculus even in the presence of recursive types and processes, which is ultimately what we care about for the resulting concurrent programming language.

In addition to replacing some of the rules with their axiomatic form, we also make the structural rules implicit in the standard manner: antecedents A_{U} are propagated to all premises of rule (implicit contraction) and are allowed in zero premise rules (implicit weakening). The rules for the resulting system LNL† are summarized in Fig. 2. We obtain the following theorem.

Theorem 3.1 (Adequacy of LNL†).

(i) Weakening and contraction are admissible in LNL†.
(ii) If $\Psi \vdash A$ in LNL then $\Psi \vdash A$ in LNL†.
(iii) If $\Psi \vdash A$ in LNL† then $\Psi \vdash A$ in LNL.

Proof. By straightforward inductions over the given deductions. For part (ii) we use cuts with the zero-premise rules. For part (iii) we use the LNL proofs of the zero-premise sequents. □

A simple variant of this system omits the premises marked $B|_{\mathsf{U}}$ in the $\oplus L$, $\mathbf{1}L$, and $\otimes L$ rules. These premises are logically redundant and, moreoever, contraction can be derived straightforwardly as an instance of cut:

$$\dfrac{A_{\mathsf{U}} \vdash A_{\mathsf{U}} \quad \Psi, A_{\mathsf{U}}, A_{\mathsf{U}} \vdash C_k}{\Psi, A_{\mathsf{U}} \vdash C_k}\,\mathsf{cut}$$

We prefer to keep them for two reasons: logically we adhere to the principle of implicit structural rules and computationally these additional premises allow the programmer to reuse channels if they wish to do so.

4 AN ASYNCHRONOUS OPERATIONAL SEMANTICS

We now assign process expressions to LNL† and give an asynchronous operational semantics. We consider some examples of our process notation, starting with $\oplus\{\ell : A^\ell\}_{\ell \in L}$.

$$\dfrac{i \in L}{y : A^i \vdash x.i(y) :: (x : \oplus\{\ell : A^\ell\}_{\ell \in L})}\,\oplus R_i^0$$

$$\frac{}{\Psi_\cup, A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Psi_1 \geq m \geq k \quad \Psi_\cup, \Psi_1 \vdash A_m \quad \Psi_\cup, \Psi_2, A_m \vdash C_k}{\Psi_\cup, \Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}$$

$$\frac{i \in L}{\Psi_\cup, A_i \vdash \oplus\{\ell : A^\ell_m\}_{\ell\in L}}\ \oplus R^0_i \qquad \frac{(\text{for all } \ell \in L) \quad \Psi, (\oplus\{\ell : A^\ell_m\}_{\ell\in L})|_\cup, A^\ell_m \vdash C_k}{\Psi, \oplus\{\ell : A^\ell_m\}_{\ell\in L} \vdash C_k}\ \oplus L$$

$$\frac{}{\Psi_\cup \vdash \mathbf{1}}\ \mathbf{1}R^0 \qquad \frac{\Psi, \mathbf{1}_m|_\cup \vdash C_k}{\Psi, \mathbf{1}_m \vdash C_k}\ \mathbf{1}L$$

$$\frac{}{\Psi_\cup, A_m, B_m \vdash A_m \otimes B_m}\ \otimes R^0 \qquad \frac{\Psi, (A_m \otimes B_m)|_\cup, A_m, B_m \vdash C_k}{\Psi, A_m \otimes B_m \vdash C_k}\ \otimes L$$

$$\frac{(\text{for all } \ell \in L) \quad \Psi \vdash A^\ell_m}{\Psi \vdash \&\{\ell : A^\ell_m\}_{\ell\in L}}\ \&R \qquad \frac{i \in L}{\Psi_\cup, \&\{\ell : A^\ell_m\}_{\ell\in L} \vdash A^i_m}\ \&L^0_i$$

$$\frac{\Psi, A_m \vdash B_m}{\Psi \vdash A_m \multimap B_m}\ \multimap R \qquad \frac{}{\Psi_\cup, A_m, A_m \multimap B_m \vdash B_m}\ \multimap L^0$$

$$\frac{}{\Psi_\cup, A_\cup \vdash \downarrow A_\cup}\ \downarrow R^0 \qquad \frac{\Psi, A_\cup \vdash C_\mathsf{L}}{\Psi, \downarrow A_\cup \vdash C_\mathsf{L}}\ \downarrow L$$

$$\frac{\Psi \vdash A_\mathsf{L}}{\Psi \vdash \uparrow A_\mathsf{L}}\ \uparrow R \qquad \frac{}{\Psi_\cup, \uparrow A_\mathsf{L} \vdash A_\mathsf{L}}\ \uparrow L^0$$

Fig. 2. LNL† with implicit structural rules
$B|_\cup$ in rules $\oplus L$, $\mathbf{1}L$, and $\otimes L$ means B is present iff $m = \cup$

We read $x.i(y)$ as "*send label i along x with continuation channel y*". Conversely, the recipient remains unchanged from the usual session type interpretation.

$$\frac{\text{for all } \ell \in L \quad \Delta, y : A^\ell \vdash Q_\ell[y] :: (z : C)}{\Delta, x : \oplus\{\ell : A^\ell\}_{\ell\in L} \vdash (\mathsf{case}\, x\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell\in L}) :: (z : C)}\ \oplus L$$

Now, a provider $c.i(d)$ (representing a single, asynchronous message) should interact with a client $(\mathsf{case}\, c\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell\in L})$ by transitioning to $Q_i[d]$, selecting the branch i and substituting the continuation channel d. We present this semantics in the style of *substructural operational semantics* [Cervesato et al. 2002; Pfenning 2004; Simmons 2012] which uses *multiset rewriting* [Cervesato and Scedrov 2009]. A *process configuration* is a collection of semantic objects of the form $\mathrm{proc}(c, P)$, which indicates that process P is executing and providing along channel c. We do not explicitly record the channels that P uses, because under the benign restriction that internal and external choice are never empty, these consist just of the channels free in P. For simplicity, we will make this restriction in the remainder of this paper. In a configuration, all channels c must be distinct, and the order of the objects is irrelevant. A multiset rewriting rule for semantic objects ϕ_j, ψ_k and channels a_l has the form

$$\phi_1, \ldots, \phi_m \longrightarrow \psi_1, \ldots, \psi_p \quad (a_1, \ldots a_n \text{ fresh})$$

It matches the objects ϕ_j against objects in the configuration, ignoring order but not allowing duplication, and replaces the matching objects by ψ_k after creating fresh channels a_l. Note that all

channels free in ψ_k must either occur free in ϕ_j or be among a_l, where the a_l chosen must be fresh and may not occur elsewhere in the configuration.

The first rule then has the form

$$\mathsf{proc}(c, c.i(d)), \mathsf{proc}(e, \mathsf{case}\, c\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}) \longrightarrow \mathsf{proc}(e, Q_i[d])$$

The complete process assignment can be found in Fig. 3 and the operational rules *for the linear fragment* in Fig. 4. A few further remarks on these rules. We notice that only cuts create fresh channels. In part, this is because there is no sending constructs: all "sending" of messages is accomplished by spawning an appropriate process using cut. Another observation is the difference to the asynchronous π-calculus where messages may arrive out of order. In the typed setting this would be disastrous if consecutive messages had different type, so we employ use-once channels (as already proposed by Kobayashi et al. [1996]) together with continuation channels. The continuation channels in effect form a queue that guarantees in-order arrival and thereby, ultimately, session fidelity. Here, however, this structure arises entirely from the logical side and reflects the cut reductions in LNL†. We have generally used c for the principal channel of communication and d for its intended continuation.

Finally, consider the identity rule.

$$\frac{}{y : A \vdash (x \leftarrow y) :: (x : A)}\ \mathsf{id}$$

We read the process expression $(x \leftarrow y)$ as "x *is implemented by* y" and call it *forwarding*. Intuitively, the process $(x \leftarrow y)$ is supposed to provide x and fulfills that responsibility by forwarding to y. This is sound because both channels have the same type A.

In cut elimination, there are two reductions when cut meets identity.

$$\frac{\dfrac{}{A \vdash A}\ \mathsf{id} \qquad \overset{P}{\Delta \vdash A}}{\Delta \vdash A}\ \mathsf{cut} \quad \Longrightarrow \quad \overset{P}{\Delta \vdash A} \qquad\qquad \frac{\overset{Q}{\Delta', A \vdash C} \qquad \dfrac{}{A \vdash A}\ \mathsf{id}}{\Delta', A \vdash C}\ \mathsf{cut} \quad \Longrightarrow \quad \overset{Q}{\Delta', A \vdash C}$$

These give rise to the two rules

$$\begin{array}{lllll} (\mathsf{id}C) & \mathsf{proc}(d, P[d]), \mathsf{proc}(c, c \leftarrow d) & \longrightarrow & \mathsf{proc}(c, P[c]) \\ (\mathsf{id}C') & \mathsf{proc}(c, c \leftarrow d), \mathsf{proc}(e, Q[c]) & \longrightarrow & \mathsf{proc}(e, Q[d]) \end{array}$$

In a closed system where a whole configuration promises interaction with the outside world along a single channel c_0, the rule id$C$ is preferable because with idC' a forwarding process $\mathsf{proc}(c_0, c_0 \leftarrow d)$ at the interface might never be eliminated. However, lower-level considerations might influence the choice of whether to use just one or maybe even both of these rules. Fortunately, they are confluent in the sense that applying one or the other leads to configurations that are related by renaming an internal channel and would therefore be considered identical. Other reasons for preferring id$C$ will emerge when we consider shared channels and the shared memory semantics.

4.1 Recursive Types and Processes

For the examples we add recursive types and recursively defined processes. Without further restrictions, they break the Curry-Howard isomorphism; appropriate investigation of least and greatest fixed point interpretations are the subject of ongoing work.

We encode recursive types via (potentially mutually recursive) type definitions stored in a global signature. Since our types are equirecursive we stipulate that they must be *contractive* [Gay and Hole 2005]. This allows us to silently replace type names by their definitions. In addition, we allow

$$\frac{\Psi_1 \geq m \geq k \quad \Psi_\cup, \Psi_1 \vdash P[x] :: (x : A_m) \quad \Psi_\cup, \Psi_2, x : A_m \vdash Q[x] :: (z : C_k)}{\Psi_\cup, \Psi_1, \Psi_2 \vdash (x \leftarrow P[x] \,;\, Q[x]) :: (z : C_k)}\ \mathsf{cut}$$

$$\frac{}{\Psi_\cup, y : A_m \vdash (x \leftarrow y) :: (x : A_m)}\ \mathsf{id}$$

$$\frac{i \in L}{\Psi_\cup, y : A_i \vdash x.i(y) :: (x : \oplus\{\ell : A^\ell_m\}_{\ell \in L})}\ \oplus R^0_i$$

$$\frac{(\text{for all } \ell \in L) \quad \Psi, x : (\oplus\{\ell : A^\ell_m\}_{\ell \in L})|_\cup, y : A^\ell_m \vdash Q[y] :: (z : C_k)}{\Psi, x : \oplus\{\ell : A^\ell_m\}_{\ell \in L} \vdash (\mathsf{case}\, x\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}) :: (z : C_k)}\ \oplus L$$

$$\frac{}{\Psi_\cup \vdash (x.\langle\,\rangle) :: (x : \mathbf{1})}\ \mathbf{1}R^0 \qquad \frac{\Psi, x : \mathbf{1}_m|_\cup \vdash Q :: (z : C_k)}{\Psi, x : \mathbf{1}_m \vdash (\langle\,\rangle \leftarrow x \,;\, Q) :: (z : C_k)}\ \mathbf{1}L$$

$$\frac{}{\Psi_\cup, w : A_m, y : B_m \vdash (x.\langle w, y\rangle) :: (x : A_m \otimes B_m)}\ \otimes R^0$$

$$\frac{\Psi, x : (A_m \otimes B_m)|_\cup, w : A_m, y : B_m \vdash Q[w, y] :: (z : C_k)}{\Psi, x : A_m \otimes B_m \vdash (\langle w, y\rangle \leftarrow x \,;\, Q[w, y]) :: (z : C_k)}\ \otimes L$$

$$\frac{(\text{for all } \ell \in L) \quad \Psi \vdash Q_\ell[y] :: (y : A^\ell_m)}{\Psi \vdash (\mathsf{case}\, x\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}) :: (x : \&\{\ell : A^\ell_m\}_{\ell \in L})}\ \&R$$

$$\frac{(i \in L)}{\Psi_\cup, x : \&\{\ell : A^\ell_m\}_\ell \in L \vdash x.i(y) :: (y : A^i_m)}\ \&L^0_i$$

$$\frac{\Psi, w : A_m \vdash P[w, y] :: (y : B_m)}{\Psi \vdash (\langle w, y\rangle \leftarrow x \,;\, P[w, y]) :: (x : A_m \multimap B_m)}\ \multimap R$$

$$\frac{}{\Psi_\cup, w : A_m, x : A_m \multimap B_m \vdash x.\langle w, y\rangle :: (y : B_m)}\ \multimap L^0$$

$$\frac{}{\Psi_\cup, y : A_\cup \vdash x.\mathsf{shift}(y) :: (x : \downarrow A_\cup)}\ \downarrow R^0 \qquad \frac{\Psi, y : A_\cup \vdash Q[y] :: (z : C_\lfloor)}{\Psi, x : \downarrow A_\cup \vdash (\mathsf{shift}(y) \leftarrow x \,;\, Q[y]) :: (z : C_\lfloor)}\ \downarrow L$$

$$\frac{\Psi \vdash P[y] :: (y : A_\lfloor)}{\Psi \vdash (\mathsf{shift}(y) \leftarrow x \,;\, P[y]) :: (x : \uparrow A_\lfloor)}\ \uparrow R \qquad \frac{}{\Psi_\cup, x : \uparrow A_\lfloor \vdash (x.\mathsf{shift}(y)) :: (y : A_\lfloor)}\ \uparrow L^0$$

Fig. 3. LNL† with process expressions

recursive process definitions of the form

$$\begin{array}{l} y_1{:}B_1, \ldots, y_n{:}B_n \vdash f : (x : A) \\ x \leftarrow f \leftarrow y_1, \ldots, y_n = P[x, y_1, \ldots, y_n] \end{array}$$

The first line declares the type of f, while the second defines f in terms of a process expression P. In a program we invoke it as a special form of cut

$$x \leftarrow f \leftarrow y_1, \ldots, y_n \,;\, Q[x]$$

$$
\begin{array}{lllll}
(\mathrm{cut}C) & \mathrm{proc}(c, x \leftarrow P[x]\ ;\ Q[x]) & \longrightarrow & \mathrm{proc}(a, P[a]), \mathrm{proc}(c, Q[a]) & (a \text{ fresh}) \\
(\mathrm{id}C) & \mathrm{proc}(d, P[d]), \mathrm{proc}(c, c \leftarrow d) & \longrightarrow & \mathrm{proc}(c, P[c]) & \\
(\oplus C) & \mathrm{proc}(c, c.i(d)), \mathrm{proc}(e, \mathrm{case}\ c\ (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}) & \longrightarrow & \mathrm{proc}(e, Q_i[d]) & \\
(\mathbf{1}C) & \mathrm{proc}(c, c.\langle\,\rangle), \mathrm{proc}(e, \langle\,\rangle \leftarrow c\ ;\ Q) & \longrightarrow & \mathrm{proc}(e, Q) & \\
(\otimes C) & \mathrm{proc}(c, c.\langle e, d\rangle), \mathrm{proc}(e, \langle w, y\rangle \leftarrow c\ ;\ Q[w, y]) & \longrightarrow & \mathrm{proc}(e, Q[e, d]) & \\
(\mathbin{\&}C) & \mathrm{proc}(c, \mathrm{case}\ c\ (\ell(y) \Rightarrow P_\ell[y])), \mathrm{proc}(e, c.i(d)) & \longrightarrow & \mathrm{proc}(d, P_i[d]) & \\
(\multimap C) & \mathrm{proc}(c, \langle w, y\rangle \leftarrow c\ ;\ P[w, y]), \mathrm{proc}(d, c.\langle e, d\rangle) & \longrightarrow & \mathrm{proc}(d, P[e, d]) &
\end{array}
$$

Fig. 4. LNL† operational semantics, linear fragment

which evolves to an ordinary cut with the following rule:

$$\mathrm{proc}(c, x \leftarrow f \leftarrow d_1, \ldots, d_n\ ;\ Q[x]) \longrightarrow \mathrm{proc}(a, x \leftarrow P[x, d_1, \ldots, d_n]\ ;\ Q[x])$$

If there is no continuation $Q[x]$ we have the special form of a tail call $x \leftarrow f \leftarrow y_1, \ldots, y_n$ which is a syntactic abbreviation for a cut with an identity:

$$(x \leftarrow f \leftarrow y_1, \ldots, y_n) = (x' \leftarrow f \leftarrow y_1, \ldots, y_n\ ;\ x \leftarrow x')$$

for a fresh x'. For brevity, we abbreviate $y_1, \ldots, y_n = \overline{y}$ in some transition rules.

As a last syntactic extension we consider a version of cut where the premises are reversed.

$$\frac{\Psi_1 \geq m \geq k \quad \Psi_{\mathrm{U}}, \Psi_2, x : A_m \vdash Q[x] :: (z : C_k) \quad \Psi_{\mathrm{U}}, \Psi_1 \vdash P[x] :: (x : A_m)}{\Psi_{\mathrm{U}}, \Psi_1, \Psi_2 \vdash (x \leftarrow Q[x]\ ;\ P[x]) :: (z : C_k)}\ \mathrm{cut}^{\mathrm{rev}}$$

From a logical perspective this is of course completely redundant, but it will allow us to write programs syntactically in a more natural order. We overload the notation since it can easily be disambiguated.

4.2 Example: Bit Streams

Consider the recursive type

$$\mathit{bits} = \oplus\{\mathrm{b0} : \mathit{bits}, \mathrm{b1} : \mathit{bits}, \$: \mathbf{1}\}$$

Any process providing a channel $x : \mathit{bits}$ should send a potentially infinite sequence of messages b0 and b1. If it is finite, after the last bit it should send the label \$. The continuation is then of type $\mathbf{1}$, which means it should close the channel by sending $\langle\,\rangle$. As an example, we consider how to send the number $6 = (110)_2$. The representation of numbers is in "little endian" form, that is, we actually need to send the following sequence: b0, b1, b1, \$, $\langle\,\rangle$, where the last message signifies the closure of the channel. Using the reverse cut notation, we can implement the process *six*

$$\cdot \vdash \mathit{six} :: (x : \mathit{bits})$$

as follows

$$
\begin{array}{lll}
x \leftarrow \mathit{six} = x_1 \leftarrow x.\mathrm{b0}(x_1)\ ; & \% \text{ send b0 along } x, \text{ continue as } x_1 \\
\quad x_2 \leftarrow x_1.\mathrm{b1}(x_2)\ ; & \% \text{ send b1 along } x_1, \text{ continue as } x_2 \\
\quad x_3 \leftarrow x_2.\mathrm{b1}(x_3)\ ; & \% \text{ send b1 along } x_2, \text{ continue as } x_3 \\
\quad x_4 \leftarrow x_3.\$(x_4)\ ; & \% \text{ send \$ along } x_3, \text{ continue as } x_4 \\
\quad x_4.\langle\,\rangle & \% \text{ close } x_4
\end{array}
$$

When the comment says "send" this is of course implemented as a cut (spawn) that creates a new channel representing the continuation. Executing this program providing along some initial channel $c_0 : \mathit{bits}$ yields the following configuration:

$$\begin{aligned}\mathsf{proc}(c_0, c_0 \leftarrow \mathit{six}) \longrightarrow^* \; & \mathsf{proc}(c_4, c_4.\langle\,\rangle),\\ & \mathsf{proc}(c_3, c_3.\$(c_4)),\\ & \mathsf{proc}(c_2, c_2.\mathsf{b1}(c_3)),\\ & \mathsf{proc}(c_1, c_1.\mathsf{b1}(c_2)),\\ & \mathsf{proc}(c_0, c_0.\mathsf{b0}(c_1))\end{aligned}$$

The resulting structure of messages can easily be seen to represent the desired message queue where the channels act as a kind of pointer. We will exploit this analogy in our shared memory semantics.

As a second example consider a program *plus1* for incrementing a bit stream. It is a transducer that takes a stream representing the number n to a stream that represents the number $n + 1$. The transducer *plus1* is a client to a bit stream along a channel y and provides a channel x. When y is b0 it outputs b1 along x and is finished: from then on, the output stream along x behaves exactly like the remaining input stream y. We implement this by forwarding $x \leftarrow y$. When y is b1 we have to output b0, but we also have to increment the rest of the stream (the "carry") which we accomplish by a recursive call to *plus1*.

$y : \mathit{bits} \vdash \mathit{plus1} :: (x : \mathit{bits})$

$$\begin{array}{lll}
x \leftarrow \mathit{plus1} \leftarrow y = & & \\
\quad \mathsf{case}\; y\; (\; \mathsf{b0}(y') \Rightarrow & x' \leftarrow x.\mathsf{b1}(x')\;; & \%\text{ send b1 along } x\text{, continue as } x'\\
\qquad & x' \leftarrow y' & \%\text{ implement } x' \text{ by } y'\\
\qquad \mid \mathsf{b1}(y') \Rightarrow & x' \leftarrow x.\mathsf{b0}(x')\;; & \%\text{ send b0 along } x\text{, continue as } x'\\
\qquad & x' \leftarrow \mathit{plus1} \leftarrow y' & \%\text{ continue to increment, modeling the carry bit}\\
\qquad \mid \$(y') \Rightarrow & x' \leftarrow x.\mathsf{b1}(x')\;; & \%\text{ send b1 along } x\text{, continue as } x'\\
\qquad & x'' \leftarrow x'.\$(x'')\;; & \%\text{ send \$ along } x'\text{, continue as } x''\\
\qquad & \langle\,\rangle \leftarrow y'\;; & \%\text{ wait for } y' \text{ to close}\\
\qquad & x''.\langle\,\rangle\;) & \%\text{ close } x''
\end{array}$$

We can use the *plus1* process, for example, to compute the number 8.

$\cdot \vdash \mathit{eight} :: (x : \mathit{bits})$

$$\begin{aligned}x \leftarrow \mathit{eight} = \; & x_6 \leftarrow \mathit{six}\;;\\ & x_7 \leftarrow \mathit{plus1} \leftarrow x_6\;;\\ & x \leftarrow \mathit{plus1} \leftarrow x_7\end{aligned}$$

We leave it to the reader to verify that this calculates the correct configuration representing the number 8. Moreover, this pipeline exhibits some parallelism as bits flow through the two *plus1* processes.

4.3 Example: A Binary Counter

A counter is process that can accept messages inc or val, in the form of an *external choice*. When receiving inc it increments its value and behaves again like a counter; when receiving val it turns into a stream of bits representing its current value.

$\mathit{bits} = \oplus\{\mathsf{b0} : \mathit{bits}, \mathsf{b1} : \mathit{bits}, \$: \mathbf{1}\}$
$\mathit{ctr} = \&\{\mathsf{inc} : \mathit{ctr}, \mathsf{val} : \mathit{bits}\}$

The type *ctr* provides a behavioral abstraction to the client: internal representations are completely invisible. In that sense a process providing $x : \mathit{ctr}$ can be thought of as an object with *methods* inc and val [Balzer and Pfenning 2015].

We implement a counter as a linear network of processes that behave like a bit 1, bit 0, or the counter with value zero. The first two provide the lowest order bit and are *clients* to the process representing the higher order bits, which is once again a little endian representation.

$y : ctr \vdash bit0 :: (x : ctr)$
$y : ctr \vdash bit1 :: (x : ctr)$
$\cdot \vdash zero :: (x : ctr)$

We present the implementations with short explanations in the comments. Note that here we use the ordinary rather than the reverse syntax for cut in the case of external choice.

$x \leftarrow bit0 \leftarrow y =$
  case x ($\mathsf{inc}(x') \Rightarrow x' \leftarrow bit1 \leftarrow y$ % transition to *bit1* in tail call
  | $\mathsf{val}(x') \Rightarrow x'' \leftarrow x'.\mathsf{b0}(x'')$; % respond with b0, continue as x''
  $y' \leftarrow y.\mathsf{val}(y')$; % request higher bits from y
  $x'' \leftarrow y'$) % forward bits

$x \leftarrow bit1 \leftarrow y =$
  case x ($\mathsf{inc}(x') \Rightarrow y' \leftarrow y.\mathsf{inc}(y')$; % send on carry bit along y, continue as y'
  $x' \leftarrow bit0 \leftarrow y'$ % transition to *bit0* in tail call
  | $\mathsf{val}(x') \Rightarrow x'' \leftarrow x'.\mathsf{b1}(x'')$; % respond with b1, continue as x''
  $y' \leftarrow y.\mathsf{val}(y')$; % request higher bits from y
  $x'' \leftarrow y'$) % forward bits

$x \leftarrow zero =$
  case x ($\mathsf{inc}(x') \Rightarrow z \leftarrow zero$; % spawn new *zero* process
  $x' \leftarrow bit1 \leftarrow z$ % continue as *bit1*
  | $\mathsf{val}(x') \Rightarrow x'' \leftarrow x'.\$(x'')$; % respond with $, continue as x''
  $x''.\langle\rangle$) % close channel and terminate

If we start with zero and increment twice, we obtain a process network with three processes: *zero*, *bit1*, *bit0*, properly plugged together in this order.

$\cdot \vdash two :: (x : ctr)$

$x \leftarrow two =$
  $z \leftarrow zero$; % start a counter z at *zero*
  $z' \leftarrow z.\mathsf{inc}(z')$; % increment z, continue as z'
  $z'' \leftarrow z'.\mathsf{inc}(z'')$; % increment z', continue as z''
  $x \leftarrow z''$ % implement x as z''

We can now execute this as expected

$$\begin{aligned}\mathsf{proc}(c_0, c_0 \leftarrow two) \longrightarrow^* \; & \mathsf{proc}(c_2, c_2 \leftarrow zero), \\ & \mathsf{proc}(c_1, c_1 \leftarrow bit1 \leftarrow c_2), \\ & \mathsf{proc}(c_0, c_0 \leftarrow bit0 \leftarrow c_1)\end{aligned}$$

We see here the distinction between positive types (as in the case of bit streams), where we computed processes that can be interpreted as a message queue, and negatives types, where the configuration evolved to a network of processes that accept further messages in the style of concurrent object-oriented programming. We can convert between these two representations.

$y : ctr \vdash c2b :: (x : bits)$

$x \leftarrow c2b \leftarrow y = y' \leftarrow y.\mathsf{val}(y') ; x \leftarrow y'$

$y : bits \vdash b2c :: (x : ctr)$

$x \leftarrow b2c \leftarrow y' =$
  case y ($\mathsf{b0}(y') \Rightarrow z \leftarrow b2c \leftarrow y'$; % convert remaining stream, concurrently
  $x \leftarrow bit0 \leftarrow z$ % lowest order bit is 0

$$
\begin{array}{lll}
\mid \mathsf{b1}(y') \Rightarrow z \leftarrow \mathit{b2c} \leftarrow y' \,; & \% \text{ convert remaining stream, concurrently} \\
\qquad\quad x \leftarrow \mathit{bit1} \leftarrow z & \% \text{ lowest order bit is 1} \\
\mid \$(y') \Rightarrow \langle\,\rangle \leftarrow y' \,; & \% \text{ wait for } y' \text{ to terminate} \\
\qquad\quad x \leftarrow \mathit{zero}) & \% \text{ convert empty bit stream to zero}
\end{array}
$$

5 INCORPORATING NON-LINEAR TYPES

We have presented the logical and typing rules in Figures 2 and 3 for a mixed linear and non-linear system, but we have presented the operational semantics only for the linear fragment. In this section we fill this gap and provide some examples.

A key generalization in the operational semantics is that now a process may have multiple clients. How does this come about? Consider the following example of cut in LNL†, where the modes of C and D are insignificant:

$$
\dfrac{\dfrac{}{A_\mathsf{U} \vdash A_\mathsf{U} \oplus B_\mathsf{U}}\,\oplus R_1^0 \qquad \dfrac{\overset{Q}{A_\mathsf{U} \oplus B_\mathsf{U} \vdash C} \qquad \overset{R}{A_\mathsf{U} \oplus B_\mathsf{U}, C \vdash D}}{A_\mathsf{U} \oplus B_\mathsf{U} \vdash D}\,\text{cut}}{A_\mathsf{U} \vdash C}\,\text{cut}
$$

We should be able to commute this cut upwards so the proof in the first premise, let's call it P can interact with both Q and R. There are two approaches to permitting this. The first is to use explicit rules for contraction and weakening. In this approach, the most natural semantics duplicates or discards P. This approach has been followed by Pruiksma et al. [2018b]. A second approach is to use implicit weakening and contraction as in Fig. 2 and *share* P between Q and R. We follow this here since it appears more amenable to a shared memory interpretation (see Sec. 7).

There are two key insights to model sharing elegantly in our formal semantics. The first is that we should make "messages" *persistent semantic objects* $!\phi$. Fortunately, this is a standard concept in substructural operational semantics and multiset rewriting. A generalized multiset rewriting rules allows *persistent premises* $!\phi$ in the antecedents which we match against persistent semantics objects. However, the persistent objects are not removed from the configuration. We can also add new persistent semantics objects by using $!\psi$ in the succedent. When we need to distinguish explicitly we will refer to the ordinary objects in the semantics as *ephemeral*.

The second insight is that we cannot simply make all processes $\mathsf{proc}(c_\mathsf{U}, P)$ that provide along a shared (that is, non-linear) channel persistent. For example, say that P starts with a cut. If this object were persistent, the cut would spawn unboundedly many processes since the operational rule would continue to be applicable. We solve this problem by introducing three forms of semantic objects

(1) $\mathsf{proc}(c_m, P)$ which is always ephemeral and evolves according the operational semantics,
(2) $\mathsf{msg}(c_m, P)$ which represents a message and is persistent if $m = \mathsf{U}$, otherwise ephemeral,
(3) $\mathsf{srv}(c_m, P)$ which represents a service and is persistent if $m = \mathsf{U}$, otherwise ephemeral.

The revised and generalized operational semantics is presented in Fig. 5. We write generically $c.M$ for a process sending along c, and $P\langle c\rangle$ for a process receiving along c. The precise syntax is summarized at the beginning of Sec. 6. We also use $!_m\phi$ as an abbreviation for an ephemeral ϕ if $m = \mathsf{L}$ and a persistent $!\phi$ if $m = \mathsf{U}$. In the rules for the shift the modes are predetermined by the typing rules, so we have made them explicit there.

An object $!\mathsf{msg}(c_\mathsf{U}, c_\mathsf{U}.M)$ represents a *multicast*, since this message is persistent will therefore be able to interact with all clients of c_U. The sender may not even be aware of all of its clients. Conversely, an object $!\mathsf{srv}(c_\mathsf{U}, P\langle c_\mathsf{U}\rangle)$ corresponds to a *copy-on-receive* since a fresh copy of P's continuation will be spawned when this server interacts with one of its clients.

$$
\begin{array}{ll}
(\mathrm{cut}C) & \mathsf{proc}(c, x_m \leftarrow P[x_m]\ ;\ Q[x_m]) \longrightarrow \mathsf{proc}(a_m, P[a_m]), \mathsf{proc}(c, Q[a_m]) \quad (a_m \text{ fresh}) \\
(\mathrm{id}C_1) & !_m\mathsf{msg}(d_m, d_m.M), \mathsf{proc}(c_m, c_m \leftarrow d_m) \longrightarrow !_m\mathsf{msg}(c_m, c_m.M) \\
(\mathrm{id}C_2) & !_m\mathsf{srv}(d_m, P\langle d_m\rangle), \mathsf{proc}(c_m, c_m \leftarrow d_m) \longrightarrow !_m\mathsf{srv}(c_m, P\langle c_m\rangle) \\
(\mathrm{call}) & \mathsf{proc}(c, x \leftarrow f \leftarrow \overline{d}\ ;\ Q[x]) \longrightarrow \mathsf{proc}(a, x \leftarrow P[x, \overline{d}]\ ;\ Q[x]) \\
 & \text{when } z \leftarrow f \leftarrow \overline{y} = P[z, \overline{y}] \\
(\mathrm{msg}) & \mathsf{proc}(c_m, c.M) \longrightarrow !_m\mathsf{msg}(c_m, c.M) \\
(\mathrm{srv}) & \mathsf{proc}(c_m, P\langle c\rangle) \longrightarrow !_m\mathsf{srv}(c_m, P\langle c\rangle) \\
(\oplus C) & !_m\mathsf{msg}(c_m, c.i(d)), \mathsf{proc}(e, \mathsf{case}\ c_m\ (\ell(y) \Rightarrow Q_\ell[y])_{\ell\in L}) \longrightarrow \mathsf{proc}(e, Q_i[d]) \\
(\mathbf{1}C) & !_m\mathsf{msg}(c_m, c.\langle\,\rangle), \mathsf{proc}(e, \langle\,\rangle \leftarrow c\ ;\ Q) \longrightarrow \mathsf{proc}(e, Q) \\
(\otimes C) & !_m\mathsf{msg}(c_m, c.\langle b, d\rangle), \mathsf{proc}(e, \langle w, y\rangle \leftarrow c\ ;\ Q[w, y]) \longrightarrow \mathsf{proc}(e, Q[b, d]) \\
(\mathbin{\&}C) & !_m\mathsf{srv}(c_m, \mathsf{case}\ c\ (\ell(y) \Rightarrow P_\ell[y])), \mathsf{proc}(d, c.i(d)) \longrightarrow \mathsf{proc}(d, P_i[d]) \\
(\multimap C) & !_m\mathsf{srv}(c_m, \langle w, y\rangle \leftarrow c\ ;\ P[w, y]), \mathsf{proc}(d, c.\langle b, d\rangle) \longrightarrow \mathsf{proc}(d, P[b, d]) \\
(\downarrow C) & \mathsf{msg}(c_{\mathsf{L}}, c_{\mathsf{L}}.\mathsf{shift}(d_{\mathsf{U}})), \mathsf{proc}(e, \mathsf{shift}(y_{\mathsf{U}}) \leftarrow c_{\mathsf{U}}\ ;\ Q[y_{\mathsf{U}}]) \longrightarrow \mathsf{proc}(e, Q[d_{\mathsf{U}}]) \\
(\uparrow C) & !_{\mathsf{U}}\mathsf{srv}(c_{\mathsf{U}}, \mathsf{shift}(y_{\mathsf{L}}) \leftarrow c_{\mathsf{U}}\ ;\ P[y_{\mathsf{L}}]), \mathsf{proc}(d_{\mathsf{L}}, c_{\mathsf{U}}.\mathsf{shift}(d_{\mathsf{L}})) \longrightarrow \mathsf{proc}(d_{\mathsf{L}}, P[d_{\mathsf{L}}])
\end{array}
$$

Fig. 5. LNL† operational semantics
$!_m\phi = \phi$ if $m = \mathsf{L}$ and $!_m\phi = !\phi$ if $m = \mathsf{U}$

The identity is also interesting. It seems possible to convert an object $\mathsf{proc}(c_m, c_m \leftarrow d_m)$ into a message or service (depending on the polarity of the type of c_m). This slightly awkard approach would generalize the (idC') rule we discarded. The (id$C$) rules seems more amenable for modification for mixed linear and non-linear channels in that it could either interact with a message (receive a mssage) or a server. A complication the revised rules need to account for is that both d_m and c_m may have multiple clients if $m = \mathsf{U}$. When $m = \mathsf{L}$ it specializes essentially to the previous rule, where the message/service distinction is redundant.

$$
\begin{array}{lll}
(\mathrm{id}C_1) & !_m\mathsf{msg}(d_m, d_m.M), \mathsf{proc}(c_m, c_m \leftarrow d_m) & \longrightarrow\ !_m\mathsf{msg}(c_m, c_m.M) \\
(\mathrm{id}C_2) & !_m\mathsf{srv}(d_m, P\langle d_m\rangle), \mathsf{proc}(c_m, c_m \leftarrow d_m) & \longrightarrow\ !_m\mathsf{srv}(c_m, P\langle c_m\rangle)
\end{array}
$$

In idC_2 is looks as if we have to copy P if $m = \mathsf{U}$, but in an implementation presumably a single process can remember it needs to serve two (or, in general, n) channels. Copying messages (which are always small) to send along c_m in idC_1 is more straightforward.

5.1 Example: Circuits

We call channels c_{U} that are subject to weakening and contraction *shared channels*. As an example that requires shared channels we use circuits. We start by programming a nor gate that processes infinite streams of zeros and ones.

$$\mathsf{bits}^\infty_{\mathsf{U}} = \oplus\{\mathsf{b0} : \mathsf{bits}^\infty_{\mathsf{U}}, \mathsf{b1} : \mathsf{bits}^\infty_{\mathsf{U}}\}$$

$$x : \mathsf{bits}^\infty_{\mathsf{U}}, y : \mathsf{bits}^\infty_{\mathsf{U}} \vdash \mathit{nor} :: (z : \mathsf{bits}^\infty_{\mathsf{U}})$$

$$
\begin{array}{l}
z \leftarrow \mathit{nor} \leftarrow x, y = \\
\quad \mathsf{case}\ x\ (\ \mathsf{b0}(x') \Rightarrow \mathsf{case}\ y\ (\ \mathsf{b0}(y') \Rightarrow z' \leftarrow z.\mathsf{b1}(z')\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad z' \leftarrow \mathit{nor} \leftarrow x', y' \\
\qquad\qquad\qquad\quad \mid\ \mathsf{b1}(y') \Rightarrow z' \leftarrow z.\mathsf{b0}(z')\ ;
\end{array}
$$

$$
\begin{array}{rl}
 & z' \leftarrow nor \leftarrow x', y' \,) \\
\mid \mathsf{b1}(x') \Rightarrow \mathsf{case}\, y \;\; (\, \mathsf{b0}(y') \Rightarrow & z' \leftarrow z.\mathsf{b0}(z') \;; \\
 & z' \leftarrow nor \leftarrow x', y' \\
 \mid \mathsf{b1}(y') \Rightarrow & z' \leftarrow z.\mathsf{b0}(z') \;; \\
 & z' \leftarrow nor \leftarrow x', y' \,) \,)
\end{array}
$$

This is somewhat verbose, but note that all channels here are shared. For this particular gate they could also be linear because neither is reused, but in this paper we do not treat this mode polymorphism. This illustrates that programming can be uniform at different modes, which is a significant advantage of LNL over linear logic with an exponential $!A$. Our implementation of *nor* has the property that for bits A, B, and C with $C = \neg(A \vee B)$, the following transitions are possible and characterize *nor*:

$$
\begin{array}{l}
!\mathsf{msg}(a, a.A(a')), !\mathsf{msg}(b, b.B(b')), \mathsf{proc}(c, c \leftarrow nor \leftarrow a, b) \\
\quad \longrightarrow^* \mathsf{proc}(c', c' \leftarrow nor \leftarrow a', b'), !\mathsf{msg}(c, c.C(c')) \quad (c' \text{ fresh})
\end{array}
$$

The persistent messages on shared channels a and b continue to be available after the transitions: as shared channels they may be used elsewhere. A form if distributed garbage collection, reference counting, or a more precise semantics (see Pruiksma et al. [2018b]) would be necessary to eliminate unreachable persistent semantic objects.

When we build an or-gate out of a nor-gate we need to exploit sharing to implement simple negation.

$$x : \mathsf{bits}^{\infty}_{\mathsf{U}}, y : \mathsf{bits}^{\infty}_{\mathsf{U}} \vdash or :: (z : \mathsf{bits}^{\infty}_{\mathsf{U}})$$

$$
\begin{array}{l}
z \leftarrow or \leftarrow x, y = \\
\quad u \leftarrow nor \leftarrow x, y \\
\quad z \leftarrow nor \leftarrow u, u
\end{array}
$$

An analogous computation to the above is possible, except that a shared intermediate channel d with $!\mathsf{msg}(d, d.D(d'))$ will also be created with $D = \neg(A \vee B)$ and $C = \neg D = A \vee B$.

$$
\begin{array}{l}
!\mathsf{msg}(a, a.A(a')), !\mathsf{msg}(b, b.B(b')), \\
\mathsf{proc}(c, c \leftarrow or \leftarrow a, b) \\
\quad \longrightarrow^* \; !\mathsf{msg}(d, d.D(d')) \\
\qquad\quad \mathsf{proc}(d', d' \leftarrow nor \leftarrow a', b'), \\
\qquad\quad \mathsf{proc}(c', c' \leftarrow nor \leftarrow d', d'), \\
\qquad\quad !\mathsf{msg}(c, c.C(c')) \quad (c', d' \text{ fresh})
\end{array}
$$

5.2 Example: Map

Mapping a process over a list exemplifies several new ideas: a mixed linear/non-linear program, parametric definitions, and multiplicatives. We define a whole family of types indexed by a type A, which is not formally part of the language but is expressed at the metalevel.

$$list_A = \oplus\{\mathsf{cons} : A \otimes list_A, \mathsf{nil} : \mathbf{1}\}$$

Such a list should not be viewed as a data structure in memory. Instead, it is a behavioral description of a stream of messages. A process that maps a channel of type A to one of type B will itself have type $A \multimap B$. However, this process must be shared since it needs to be applied to every element. We therefore obtain the following type and definition, where all channels not annotated are linear.

$$
\begin{array}{l}
\mathsf{f}_{\mathsf{U}} : \uparrow(A_{\mathsf{L}} \multimap B_{\mathsf{L}}), l : list_A \vdash map :: (k : list_B) \\
k \leftarrow map \leftarrow f_{\mathsf{U}}, l = \\
\quad \mathsf{case}\, l \;\; (\, \mathsf{cons}(l') \Rightarrow \langle x, l'' \rangle \leftarrow l' \;; \qquad \% \text{ receive element } x : A \text{ with continuation } l'' \\
\qquad\qquad\qquad\quad f' \leftarrow f_{\mathsf{U}}.\mathsf{shift}(f') \;; \quad \% \text{ obtain a fresh linear instance } f' \text{ of } f_{\mathsf{U}} \\
\qquad\qquad\qquad\quad y \leftarrow f'.\langle x, y \rangle \;; \qquad \% \text{ send } x \text{ to } f', \text{ response will be along fresh } y
\end{array}
$$

$$
\begin{array}{lll}
 & k' \leftarrow k.\mathsf{cons}(k') \; ; & \% \text{ select cons} \\
 & k'' \leftarrow k'.\langle y, k'' \rangle \; ; & \% \text{ send } y \text{ with continuation } k'' \\
 & k'' \leftarrow \mathit{map} \leftarrow f_{\mathsf{U}}, l'' & \% \text{ recurse with continuation channels} \\
\mid \mathsf{nil}(l') \Rightarrow & k' \leftarrow k.\mathsf{nil}(k') \; ; & \% \text{ select nil} \\
 & \langle\,\rangle \leftarrow l' \; ; & \% \text{ wait for } l' \text{ to close} \\
 & k'.\langle\,\rangle\,) & \% \text{ close } k' \text{ and terminate}
\end{array}
$$

There is some potential parallelism here: each instance of f can run concurrently with mapping over the remainder of the list since only the result channel y (which is locally created as a destination in *map*) is communicated to the client of *map*.

As a simple parametric use of this, consider a process id_A that behaves as the identity.

$$
\begin{array}{l}
\vdash \mathit{id} :: (f_{\mathsf{U}} : \uparrow(A \multimap A)) \\
f_{\mathsf{U}} \leftarrow \mathit{id} = \\
\quad \mathsf{shift}(f') \leftarrow f_{\mathsf{U}} \; ; \qquad \% \text{ obtain fresh linear channel } f' \\
\quad \langle x, f'' \rangle \leftarrow f' \; ; \qquad \% \text{ receive } x \text{ and continuation } f'' \\
\quad f'' \leftarrow x \qquad \% \text{ forward } x \text{ to } f'' \text{ and terminate this instance of } \mathit{id}
\end{array}
$$

$$
\begin{array}{l}
k : \mathit{list}_A \vdash \mathit{copy} :: (l : \mathit{list}_A) \\
l \leftarrow \mathit{copy} \leftarrow k = \\
\quad f_{\mathsf{U}} \leftarrow \mathit{id} \qquad \% \text{ start persistent identity process} \\
\quad l \leftarrow \mathit{map} \leftarrow f_{\mathsf{U}}, k \qquad \% \text{ map } f_{\mathsf{U}} \text{ over } k \text{ to obtain } l
\end{array}
$$

6 METATHEORY

Since our LNL† is very directly based on a variant of the sequent calculus the metatheoretical properties we care about are not particularly difficult to establish. We define two equivalent notions of configuration typing, one more suitable for session fidelity, the other more suitable for deadlock freedom.

We summarize the grammar of messages and services. Recall that message objects have the form $!_m\mathsf{msg}(c_m, c_m.M)$ (representing the right rule of a positive type $\oplus$, 1, $\otimes$, $\downarrow$) and service objects $!_m\mathsf{srv}(c_m, P\langle c \rangle)$ (representing the right rule of a negative type $\&$, $\multimap$, $\uparrow$.)

$$
\begin{array}{llll}
\text{Messages} & M & ::= & i(d) \mid \langle\,\rangle \mid \langle b, d \rangle \mid \mathsf{shift}(d) \\
\text{Services} & P\langle c \rangle & ::= & \mathsf{case}\, c\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L} \mid \langle z, y \rangle \leftarrow c \; ; Q[z, y] \mid \mathsf{shift}(y) \leftarrow c \; ; Q[y]
\end{array}
$$

6.1 Session Fidelity

We define $\Psi' \vdash C :: \Psi$ to express that the collection of semantic objects in C provides all the channels in Ψ and uses all the channels in Ψ'. While configurations C are unordered (subject to exchange) their typing derivations analyze them in an order where each provider $\phi(c, P)$ precedes

all clients of c.

$$\frac{}{\Psi \vdash (\cdot) :: \Psi}\ \mathsf{id} \qquad \frac{\Psi_0 \vdash C_1 :: \Psi_1 \quad \Psi_1 \vdash C_2 :: \Psi_2}{\Psi_0 \vdash (C_1, C_2) :: \Psi_2}\ \mathsf{compose}$$

$$\frac{\Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash P :: (c_m : A_m)}{\Psi_{\mathsf{U}}, \Psi', \Psi \vdash \mathsf{proc}(c, P) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{proc}$$

$$\frac{\Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash c_m.M :: (c_m : A_m)}{\Psi_{\mathsf{U}}, \Psi', \Psi \vdash !_m\mathsf{msg}(c_m, c_m.M) :: (\Psi_{\mathsf{U}}, \Psi', c : A_m)}\ \mathsf{msg}$$

$$\frac{\Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash P\langle c_m\rangle :: (c_m : A_m)}{\Psi_{\mathsf{U}}, \Psi', \Psi \vdash !_m\mathsf{srv}(c_m, P\langle c_m\rangle) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{srv}$$

One key aspect of the rules is that persistent semantic objects !msg and !srv may only depend on shared channels $d_{\mathsf{U}} : B_{\mathsf{U}}$. The other is that shared channels in Ψ_{U} may be used in P but also by other clients because it also appears in the right-hand context.

Recall the presupposition that in a configuration C all channels c with $\phi(c, P)$ are distinct. This ensures that there is no confusion among the typing of channels, including linear channels that do not appear in the external interface to a configuration C because they are provided by one object in C and used by another.

Theorem 6.1 (Session Fidelity). *If $\Psi' \vdash C :: \Psi$ and $C \longrightarrow \mathcal{D}$ then $\Psi' \vdash \mathcal{D} :: (\Psi, \Psi_{\mathsf{U}})$ where Ψ_{U} may be empty or consist of a fresh channel a_{U} not in Ψ or Ψ'.*

Proof. By cases over the possible transitions. In each case we find the corresponding objects in C and apply inversion to the typing of the object to infer the typing of the continuations. These are then sufficient to type the objects on the right-hand sides of the transitions and insert them into the evidently correct place into the typing derivation. With linear channels and ephemeral objects, the process, message, or service is removed together with the client. With shared channels, the persistent message or service remains in place so it can type the remaining clients. When executing a cut that creates a new shared channel a_{U}, this channel persists to the very right-hand side of the configuration and populates Ψ_{U} in the theorem statement. □

In order to prove freedom from deadlocks, that is, global progress, we introduce another typing for closed configurations that provides a suitable basis for the induction.

$$\frac{}{\models (\cdot) :: (\cdot)}\ \mathsf{id}$$

$$\frac{\models C :: (\Psi_{\mathsf{U}}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash P :: (c_m : A_m)}{\models (C, \mathsf{proc}(c, P)) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{proc}$$

$$\frac{\models C :: (\Psi_{\mathsf{U}}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash c_m.M :: (c_m : A_m)}{\models (C, !_m\mathsf{msg}(c_m, c_m.M)) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{msg}$$

$$\frac{\models C :: (\Psi_{\mathsf{U}}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash P\langle c_m\rangle :: (c_m : A_m)}{\models (C, !_m\mathsf{srv}(c_m, P\langle c_m\rangle)) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{srv}$$

Lemma 6.2 (Typing Equivalence). *$\cdot \vdash C :: \Psi$ iff $\models C :: \Psi$*

Proof. By simple inductions, re-associating the configuration typing to the left. □

Messages and services are blocked on the channel that they provide, waiting for a process to interact with. They therefore play somewhat the role of "values" in functional languages.

Theorem 6.3 (Deadlock Freedom). *If* $\models C :: \Psi$ *then*

(i) either $C \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$*, or*
(ii) C *consists entirely of objects* $!_m\mathsf{msg}(c_m, c_m.M)$ *or* $!_m\mathsf{srv}(c_m, P\langle c_m\rangle)$.

Proof. By induction on the structure of the given derivation. For nonempty configurations C we single out the rightmost process, message, or service $C = (C', \phi)$ and apply the induction hypothesis to C'. If C' or ϕ can step by themselves, so can C. Otherwise C' consists entirely of messages and services. At this point, we apply inversion to the typing of $\phi = \mathsf{proc}(c, P)$, which must be trying to communicate along a channel d it uses. Again by inversion, we find the provider of d is a matching message (positive type) or service (negative type) and the interaction can take place. □

Session fidelity and deadlock freedom support some simple corollaries. We call a configuration $\mathcal{F}$ *final* if $\mathcal{F}$ cannot make a transition. Then, if $\mathsf{proc}(c_0, P)$ with $\cdot \vdash P :: (c_0 : \mathbf{1}_{\mathsf{L}})$ and $\mathsf{proc}(c_0, P) \longrightarrow^* \mathcal{F}$ for a final $\mathcal{F}$, then $\mathcal{F} = (!\mathcal{F}', \mathsf{msg}(c_0, c_0.\langle\,\rangle))$ where every object in $\mathcal{F}$ is a persistent message or service.

7 A SHARED MEMORY SEMANTICS

The semantics of LNL† was carefully designed so that we did not need to send any code objects across the network, only concrete data such as labels or, for indirect access, channels. With a slight shift of perspective we can reuse almost the exact semantics to obtain a feasible shared memory implementation of LNL†.

The first, and perhaps obvious idea is to interpret channels as memory addresses. Since only cut allocates new channels, only cut allocates memory. That is perhaps already not entirely expected.

As for the operational semantics, based on the information contents of proofs, we have so far thought of the right rules for positive connectives ($\oplus$, $\mathbf{1}$, $\otimes$, $\downarrow$) and the left rules for negative connectives ($\&$, $\multimap$, $\uparrow$) as sending. A reasonable expectation might be that sending rules *write* a message to memory and, conversely, receiving rules *read* a message from memory. This approach, however, while feasible in the purely linear fragment breaks down in the presence of channels with multiple clients. Instead we follow these simple principles:

(1) *Cut allocates*
(2) *Right rules write*
(3) *Left rules read*
(4) *Identity moves or copies*

We begin by unifying the notions of message and service into the notion of a *memory cell*. We have new semantic objects of the form $!_m\mathsf{cell}(c_m, V^?)$ which may be emphemeral (for $m = \mathsf{L}$) or persistent (for $m = \mathsf{U}$). The contents of the cell can be '_' (it is empty) or V (it has been filled by a value V). The continuation channel in many process expressions becomes the address of the continuation of the data structure in memory. We may refer to this as a pointer or *continuation*

reference. Values then follow this grammar:

Pos. Vals.	V^+	::=	$i(d)$	$(\oplus)$	label k with cont. ref. d
		|	$\langle\,\rangle$	$(\mathbf{1})$	final value
		|	$\langle c, d\rangle$	$(\otimes)$	address c and cont. ref. d
		|	$\mathsf{shift}(d_{\mathsf{U}})$	$(\downarrow)$	ephemeral reference to persistent cell d_{U}
Neg. Vals.	V^-	::=	$(\ell(y) \Rightarrow Q_\ell[y])_{\ell\in L}$	$(\&)$	dispatch table based on label ℓ with cont. ref. y
		|	$\langle x, y\rangle \Rightarrow Q[x, y]$	$(\multimap)$	parameterized program continuation
		|	$\mathsf{shift}(y_{\mathsf{L}}) \Rightarrow Q[y_{\mathsf{L}}]$	$(\uparrow)$	persistent program continuation
Values	V	::=	$V^+ \mid V^-$		

We now rewrite the previous transition rules using cells. First, cut allocates a fresh cell.

$$(\mathsf{cut}C)\quad \mathsf{proc}(c, x_m \leftarrow P[x_m]\,;\, Q[x_m]) \longrightarrow \mathsf{cell}(a_m, \_), \mathsf{proc}(a_m, P[a_m]), \mathsf{proc}(c, Q[a_m])\quad (a_m \text{ fresh})$$

The object $\mathsf{cell}(a_m, \_)$ here is ephemeral independent of the mode m because it has been allocated but not yet initialized.

All the rules maintain the invariant that if there is an object $\mathsf{proc}(a, P)$ then we have a corresponding $\mathsf{cell}(a, \_)$. We think of a as the *destination* for the value of P. By the very nature of the system, all computations have a destination.

Processes whose destination is of positive type will write to the destination cell. This is independent of whether the address is of linear or shared mode.

$$(\oplus W)\quad \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, c.i(d)) \longrightarrow !_m\mathsf{cell}(c_m, i(d))$$
$$(\oplus D)\quad !_m\mathsf{cell}(c_m, i(d)), \mathsf{proc}(e, \mathsf{case}\, c_m\, (\ell(y) \Rightarrow Q_\ell[y])) \longrightarrow \mathsf{proc}(e, Q_k[d])$$

When a cell is written to (here in the $\oplus W$ rule) then it becomes persistent if the address is shared. The process reading from a cell c_m may block until the cell is initialized, which could be implemented via a lock or other low level shared memory primitives. Recall also that persistent objects $!\phi$ will remain in the configuration unchanged so it can interact with other potential clients.

As an example of a negative type, consider the transition for type $A \multimap B$.

$$(\multimap W)\quad \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, \langle z, y\rangle \leftarrow c\,;\, Q[z, y]) \longrightarrow !_m\mathsf{cell}(c_m, \langle z, y\rangle \Rightarrow Q[z, y])$$
$$(\multimap D)\quad !_m\mathsf{cell}(c_m, \langle z, y\rangle \Rightarrow Q[z, y]), \mathsf{proc}(d, c.\langle b, d\rangle) \longrightarrow \mathsf{proc}(d, Q[b, d])$$

The process executing a receive under the message-passing semantics instead writes a continuation expression to memory. At a lower level of abstraction, we would expect the implementation to construct a closure and store a pointer to the closure in c_m. Again, writing to the cell means it takes on the structural properties of its address. Reading from the cell instantiates it with the argument channel and continuation address and executes the continuation.

Writing a closure to memory is a luxury afforded by shared memory: we should not expect in general to be able send a closure as a primitive operation in the message-passing setting.

Finally, we consider forwarding. The two different rules of the message-passing semantics in Fig. 5 are unified to

$$(\mathsf{id}C)\quad !_m\mathsf{cell}(d_m, V), \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, c_m \leftarrow d_m) \longrightarrow !_m\mathsf{cell}(c_m, V)$$

Depending on m this either *moves* the value V (when $m = \mathsf{L}$) or copies the value V from cell c to d (when $m = \mathsf{U}$). In either case, values V should be expected to be small (pairs of labels and pointers, possibly a pointer to a closure), so this operation is meaningful from an (abstract) efficiency standpoint.

Since there are two forms if identity reduction in the sequent calculus, we would expect an alternative semantics to exist. Indeed, a configuration $\mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, c_m \leftarrow d_m)$ could transition to $!_m\mathsf{cell}(c_m, \mathsf{FWD}(d_m))$. Then, references to c_m have to follow chains of forwarding pointers.

$$
\begin{array}{ll}
(\mathrm{cut}C) & \mathsf{proc}(c, x_m \leftarrow P[x_m]\ ;\ Q[x_m]) \longrightarrow \mathsf{cell}(a_m, \_), \mathsf{proc}(a_m, P[a_m]), \mathsf{proc}(c, Q[a_m]) \quad (a_m \text{ fresh}) \\
(\mathrm{id}C) & !_m\mathsf{cell}(d_m, V), \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, c_m \leftarrow d_m) \longrightarrow !_m\mathsf{cell}(c_m, V) \\
(\mathrm{call}) & \mathsf{proc}(c, x \leftarrow f \leftarrow \overline{d}\ ;\ Q[x]) \longrightarrow \mathsf{proc}(a, x \leftarrow P[x, \overline{d}]\ ;\ Q[x]) \\
 & \text{when } z \leftarrow f \leftarrow \overline{y} = P[z, \overline{y}] \\
(\mathrm{write}) & \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, c\ @\ V) \longrightarrow !_m\mathsf{cell}(c_m, V) \\
(\oplus D) & !_m\mathsf{cell}(c_m, i(d)), \mathsf{proc}(e, \mathsf{case}\ c_m\ (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}) \longrightarrow \mathsf{proc}(e, Q_i[d]) \\
(\mathbf{1}D) & !_m\mathsf{cell}(c_m, \langle\,\rangle), \mathsf{proc}(e, \langle\,\rangle \leftarrow c\ ;\ Q) \longrightarrow \mathsf{proc}(e, Q) \\
(\otimes D) & !_m\mathsf{cell}(c_m, \langle b, d\rangle), \mathsf{proc}(e, \langle w, y\rangle \leftarrow c\ ;\ Q[w, y]) \longrightarrow \mathsf{proc}(e, Q[b, d]) \\
(\mathbin{\&} D) & !_m\mathsf{cell}(c_m, (\ell(y) \Rightarrow P_\ell[y])), \mathsf{proc}(c, c.i(d)) \longrightarrow \mathsf{proc}(d, P_i[d]) \\
(\multimap D) & !_m\mathsf{cell}(c_m, \langle w, y\rangle \Rightarrow P[w, y]), \mathsf{proc}(d, c.\langle b, d\rangle) \longrightarrow \mathsf{proc}(d, P[b, d]) \\
(\downarrow D) & \mathsf{cell}(c_{\mathsf{L}}, \mathsf{shift}(d_{\mathsf{U}})), \mathsf{proc}(e, \mathsf{shift}(y_{\mathsf{U}}) \leftarrow c_{\mathsf{U}}\ ;\ Q[y_{\mathsf{U}}]) \longrightarrow \mathsf{proc}(e, Q[d_{\mathsf{U}}]) \\
(\uparrow D) & !_{\mathsf{U}}\mathsf{cell}(c_{\mathsf{U}}, \mathsf{shift}(y_{\mathsf{L}}) \Rightarrow P[y_{\mathsf{L}}]), \mathsf{proc}(d_{\mathsf{L}}, c_{\mathsf{U}}.\mathsf{shift}(d_{\mathsf{L}})) \longrightarrow \mathsf{proc}(d_{\mathsf{L}}, P[d_{\mathsf{L}}])
\end{array}
$$

Fig. 6. LNL† shared memory semantics
See Fig. 7 for the definition of $c\ @\ V$

Besides tag checking (cell is empty, has a value, or has a forwarding pointer) these forwarding cells may accumulate when persistent. Moreover, the metatheory is less satisfying since the inversion properties in this formulation are weaker, so we prefer the version we have presented.

The complete set of rules can be found in Fig. 6. We have unified the rules that write using the notation introduced in Sec. 7.4.

7.1 Example: Bit Streams Revisited

Bit streams defined in Sec. 4.2 are an example of a *purely positive type*.

$$bits = \oplus\{\mathsf{b0} : bits, \mathsf{b1} : bits, \$: \mathbf{1}\}$$

This means if we have any program f such that

$$\cdot \vdash f :: (x : bits)$$

and we execute a terminating program $\mathsf{proc}(c_0, c_0 \leftarrow f)$ the final configuration will consist of a linked list of bits, plus potentially some persistent (unreachable) cells. We take the same program *six* but change the comments to describe the shared memory behavior.

$$\cdot \vdash six :: (x : bits)$$

$$
\begin{array}{lll}
x \leftarrow six = x_1 \leftarrow x.\mathsf{b0}(x_1)\ ; & \% \text{ allocate } x_1 \text{ and write } \mathsf{b0}(x_1) \text{ to } x \\
\quad x_2 \leftarrow x_1.\mathsf{b1}(x_2)\ ; & \% \text{ allocate } x_2 \text{ and write } \mathsf{b1}(x_2) \text{ to } x_1 \\
\quad x_3 \leftarrow x_2.\mathsf{b1}(x_3)\ ; & \% \text{ allocate } x_3 \text{ and write } \mathsf{b1}(x_3) \text{ to } x_2 \\
\quad x_4 \leftarrow x_3.\$(x_4)\ ; & \% \text{ allocate } x_4 \text{ and write } \$(x_4) \text{ to } x_3 \\
\quad x_4.\langle\,\rangle & \% \text{ write } \langle\,\rangle \text{ to } x_4
\end{array}
$$

The write operations do not need to happen in the order described (we still have concurrency!), but when terminated we will have reached the configuration indicated below:

$$
\begin{array}{ll}
\mathsf{proc}(c_0, c_0 \leftarrow six) \longrightarrow^* & \mathsf{cell}(c_4, \langle\,\rangle), \\
 & \mathsf{cell}(c_3, \$(c_4)), \\
 & \mathsf{cell}(c_2, \mathsf{b1}(c_3)), \\
 & \mathsf{cell}(c_1, \mathsf{b1}(c_2)), \\
 & \mathsf{cell}(c_0, \mathsf{b0}(c_1))
\end{array}
$$

Surprisingly, we have the exact same program and outcome if we defined

$$bits_{\mathsf{U}} = \oplus_{\mathsf{U}}\{\mathsf{b0} : bits_{\mathsf{U}}, \mathsf{b1} : bits_{\mathsf{U}}, \$: \mathbf{1}_{\mathsf{U}}\}$$

except that all the cells would be persistent. If we had a transducer such as the *plus1* process, then the difference in modes would manifest itself in the that ephemeral cells are deallocated when read while persistent cells, well, are persistent and would therefore have to be deallocated by a garbage collector.

7.2 Example Revisited: Circuits

The example of circuits over infinite bit streams from Sec. 5.1 can be transliterated into this new semantics, which forces addresses to be shared and therefore channels to be persistent.

$$\mathsf{bits}^{\infty}_{\mathsf{U}} = \oplus\{\mathsf{b0} : \mathsf{bits}^{\infty}_{\mathsf{U}}, \mathsf{b1} : \mathsf{bits}^{\infty}_{\mathsf{U}}\}$$

$$x : \mathsf{bits}^{\infty}_{\mathsf{U}}, y : \mathsf{bits}^{\infty}_{\mathsf{U}} \vdash nor :: (z : \mathsf{bits}^{\infty}_{\mathsf{U}})$$

$$x : \mathsf{bits}^{\infty}_{\mathsf{U}}, y : \mathsf{bits}^{\infty}_{\mathsf{U}} \vdash or :: (z : \mathsf{bits}^{\infty}_{\mathsf{U}})$$

$$\begin{array}{l} z \leftarrow or \leftarrow x, y = \\ \quad u \leftarrow nor \leftarrow x, y \\ \quad z \leftarrow nor \leftarrow u, u \end{array}$$

We do not repeat the definition of *nor*, but the *or* program now has the following effect on the configuration (which includes memory cells):

$$\begin{array}{ll} & !\mathsf{cell}(a, A(a')), !\mathsf{cell}(b, B(b')), \mathsf{proc}(c, c \leftarrow or \leftarrow a, b) \\ \longrightarrow^* & !\mathsf{cell}(d, D(d')) \\ & \mathsf{cell}(d', \_), \mathsf{proc}(d', d' \leftarrow nor \leftarrow a', b'), \\ & \mathsf{cell}(c', \_), \mathsf{proc}(c', c' \leftarrow nor \leftarrow d', d'), \\ & !\mathsf{cell}(c, C(c')) \quad (d, d', c' \text{ fresh}) \end{array}$$

7.3 Example Revisited: Map

Map provides an example of a negative linear type $A \multimap B$ and a negative shared type $\uparrow(A \multimap B)$. The program is exactly the same as given in Sec. 5.2, so we do not repeat it here, only the type definition and process declaration.

$$list_A = \oplus\{\mathsf{cons} : A \otimes list_A, \mathsf{nil} : \mathbf{1}\}$$
$$\mathsf{f}_{\mathsf{U}} : \uparrow(A_{\mathsf{L}} \multimap B_{\mathsf{L}}), l : list_A \vdash map :: (k : list_B)$$

If we execute a program of type $\cdot \vdash g :: (x : list_A)$ and it terminates, it will construct a memory representation of a list in the form

$$\begin{array}{l} \mathsf{cell}(c_0, \mathsf{cons}(c_1)), \\ \mathsf{cell}(c_1, \langle a_1, c_2 \rangle), \\ \mathsf{cell}(c_2, \mathsf{cons}(c_3)), \\ \mathsf{cell}(c_3, \langle a_2, c_4 \rangle), \\ \ldots, \\ \mathsf{cell}(c_{2n+1}, \mathsf{nil}(c_{2n+1})), \\ \mathsf{cell}(c_{2n+2}, \langle\,\rangle) \end{array}$$

where $a_1, \ldots, a_n$ are the addresses of the list elements of type A. When composed with a process such as

$$\mathsf{cell}(d_0, \_), \mathsf{cell}(d_0, d_0 \leftarrow copy \leftarrow c_0)$$

then we transition to a final state containing an exact replica of the cells from the beginning in newly allocated cells, together with a persistent reference to the identity process waiting for a shift.

$$
\begin{array}{lcl}
c \mathbin{@} i(d) & = & c.i(d) \\
c \mathbin{@} \langle\,\rangle & = & c.\langle\,\rangle \\
c \mathbin{@} \langle b, d\rangle & = & c.\langle b, d\rangle \\
c \mathbin{@} \mathsf{shift}(d_{\mathsf{U}}) & = & c.\mathsf{shift}(d_{\mathsf{U}}) \\
c \mathbin{@} (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L} & = & \mathsf{case}\, c\, (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L} \\
c \mathbin{@} \langle z, y\rangle \Rightarrow Q[z, y] & = & \langle z, y\rangle \leftarrow c\ ;\ Q[z, y] \\
c \mathbin{@} \mathsf{shift}(y_{\mathsf{L}}) \Rightarrow Q[y_{\mathsf{L}}] & = & \mathsf{shift}(y_{\mathsf{L}}) \leftarrow c\ ;\ Q[y_{\mathsf{L}}]
\end{array}
$$

Fig. 7. Value composition and decomposition

$\mathsf{cell}(d_0, \mathsf{cons}(d_1)), \mathsf{cell}(d_1, \langle a_1, d_2\rangle), \ldots, \mathsf{cell}(d_{2n+2}, \langle\,\rangle),$
$!\mathsf{cell}(f_{\mathsf{U}}, \mathsf{shift}(f') \Rightarrow \langle x, f''\rangle \leftarrow f'\ ;\ f'' \leftarrow x)$

The *map* process repeatedly invoked this by providing freshly allocated (linear) destinations for f'.

7.4 Metatheory Revisited

The development in Sec. 6 carries over with very few changes. We use cells instead of messages and services, while identity and composition remain unchanged. The typing rule for process object changes to

$$\frac{\Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash P :: (c_m : A_m)}{\Psi_{\mathsf{U}}, \Psi', \Psi \vdash^M \mathsf{cell}(c, \_), \mathsf{proc}(c, P) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{proc}$$

$$\frac{\Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash c \mathbin{@} V : (c_m : A_m)}{\Psi_{\mathsf{U}}, \Psi', \Psi \vdash^M !_m\mathsf{cell}(c_m, V) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{cell}$$

The easiest way to type values is to turn them back into process expression which is described in Fig.7.

We obtain session fidelity, which we now call type preservation since we are no longer in a message-passing setting.

Theorem 7.1 (Type Preservation). *If $\Psi' \vdash^M C :: \Psi$ and $C \longrightarrow \mathcal{D}$ then $\Psi' \vdash^M \mathcal{D} :: (\Psi, \Psi_{\mathsf{U}})$ where Ψ_{U} may be empty or consist of a fresh address not in Ψ or Ψ'.*

Proof. As in session fidelity (Thm 6.1). □

Similarly, we modify the typing of for the progress theorem.

$$\frac{}{\models^M (\cdot) :: (\cdot)}\ \mathsf{id}$$

$$\frac{\models^M C :: (\Psi_{\mathsf{U}}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash P :: (c_m : A_m)}{\models^M (C, \mathsf{cell}(c, \_), \mathsf{proc}(c, P)) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_M)}\ \mathsf{proc}$$

$$\frac{\models^M C :: (\Psi_{\mathsf{U}}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash c_m \mathbin{@} V :: (c_m : A_m)}{\models^M (C, !_m\mathsf{cell}(c_m, V)) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{cell}$$

The two forms of typing are equivalent, as before, but we elide the straightforward statement and proof.

Theorem 7.2 (Global Progress). *If $\models^M C :: \Psi$ then*

(i) either $C \longrightarrow \mathcal{D}$ for some $\mathcal{D}$, or
(ii) C consists entirely of memory cells $!_m\mathsf{cell}(c_m, V)$

Proof. As for deadlock freedom (Thm. 6.3). □

We can also ask about the relationship between the message passing and shared memory semantics. Fortunately, we have carefully constructed them so that is very easy. We define $\mathcal{C} \sim \mathcal{D}$

$$\begin{array}{l} \mathsf{proc}(c, P) \sim \mathsf{cell}(c, \_), \mathsf{proc}(c, P) \\ !_m\mathsf{msg}(c_m, c \mathbin{@} V^+) \sim !_m\mathsf{cell}(c_m, V^+) \\ !_m\mathsf{srv}(c_m, c \mathbin{@} V^-) \sim !_m\mathsf{cell}(c_m, V^-) \end{array}$$

and extend this compositionally to whole configuration. Then we have a strong bisimulation between the two formulations of the operational semantics.

Theorem 7.3 (Bisimulation). *Assume $\mathcal{C} \sim \mathcal{D}$. Then*
(i) If $\mathcal{C} \longrightarrow \mathcal{C}'$ then $\mathcal{D} \longrightarrow \mathcal{D}'$ for a $\mathcal{D}'$ with $\mathcal{C}' \sim \mathcal{D}'$
(ii) If $\mathcal{D} \longrightarrow \mathcal{D}'$ then $\mathcal{C} \longrightarrow \mathcal{D}'$ for some $\mathcal{C}'$ with $\mathcal{C}' \sim \mathcal{D}'$

Proof. By cases on the possible reductions, using consistent renaming for freshly created channels/addresses. □

8 RELATED WORK

There have been a number of research threads relating linear logic to computation, too many to survey them all here.

One class of related work uses *linear typing* for a λ-calculus under β-reduction, or, more broadly, a functional language, to capture some intensional properties of terms or to optimize execution (see, for example, [Abramsky 1993; Bernardy et al. 2016; Igarashi and Kobayashi 2000; Mackie 1993]). *Interaction nets* proposed by Lafont [1990] are a native model of computation derived from linear logic that can interpret various other concurrency models, including for example Kahn process networks [Mackie 2016].

More closely related is work on linear types for the π-calculus and concurrent computation [Bellin and Scott 1994; Kobayashi et al. 1996]. This was later recognized to be related to *session types* that prescribe interactions between communicating processes [Honda 1993; Honda et al. 1998], eventually including buffered, asynchronous communication [Carbone et al. 2008; Gay and Vasconcelos 2010]. These lines were unified by giving Curry-Howard interpretations of intuitionistic [Caires and Pfenning 2010; Caires et al. 2016] and classical [Wadler 2012] linear logic in which linear propositions (including the exponential $!A$) correspond to session types, proofs correspond to programs, and cut reduction corresponds to synchronous communication. This is also closely related to the purely linear development by Cockett and Pastro [2009] who also provide a categorical semantics. It was later observed that we can also give a consistent asynchronous semantics to linear session-typed programs [DeYoung et al. 2012] and, conversely, provide a logical explanation for acknowledgments that implement synchronization [Pfenning and Griffith 2015]. These observations, however, depart from the proofs-as-programs interpretation in that the semantics is no longer directly based on cut reduction. The use of adjunctions to decompose $!A$ in the context of session types is due to Pfenning and Griffith [2015], but the non-linear layer was populated only by $\uparrow A$ with a more general interpretation given as a challenge. As far as we are aware, none of these works provide a system in which cut reduction corresponds to asynchronous communication, nor do they include multicast or a general non-linear intuitionistic layer. Toninho et al. [2013] integrate linear concurrent programs into an ambient functional language via a contextual monad. While this is both sound and pragmatic, it does not correspond to a uniform logical system or operational semantics as we have presented here.

On the side of parallelism in functional programming, our shared memory semantics appears to be closely related to *futures*, proposed by Halstead [1985] in the dynamically typed setting of

Multilisp. A construct (future M) immediately spawns the parallel computation of M and returns a future that can be synchronized with when the value of M is needed. A future is modeled here as a shared memory address, accessible to the thread evaluating M as well as the thread that spawned the future. A significant difference is that sequential computation is the default in Multilisp, while concurrency is the default in LNL†. An interesting item of future work is to see if the proposal by Bernardy et al. [2016] to use a double-negation translation to enforce sequentiality might be used in our setting.

9 CONCLUSION

We have presented a new variant of a mixed linear/non-linear sequent calculus for Benton's LNL with a notion of cut reduction that corresponds asynchronous communication. We provide two natural operational interpretations that also include arbitrary equirecursive types and recursively defined processes. One semantics is via message passing, in which we can recognize multicast (one send with multiple recipients) in the behavior of shared positive types, and copy-on-receive in the behavior of shared negative types. We do not believe these important constructs were present in prior concurrent languages with logical underpinnings. The second interpretation is via shared memory that exploits the asymmetry of intuitionistic linear logic: right rules always write to memory and left rules always read. Both operational interpretations are in close correspondence and satisfy the expected type preservation and progress properties.

Among the most immediate items of future work, we would like to generalize the calculus to *adjoint logic* [Pruiksma et al. 2018a; Reed 2009] where a lattice of related modes with varying structural properties is available. The semantics might then be more naturally be formalized with subexponentials [Nigam and Miller 2009] or perhaps in focused adjoint logic [Pruiksma et al. 2018b]. We would also like to incorporate sharing in the sense of Balzer and Pfenning [2017]. We conjecture this would support writeable shared memory protected by critical regions with locks. Finally, we would like to consider establishing more control over parallelism using logical means as, for example, proposed by Bernardy et al. [2016].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1718267.

REFERENCES

Samson Abramsky. 1993. Computational Interpretations of Linear Logic. *Theoretical Computer Science* 111 (1993), 3–57.

Stephanie Balzer and Frank Pfenning. 2015. Objects as Session-Typed Processes. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. ACM SIGPLAN. To appear.

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. In *International Conference on Functional Programming (ICFP)*. ACM, 37:1–37:29.

G. Bellin and P. J. Scott. 1994. One the π-Calculus and Linear Logic. *Theoretical Computer Science* 135 (1994), 11–65.

Nick Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer LNCS 933, Kazimierz, Poland, 121–135. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

Jean-Philippe Bernardy, Víctor López Juan, and Josef Svenningsson. 2016. Composable Efficient Array Computations Using Linear Types. (Oct. 2016). Unpublished manuscript.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*. Springer LNCS 6269, Paris, France, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. Special Issue on Behavioural Types.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Symposium on Principles of Programming Languages (POPL'08)*, G.Necula and P.Wadler (Eds.). ACM, San Francisco, California, USA, 273–284.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. 2002. *A Concurrent Logical Framework II: Examples and Applications*. Technical Report CMU-CS-02-102. Department of Computer Science, Carnegie Mellon University. Revised May 2003.

Iliano Cervesato and Andre Scedrov. 2009. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information and Computation* 207, 10 (Oct. 2009), 1044–1077.

J.R.B. Cockett and Craig Pastro. 2009. The Logic of Message-Passing. *Science of Computer Programming* 74 (2009), 498–533.

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Proceedings of the 21st Conference on Computer Science Logic (CSL 2012)*, P. Cégielski and A. Durand (Eds.). Leibniz International Proceedings in Informatics, Fontainebleau, France, 228–242.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-Calculus. *Acta Informatica* 42, 2–3 (2005), 191–225.

Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming* 20, 1 (Jan. 2010), 19–50.

Robert H. Halstead. 1985. Multilisp: A Language for Parallel Symbolic Computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (Oct. 1985), 501–539.

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR'93)*. Springer LNCS 715, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming Languages and Systems (ESOP'98)*. Springer LNCS 1381, 122–138.

Atsushi Igarashi and Naoki Kobayashi. 2000. Garbage collection based on a linear type system. In *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC'00)*, Vol. 152.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1996. Linearity and the Pi-Calculus. In *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL'96)*, H.-J. Boehm and G. Steele (Eds.). ACM, St. Petersburg Beach, Florida, USA, 358–371.

Y. Lafont. 1990. Interaction Nets. In *17th Symposium on Principles of Programming Languages (POPL'90)*. ACM Press, San Francisco, California, 95–108.

I. Mackie. 1993. Lilac — A Functional Programming Language Based on Linear Logic. *Journal of Functional Programming* 4, 4 (1993), 395–433.

Ian Mackie. 2016. Compiling Process Networks to Interaction Nets. In *Computing with Terms and Graphs (TERMGRAPH'16) (EPTCS)*, Vol. 225. 5–14.

Vivek Nigam and Dale Miller. 2009. Algorithmic Specifications in Linear Logic with Subexponentials. In *Proceedings of the 11th International Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, Coimbra, Portugal, 129–140.

Frank Pfenning. 2004. Substructural Operational Semantics and Linear Destination-Passing Style. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, W.-N. Chin (Ed.). Springer-Verlag LNCS 3302, Taipei, Taiwan, 196. Abstract of invited talk.

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, A. Pitts (Ed.). Springer LNCS 9034, London, England, 3–22. Invited talk.

Frank Pfenning and Robert J. Simmons. 2009. Substructural Operational Semantics as Ordered Logic Programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*. IEEE Computer Society Press, Los Angeles, California, 101–110.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. 2018a. Adjoint Logic. (April 2018). http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf Unpublished manuscript.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. 2018b. Adjoint Logic and Its Concurrent Operational Interpretation. (Jan. 2018). http://www.cs.cmu.edu/~fp/papers/adjoint18.pdf Unpublished manuscript.

Jason Reed. 2009. A Judgmental Deconstruction of Modal Logic. (May 2009). http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf Unpublished manuscript.

Robert J. Simmons. 2012. *Substructural Logical Specifications*. Ph.D. Dissertation. Carnegie Mellon University. Available as Technical Report CMU-CS-12-142.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Proceedings of the European Symposium on Programming (ESOP'13)*, M.Felleisen and P.Gardner (Eds.). Springer LNCS 7792, Rome, Italy, 350–369.

Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*. ACM Press, Copenhagen, Denmark, 273–286.