A Shared-Memory Semantics for Mixed Linear and Non-Linear Session Types

Klaas Pruiksma and Frank Pfenning

Department of Computer Science, Carnegie Mellon University
kpruiskm@cs.cmu.edu, fp@cs.cmu.edu

Abstract. We present a system of session types that uniformly integrates linear and non-linear types, building on Bentons LNL. We then reformulate the sequent calculus for LNL into a form that enforces asynchronous communication. Based on this restructured sequent calculus, we provide both a message-passing semantics and a write-once shared-memory semantics. While the message-passing semantics is relatively standard when restricted to the linear fragment, the non-linear portion allows us to give a logical explanation of multicast (sending a message to multiple clients) and of replicable services (services that create a new instance upon receiving a request from a client). Our shared-memory semantics differs in two key ways: first, it allows for sharing of code in memory, while our message-passing system requires that all messages are "small", and second, in the shared-memory setting, we do not need to handle the message buffering that occurs when multiple messages are sent along the same shared channel. We exhibit a bisimulation between the two forms of semantics, thereby proving the correctness of the shared-memory semantics. Finally, we demonstrate that our shared-memory semantics can be implemented with futures. As a consequence we gain some control over the granularity of parallelism in the runtime system through the scheduling of futures, which is difficult to discern in the original session-typed programs.

Keywords: Session types · Linear logic · Futures

1 Introduction

Binary session types [19, 20] arise in a natural way from a Curry-Howard interpretation of linear sequent calculus [8, 36], 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 [10, 16]. Unlike in the π-calculus, in the setting of linear session types, synchronous and asynchronous models of computation can simulate each other [14, 28, 3].

A first contribution of this paper is the design of LNL†, a variant of Benton's mixed linear/non-linear sequent calculus LNL [5]. In LNL†, cut reduction steps correspond directly to *asynchronous* communication. Our calculus does not satisfy traditional cut elimination, but proofs have an analytic normal form satisfying the subformula property. While it is difficult to see a feasible synchronous concurrent semantics for LNL, we show that its asynchronous formulation LNL† models multicast (simultaneously sending one message to multiple recipients) and copy-on-receive (instantiating a new copy of a service upon message receipt), in addition to all the usual communication primitives supported by session types.

As our second major contribution we show 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 to memory and instead of sending a message, a process will jump to a stored continuation.

Finally, we observe that the shared memory semantics can be implemented with futures [17], essentially because memory is treated in a write-once fashion. This provides an opportunity to control, either statically or dynamically, which threads to execute in parallel, with a simple, fully sequential semantics at one extreme end. Such a sequential interpretation for the original mixed linear/nonlinear calculus of session types is not apparent, and, to our knowledge, goes significantly beyond what previous implementations provided.

All operational interpretations leverage the expressive power of *substructural operational semantics* [27, 29, 34] in an interesting way: The structural properties of semantic objects under their multiset rewriting interpretation [12] mirror the structural properties of the types they interpret.

Our various semantic specifications remain relatively abstract, so we lay out some criteria for what we consider a *message-passing* and what a *shared-memory* semantics, over and above the non-negotiable preservation and progress properties. Both of these are models for concurrency, so the first desideratum is that we can identify threads of control. For a *message-passing semantics* our principal additional requirement is that messages are small values with no remote references except for channels. In our case, they are only labels (for internal and external choice), channels (to represent delegation or continuation channels), a unit element (to represent termination), and shift tokens (to represent a phase transition between linear and shared channels). In particular, we do not send complex process expressions. For a *shared-memory semantics*, memory should also hold only small values, but these may include references to shared *continuations*. Access to shared memory should be safe in the sense that there are no write/write conflicts, and there should be no "external" data structures such as buffers to queue up write or read requests.

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 minor variant of Benton's LNL [5] in its *parsimonious* version with an eye towards future generalization to Adjoint Logic [32, 30]. We further include additive connectives because of their fundamental significance in the computational interpretation. 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 $A \to B$ 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 [28] but this does not appear important for the results in this 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 Fig. 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 1 (Cut and Identity [5]). *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.*

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

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

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

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

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

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

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

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

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

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 [8].

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 that P is the *client* to x_i (P *uses* x_i) and the *provider* of z (P *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 $P[x_1, \ldots, x_n]$.

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

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:

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

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, while 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. It has previously been observed that, in the session-typed setting, we can encode asynchronous communication in a synchronous language [14, 3]. Instead of sending a message we spawn a (small) process whose only job it is to deliver a single message. 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 [28], 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

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

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

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

and in the other direction we use identity

$$\frac{\dfrac{}{A \vdash A}\ \mathsf{id}_A}{A \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\dfrac{}{B \vdash B}\ \mathsf{id}_B}{B \vdash A \oplus B}\ \oplus R_2$$

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:

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

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 $\oplus R_2^0$ 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 the subformula property. Perhaps more importantly, we have session fidelity and deadlock freedom for the corresponding process calculus even in the presence of recursive

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

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

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

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

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

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

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

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

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

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 a 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 can prove by straightforward structural inductions that weakening and contraction are admissible in LNL† and that a sequent is provable in LNL if and only if it is provable in LNL† (see App. A for more details).

Theorem 2 (Adequacy of LNL†). *$\Psi \vdash A$ in LNL iff $\Psi \vdash A$ in LNL†.*

4 An Asynchronous Operational Semantics

We now assign process expressions to LNL† and give an asynchronous operational semantics, consistent with prior proposals [14, 16]. We consider some examples of our process notation, starting with $\oplus\{\ell : A^\ell\}_{\ell \in L}$.

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

We read $x.i(y)$ as "*send label i along x with continuation channel y*". The recipient, by contrast, 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 for the variable y. We present this semantics in the style of *substructural operational semantics* [11, 27, 34] which uses *multiset rewriting* [12]. A *process configuration* is a collection of semantic objects of the form $\mathsf{proc}(c, P)$, 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 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 are 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 [22]) 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.

We view this rule as the provider of x making way for the provider of y. This means the forwarding process terminates in a way that is transparent to the client. We omit the corresponding cut reduction.

$$(\mathsf{id}C)\ \mathsf{proc}(d, P[d]), \mathsf{proc}(c, c \leftarrow d) \longrightarrow \mathsf{proc}(c, P[c])$$

In all the other transition rules, it is easy to identify threads of control and small messages that transfer information. Here, $\mathsf{proc}(c, c \leftarrow d)$ appears to be a message, but we will have to refine this view based on whether the type of c is positive or negative (see Sec. 5 and App. B for further remarks).

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* [15]. This allows us to silently replace type names by their definitions. In addition, we allow 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

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

which evolves to an ordinary cut with the following rule:

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Psi \vdash P[y] :: (y : A_\mathsf{L})}{\Psi \vdash \mathsf{case}\, x\, (\mathsf{shift}(y) \Rightarrow P[y]) :: (x : \uparrow A_\mathsf{L})}\ \uparrow R \qquad \frac{}{\Psi_\mathsf{U}, x : \uparrow A_\mathsf{L} \vdash x.\mathsf{shift}(y) :: (y : A_\mathsf{L})}\ \uparrow L^0$$

Fig. 3. LNL† with process expressions

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

Fig. 4. LNL† operational semantics, linear fragment

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

Uniformly, messages are received using a case construct. In the examples this can be a bit awkward, especially if we are just waiting for a provider of type $\mathbf{1}$ to terminate, so we use $\langle\,\rangle \leftarrow x\ ;\ Q$ as syntactic sugar for $\mathsf{case}\, x\, (\langle\,\rangle \Rightarrow Q)$.

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

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

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

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

Any process providing a channel $x : \mathit{bits}$ should send a potentially infinite sequence of messages $\mathsf{b0}$ and $\mathsf{b1}$. 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: $\mathsf{b0}$, $\mathsf{b1}$, $\mathsf{b1}$, \$, $\langle\,\rangle$, 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

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

When the comment says "send" this is 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.

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 $\mathsf{b0}$ it outputs $\mathsf{b1}$ 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 $\mathsf{b1}$ we have to output $\mathsf{b0}$, but we also have to increment the rest of the stream (the "carry") which we accomplish by a recursive call to *plus1*.

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

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

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 a 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.

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

The type *ctr* provides a behavioral abstraction to the client: internal representations are completely invisible. In that sense a process providing $x : ctr$ can be thought of as an object with *methods* inc and val [2].

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 reverse cut.

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

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

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

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

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

We can now execute this as expected

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

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. In App. E, we show how to convert between these two representations.

5 Incorporating Non-Linear Types

We have presented the logical and typing rules in Figs. 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. We refer to non-linear channels as *shared channels*.

Usually in cut elimination with non-linear antecedents, one cut is split into several cuts, one for each use of the antecedent in the second premise of a cut. We followed this path in Pruiksma et al. [31] in a slightly different context, but it requires some significant machinery because the provider of a channel must track all of its clients. Here we pursue an alternative where all clients *share* access to the same (persistent!) message. For a realistic implementation, this should be refined into some form of *multicast* where a single message is sent to multiple clients. One possibility for such a refinement is given in prior work [31], another would be to *broadcast* a message to all potential clients and have the unintended recipients filter out the message.

In our formal semantics, therefore, messages on shared channels are modeled as *persistent semantic objects* $!\phi$. Fortunately, this is a standard concept in substructural operational semantics and multiset rewriting. A generalized multiset rewriting rule allows *persistent premises* $!\phi$ in the antecedents which we match against persistent semantics objects. The matching persistent objects are *not* removed from the configuration. We can also add new persistent semantic 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*.

A second insight is that we cannot simply make all processes $\mathsf{proc}(c_\upsilon, P)$ that provide along a shared channel persistent. For example, suppose 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,
(2) $\mathsf{msg}(c_m, c.M)$ which represents a message M on channel c and is persistent if $m = \mathsf{U}$, otherwise ephemeral,
(3) $\mathsf{srv}(c_m, \mathsf{case}\, c\, S)$ which represents a service S provided along channel c and is persistent if $m = \mathsf{U}$, otherwise ephemeral.

$$
\begin{array}{ll}
(\mathsf{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}) \\
(\mathsf{id}C^+) & !_m\mathsf{msg}(d_m, d.M), \mathsf{proc}(c_m, c \leftarrow d) \longrightarrow !_m\mathsf{msg}(c_m, c_m.M) \\
(\mathsf{id}C^-) & !_m\mathsf{srv}(d_m, \mathsf{case}\, d\, S), \mathsf{proc}(c_m, c \leftarrow d) \longrightarrow !_m\mathsf{srv}(c_m, \mathsf{case}\, c\, S) \\
(\mathsf{call}) & \mathsf{proc}(c, x \leftarrow f \leftarrow \overline{d}\ ;\ Q[x]) \longrightarrow \mathsf{proc}(c, x \leftarrow P[x, \overline{d}]\ ;\ Q[x]) \\
 & \quad \text{when } z \leftarrow f \leftarrow \overline{y} = P[z, \overline{y}] \\
(\mathsf{msg}) & \mathsf{proc}(c_m, c.M) \longrightarrow !_m\mathsf{msg}(c_m, c.M) \\
(\mathsf{srv}) & \mathsf{proc}(c_m, \mathsf{case}\, c\, S) \longrightarrow !_m\mathsf{srv}(c_m, \mathsf{case}\, c\, S) \\
(\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, \mathsf{case}\, c\, (\langle\,\rangle \Rightarrow Q)) \longrightarrow \mathsf{proc}(e, Q) \\
(\otimes C) & !_m\mathsf{msg}(c_m, c.\langle b, d\rangle), \mathsf{proc}(e, \mathsf{case}\, c\, (\langle w, y\rangle \Rightarrow 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, \mathsf{case}\, c\, (\langle w, y\rangle \Rightarrow 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{case}\, c_{\mathsf{L}}\, (\mathsf{shift}(y_{\mathsf{U}}) \Rightarrow Q[y_{\mathsf{U}}])) \longrightarrow \mathsf{proc}(e, Q[d_{\mathsf{U}}]) \\
(\uparrow C) & !_{\mathsf{U}}\mathsf{srv}(c_{\mathsf{U}}, \mathsf{case}\, 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. 5. LNL† operational semantics; $!_{\mathsf{L}}\phi = \phi$ and $!_{\mathsf{U}}\phi = !\phi$

Messages and services follow the grammar

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

and are used as $c.M$ (send M on c) and $\mathsf{case}\, c\, S$ (serve S along c). We see that a message M represent the right rule of a *positive type* ($\oplus$, 1, $\otimes$, $\downarrow$) while a service S represents the right rule of a *negative type* ($\mathbin{\&}$, $\multimap$, $\uparrow$). The revised and generalized operational semantics is presented in Fig. 5. 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 and 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. We consequently also obtain two versions of the identity rule, depending on whether it interacts with a service or with a message.

Our criteria for a message-passing semantics are satisfied, although not perhaps in an obvious manner. Objects $\mathsf{msg}()$ all represent small messages, and objects $\mathsf{srv}()$ all represent threads of control blocking on input. Some objects $\mathsf{proc}()$ must be considered threads of control, namely in the rules $\mathsf{cut}C$, $\mathsf{id}C^+$, call, msg, srv, $\oplus C$, $\mathbf{1}C$, $\otimes C$, and $\downarrow C$. However, the $\mathsf{proc}()$ objects in rules $\mathsf{id}C^-$, $\mathbin{\&}C$, $\multimap C$, and $\uparrow C$ represent small messages. We could have made this more clear in the notation, but it would unnecessarily complicate the proof of bisimilarity

between the message-passing and shared-memory semantics since the required additional transition from $\mathsf{proc}()$ to $\mathsf{msg}()$ has no counterpart in our shared-memory interpretation. We also remark on this choice in App. C, where another reason for it arises.

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.

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

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

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

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*:

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

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 of distributed garbage collection, reference counting, or a more precise semantics (see [31]) 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.

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

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

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

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 with a mode subscript are linear.

$$\mathsf{f}_{\mathsf{u}} : \uparrow(A_{\mathsf{l}} \multimap B_{\mathsf{l}}), l : list_A \vdash map :: (k : list_B)$$

$$\begin{array}{lll}
k \leftarrow map \leftarrow f_{\mathsf{u}}, l = & & \\
\quad \mathsf{case}\, l\ (\,\mathsf{cons}(l') \Rightarrow & \langle x, l'' \rangle \leftarrow l'\ ; & \%\ \text{receive element } x : A \text{ with continuation } l'' \\
& f' \leftarrow f_{\mathsf{u}}.\mathsf{shift}(f')\ ; & \%\ \text{obtain a fresh linear instance } f' \text{ of } f_{\mathsf{u}} \\
& y \leftarrow f'.\langle x, y \rangle\ ; & \%\ \text{send } x \text{ to } f', \text{ response will be along fresh } y \\
& k' \leftarrow k.\mathsf{cons}(k')\ ; & \%\ \text{select } \mathsf{cons} \\
& k'' \leftarrow k'.\langle y, k'' \rangle\ ; & \%\ \text{send } y \text{ with continuation } k'' \\
& k'' \leftarrow map \leftarrow f_{\mathsf{u}}, l'' & \%\ \text{recurse with continuation channels} \\
\quad \mid \mathsf{nil}(l') \Rightarrow & k' \leftarrow k.\mathsf{nil}(k')\ ; & \%\ \text{select } \mathsf{nil} \\
& \langle\,\rangle \leftarrow l'\ ; & \%\ \text{wait for } l' \text{ to close} \\
& k'.\langle\,\rangle\,) & \%\ \text{close } k' \text{ and terminate}
\end{array}$$

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*. A simple client of *map* can be found in App. F.

6 Metatheory

Since LNL† is very directly based on a variant of the sequent calculus, the metatheoretical properties we care about are not particularly difficult to establish.

6.1 Session Fidelity

We define $\Psi' \vdash \mathcal{C} :: \Psi$ to express that the collection of semantic objects in $\mathcal{C}$ provides all the channels in Ψ and uses all the channels in Ψ'. While configurations $\mathcal{C}$ are unordered (subject to exchange) their typing derivations analyze them in

an order where each provider $\phi(c, P)$ precedes all clients of c.

$$\frac{}{\Psi \vdash (\cdot) :: \Psi}\ \mathsf{id} \qquad \frac{\Psi_0 \vdash \mathcal{C}_1 :: \Psi_1 \quad \Psi_1 \vdash \mathcal{C}_2 :: \Psi_2}{\Psi_0 \vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Psi_2}\ \mathsf{compose}$$

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

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

$$\frac{\Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash \mathsf{case}\, c\, S :: (c_m : A_m)}{\Psi_{\mathsf{U}}, \Psi', \Psi \vdash !_m\mathsf{srv}(c_m, \mathsf{case}\, c\, S) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{srv}$$

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 in a process, message, or service but also by other clients because Ψ_{U} also appears in the right-hand context.

Theorem 3 (Session Fidelity). *If $\Psi' \vdash \mathcal{C} :: \Psi$ and $\mathcal{C} \longrightarrow \mathcal{D}$ 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'.*

Our system also satisfies freedom from deadlocks, that is, global progress. The proof (sketched in App. C) is fairly standard.

Theorem 4 (Deadlock Freedom). *If $\cdot \vdash \mathcal{C} :: \Psi$ then*

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

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. This 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, as each client of a shared channel whose type is negative (as determined by its top-level connective) can send along the channel. In order to have these sends correspond to writes, we would need to build some sort of buffer that is foreign to a shared memory semantics and ruled out by our criteria given in the introduction. Following these simple principles instead avoids these difficulties.

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

Under this model we can still identify threads of control, but they are different from the message-passing interpretation. In particular, threads of control *terminate* when they write to memory, although in a realistic implementation, very short-lived processes would be compiled to operations directly writing to memory.

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 *continuation reference*. We obtain that a value is either a message M or a service S which functions as a continuation (As defined in Section 5).

$$\text{Value } V ::= M \mid S$$

Note that, emphatically, continuations (which are either code, or contain pointers to code) are not messages that can be sent, but there is no difficulty keeping them in shared memory.

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

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

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

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.

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

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 they can interact with other potential clients.

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

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

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 reference 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.

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

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

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 addresses, possibly a reference to a closure), so this operation is meaningful from an (abstract) efficiency standpoint.

The complete set of rules can be found in Fig. 6. Here, every $\mathsf{proc}()$ object represents a thread of control and every $\mathsf{cell}()$ object (whether ephemeral or persistents) represents a shared memory cell. It is easily verified that this satisfies our criteria mapped out for a shared-memory semantics

7.1 Example: Bit Streams Revisited

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

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

This means if we have any program f such that

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

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.

$$
\begin{array}{ll}
(\mathsf{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]) \\
 & \qquad\qquad (a_m \text{ fresh}) \\
(\mathsf{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) \\
(\mathsf{call}) & \mathsf{proc}(c, x \leftarrow f \leftarrow \overline{d}\ ;\ Q[x]) \longrightarrow \mathsf{proc}(c, x \leftarrow P[x, \overline{d}]\ ;\ Q[x]) \\
 & \text{when } z \leftarrow f \leftarrow \overline{y} = P[z, \overline{y}] \\
(\mathsf{write}^+) & \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, c.M) \longrightarrow !_m\mathsf{cell}(c_m, M) \\
(\mathsf{write}^-) & \mathsf{cell}(c_m, \_), \mathsf{proc}(c_m, \mathsf{case}\, c\, S) \longrightarrow !_m\mathsf{cell}(c_m, S) \\
(\oplus D) & !_m\mathsf{cell}(c_m, i(d)), \mathsf{proc}(e, \mathsf{case}\, c\, (\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, \mathsf{case}\, c\, (\langle\,\rangle \Rightarrow Q)) \longrightarrow \mathsf{proc}(e, Q) \\
(\otimes D) & !_m\mathsf{cell}(c_m, \langle b, d\rangle), \mathsf{proc}(e, \mathsf{case}\, c\, (\langle w, y\rangle \Rightarrow Q[w, y]) \longrightarrow \mathsf{proc}(e, Q[b, d]) \\
(\& D) & !_m\mathsf{cell}(c_m, (\ell(y) \Rightarrow P_\ell[y])_{\ell \in L}), \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{case}\, c_{\mathsf{L}}\, (\mathsf{shift}(y_{\mathsf{U}}) \Rightarrow 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

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

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

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:

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

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

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

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 are not 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.

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

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

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

$z \leftarrow \mathit{or} \leftarrow x, y =$
$\quad u \leftarrow \mathit{nor} \leftarrow x, y\ ;$
$\quad z \leftarrow \mathit{nor} \leftarrow u, u$

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

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

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.

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

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

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

where $a_1, \ldots, a_n$ are the addresses of the list elements of type A. When composed with a copying process (see App. F) such as

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

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.

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 rules for semantic object change to

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

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

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

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

Theorem 5 (Type Preservation). *If $\Psi' \vdash^{M} \mathcal{C} :: \Psi$ and $\mathcal{C} \longrightarrow \mathcal{D}$ 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. 3).

Global progress also follows as before (see App. D).

Theorem 6 (Global Progress). *If $\cdot \vdash^{M} \mathcal{C} :: \Psi$ then*

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

Proof. As for deadlock freedom (Thm. 4).

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

$$\begin{array}{l}\mathsf{proc}(c, P) \sim \mathsf{cell}(c, \_), \mathsf{proc}(c, P) \\ !_m\mathsf{msg}(c_m, c.M) \sim !_m\mathsf{cell}(c_m, M) \\ !_m\mathsf{srv}(c_m, \mathsf{case}\, c\, S) \sim !_m\mathsf{cell}(c_m, S)\end{array}$$

and extend this compositionally to whole configuration. Then we have a strong bisimulation between the two formulations of the operational semantics, even though some threads of control in the message-passing semantics become memory contents (srv objects), while some messages (proc objects in rules $\mathsf{id}C^{-}$, $\&C$, $\multimap C$, and $\uparrow C$) become threads of control.

Theorem 7 (Bisimulation). *Assume $\mathcal{C} \sim \mathcal{D}$. Then*

(i) If $\mathcal{C} \longrightarrow \mathcal{C}'$ then $\mathcal{D} \longrightarrow \mathcal{D}'$ for some $\mathcal{D}'$ 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.

7.5 Implementation of Shared Memory in Futures

Our shared memory semantics has several similarities to systems using *futures*, as described by Halstead [17] in the dynamically typed setting of Multilisp. Just as the expression $\mathsf{fut}(e)$ spawns a computation executing e and returns a future that can be synchronized with (or *touched*) when the value of e is needed, our cuts spawn new computations along with memory cells which will eventually contain the value of the newly spawned process. A key difference is that in Halstead's setting, the default behaviour (without inserting futures) is sequential, while our semantics is concurrent by default. An interesting item of future work is to see if the proposal by Bernardy et al. [6] to use a double-negation translation to enforce sequentiality might be applicable to our setting.

However, we can take advantage of the similarities between our shared-memory semantics and that of futures to give an implementation of our system in a language with futures (see App. G for some details of the implementation). Any scheduling algorithm for futures, including, for example, a simple sequential one, can then schedule our session-typed concurrent program. Our implementation also strongly suggests a *mixed linear/non-linear language of futures* that deallocates linear futures as soon as they are touched. Linear futures (without a full type system), were already sketched by Blelloch and Reid-Miller [7] and considered in the context of parallel complexity analysis. We leave detailed design and analysis of such a language for future work.

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, [1, 24, 21, 6]). *Interaction nets* proposed by Lafont [23] are a native model of computation derived from linear logic that can interpret various other concurrency models, including for example Kahn process networks [25].

More closely related is work on linear types for the π-calculus and concurrent computation [4, 22]. This was later recognized to be related to *session types* that prescribe interactions between communicating processes [19, 20], eventually including buffered, asynchronous communication [10, 16]. These lines were unified by giving Curry-Howard interpretations of intuitionistic [8, 9] and classical [36] 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 [13] who also provide a categorical semantics. It was later observed that we can also give a consistent asynchronous semantics to linear session-typed programs [14] and, conversely, provide a logical explanation for acknowledgments that implement synchronization [28].

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 has previously been proposed [28], 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. [35] 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.

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* [32, 30] where a lattice of related modes with varying structural properties is available. The semantics might then be more naturally be formalized with subexponentials [26] or perhaps in focused adjoint logic [31]. We would also like to incorporate sharing in the sense of Balzer et al. [3]. We conjecture this would support writeable shared memory protected by critical regions with locks.

References

1. Abramsky, S.: Computational interpretations of linear logic. Theoretical Computer Science **111**, 3–57 (1993)
2. Balzer, S., Pfenning, F.: Objects as session-typed processes. In: Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! 2015). ACM SIGPLAN (2015)
3. Balzer, S., Pfenning, F.: Manifest sharing with session types. In: International Conference on Functional Programming (ICFP). pp. 37:1–37:29. ACM (Sep 2017)
4. Bellin, G., Scott, P.J.: One the π-calculus and linear logic. Theoretical Computer Science **135**, 11–65 (1994)

5. Benton, N.: A mixed linear and non-linear logic: Proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94). pp. 121–135. Springer LNCS 933, Kazimierz, Poland (Sep 1994), an extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge
6. Bernardy, J.P., Juan, V.L., Svenningsson, J.: Composable efficient array computations using linear types (Oct 2016), unpublished manuscript
7. Blelloch, G.E., Reid-Miller, M.: Pipeling with futures. Theory of Computing Systems **32**, 213–239 (1999)
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010). pp. 222–236. Springer LNCS 6269, Paris, France (Aug 2010)
9. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016), special Issue on Behavioural Types.
10. Carbone, M., Honda, K., Yoshida, N.: Multiparty asynchronous session types. In: G.Necula, P.Wadler (eds.) Proceedings of the 35th Symposium on Principles of Programming Languages (POPL'08). pp. 273–284. ACM, San Francisco, California, USA (Jan 2008)
11. Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: A concurrent logical framework II: Examples and applications. Tech. Rep. CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University (2002), revised May 2003
12. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation **207**(10), 1044–1077 (Oct 2009)
13. Cockett, J., Pastro, C.: The logic of message-passing. Science of Computer Programming **74**, 498–533 (2009)
14. DeYoung, H., Caires, L., Pfenning, F., Toninho, B.: Cut reduction in linear logic as asynchronous session-typed communication. In: Cégielski, P., Durand, A. (eds.) Proceedings of the 21st Conference on Computer Science Logic. pp. 228–242. CSL 2012, Leibniz International Proceedings in Informatics, Fontainebleau, France (Sep 2012)
15. Gay, S.J., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica **42**(2–3), 191–225 (2005)
16. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. Journal of Functional Programming **20**(1), 19–50 (Jan 2010)
17. Halstead, R.H.: Multilisp: A language for parallel symbolic computation. ACM Transactions on Programming Languages and Systems **7**(4), 501–539 (Oct 1985)
18. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, second edn. (Apr 2016)
19. Honda, K.: Types for dyadic interaction. In: 4th International Conference on Concurrency Theory. pp. 509–523. CONCUR'93, Springer LNCS 715 (1993)
20. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: 7th European Symposium on Programming Languages and Systems. pp. 122–138. ESOP'98, Springer LNCS 1381 (1998)
21. Igarashi, A., Kobayashi, N.: 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 (2000)
22. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. In: Boehm, H.J., Steele, G. (eds.) Proceedings of the 23rd Symposium on Principles

of Programming Languages (POPL'96). pp. 358–371. ACM, St. Petersburg Beach, Florida, USA (Jan 1996)
23. Lafont, Y.: Interaction nets. In: 17th Symposium on Principles of Programming Languages (POPL'90). pp. 95–108. ACM Press, San Francisco, California (1990)
24. Mackie, I.: Lilac — a functional programming language based on linear logic. Journal of Functional Programming **4**(4), 395–433 (1993)
25. Mackie, I.: Compiling process networks to interaction nets. In: Computing with Terms and Graphs (TERMGRAPH'16). EPTCS, vol. 225, pp. 5–14 (2016)
26. Nigam, V., Miller, D.: Algorithmic specifications in linear logic with subexponentials. In: Proceedings of the 11th International Conference on Principles and Practice of Declarative Programming (PPDP). pp. 129–140. ACM, Coimbra, Portugal (Sep 2009)
27. Pfenning, F.: Substructural operational semantics and linear destination-passing style. In: Chin, W.N. (ed.) Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04). p. 196. Springer-Verlag LNCS 3302, Taipei, Taiwan (Nov 2004), abstract of invited talk
28. Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A. (ed.) Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015). pp. 3–22. Springer LNCS 9034, London, England (Apr 2015), invited talk
29. Pfenning, F., Simmons, R.J.: Substructural operational semantics as ordered logic programming. In: Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009). pp. 101–110. IEEE Computer Society Press, Los Angeles, California (Aug 2009)
30. Pruiksma, K., Chargin, W., Pfenning, F., Reed, J.: Adjoint logic (Apr 2018), http://www.cs.cmu.edu/ fp/papers/adjoint18b.pdf, unpublished manuscript
31. Pruiksma, K., Chargin, W., Pfenning, F., Reed, J.: Adjoint logic and its concurrent operational interpretation (Jan 2018), http://www.cs.cmu.edu/ fp/papers/adjoint18.pdf, unpublished manuscript
32. Reed, J.: A judgmental deconstruction of modal logic (May 2009), http://www.cs.cmu.edu/ jcreed/papers/jdml2.pdf, unpublished manuscript
33. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. Lisp and Symbolic Computation **6**(3/4), 289–360 (1993)
34. Simmons, R.J.: Substructural Logical Specifications. Ph.D. thesis, Carnegie Mellon University (Nov 2012), available as Technical Report CMU-CS-12-142
35. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: M.Felleisen, P.Gardner (eds.) Proceedings of the European Symposium on Programming (ESOP'13). pp. 350–369. Springer LNCS 7792, Rome, Italy (Mar 2013)
36. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th International Conference on Functional Programming. pp. 273–286. ICFP 2012, ACM Press, Copenhagen, Denmark (Sep 2012)

A Basic Properties

Theorem 2 (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.

B The Identity

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

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

These give rise to the two potential rules

$$(\mathsf{id}C)\;\; \mathsf{proc}(d, P[d]), \mathsf{proc}(c, c \leftarrow d) \longrightarrow \mathsf{proc}(c, P[c])$$
$$(\mathsf{id}C')\; \mathsf{proc}(c, c \leftarrow d), \mathsf{proc}(e, Q[c]) \longrightarrow \mathsf{proc}(e, Q[d])$$

In a closed system where a whole configuration promises interaction with the outside world along a single channel c_0, the rule $\mathsf{id}C$ is preferable because with $\mathsf{id}C'$ a forwarding process $\mathsf{proc}(c_0, c_0 \leftarrow d)$ at the interface might never be eliminated. However, lower-level considerations (implementation details) might influence the choice of which one (or maybe even both) of these rules to use. 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.

Allowing mixed linear/nonlinear types (and therefore messages and services, see Sec. 5) further complicates the picture. It seems possible to convert an object $\mathsf{proc}(c_m, c_m \leftarrow d_m)$ into a message or a service (depending on the polarity of the type of c_m). This slightly awkward approach would generalize the $(\mathsf{id}C')$ rule we discarded. The $(\mathsf{id}C)$ rule seems more amenable for modification for mixed linear and non-linear channels in that it could either interact with a message (receive a message) or with 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.

$$(\mathsf{id}C^+)\; !_m\mathsf{msg}(d_m, d_m.M), \mathsf{proc}(c_m, c_m \leftarrow d_m) \longrightarrow !_m\mathsf{msg}(c_m, c_m.M)$$
$$(\mathsf{id}C^-)\; !_m\mathsf{srv}(d_m, \mathsf{case}\, d\, S), \mathsf{proc}(c_m, c_m \leftarrow d_m) \longrightarrow !_m\mathsf{srv}(c_m, \mathsf{case}\, c\, S)$$

In $\mathsf{id}C^-$ is looks as if we have to copy S if $m = \mathsf{U}$, but in an implementation, presumably a single process can remember it needs to serve two (or, in general, n) channels. Alternately, we can think of it as spawning an identical copy of the current process. Copying messages (which are always small) to send along c_m in $\mathsf{id}C^+$ is more straightforward.

Since there are two forms of identity reduction in the sequent calculus, we would expect an alternative shared-memory semantics for identity 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. 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 that we have presented.

C Metatheory for the Message-Passing Semantics

Recall the presupposition that in a configuration $\mathcal{C}$ 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 $\mathcal{C}$ because they are provided by one object in $\mathcal{C}$ and used by another.

Theorem 3 (Session Fidelity). *If $\Psi' \vdash \mathcal{C} :: \Psi$ and $\mathcal{C} \longrightarrow \mathcal{D}$ 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 $\mathcal{C}$ 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 form of typing for closed configurations that provides a suitable

basis for the induction.

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

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

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

$$\frac{\models \mathcal{C} :: (\Psi_{\mathsf{U}}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_{\mathsf{U}}, \Psi \vdash \mathsf{case}\, c\, S :: (c_m : A_m)}{\models (\mathcal{C}, !_m\mathsf{srv}(c_m, \mathsf{case}\, c\, S)) :: (\Psi_{\mathsf{U}}, \Psi', c_m : A_m)}\ \mathsf{srv}$$

Lemma 1 (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. Besides the connection to the shared-memory semantics, this is another reason why have designated exactly these objects as msg or srv even though other processes proc also act like messages.

Theorem 4 (Deadlock Freedom). *If* $\models \mathcal{C} :: \Psi$ *then*

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

Proof. By induction on the structure of the given derivation. For nonempty configurations $\mathcal{C}$ we single out the rightmost process, message, or service $\mathcal{C} = (\mathcal{C}', \phi)$ and apply the induction hypothesis to $\mathcal{C}'$. If $\mathcal{C}'$ or ϕ can step by themselves, so can $\mathcal{C}$. Otherwise $\mathcal{C}'$ 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.

D Metatheory for the Shared Memory Semantics

The proof of type preservation for the shared-memory semantics is completely analogous to the message-passing semantics and we therefore do not detail it here further.

Regarding progress, we design typing for *closed* configurations along the same lines as for the shared memory semantics.

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

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

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

$$\frac{\models^M \mathcal{C} :: (\Psi_\mathsf{U}, \Psi', \Psi) \quad \Psi \geq m \quad \Psi_\mathsf{U}, \Psi \vdash \mathsf{case}\, c\, S :: (c_m : A_m)}{\models^M (\mathcal{C}, !_m\mathsf{cell}(c_m, S)) :: (\Psi_\mathsf{U}, \Psi', c_m : A_m)}\ \mathsf{cell}^-$$

The two forms of typing for shared memory configurations are equivalent, just as before, so we elide the straightforward statement and proof.

Theorem 6 (Global Progress). *If* $\models^M \mathcal{C} :: \Psi$ *then*

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

Proof. As for deadlock freedom (Thm. 4).

E Converting Between Internal and External Choice

Continuing the example from Sec. 4.2, here is one way to convert between bit streams and counters.

$y : \mathit{ctr} \vdash \mathit{c2b} :: (x : \mathit{bits})$
$x \leftarrow \mathit{c2b} \leftarrow y = y' \leftarrow y.\mathsf{val}(y')\ ;\ x \leftarrow y'$

$y : \mathit{bits} \vdash \mathit{b2c} :: (x : \mathit{ctr})$
$x \leftarrow \mathit{b2c} \leftarrow y' =$
  $\mathsf{case}\, y\ (\ \mathsf{b0}(y') \Rightarrow z \leftarrow \mathit{b2c} \leftarrow y'\ ;$ % convert remaining stream, concurrently
  $x \leftarrow \mathit{bit0} \leftarrow z$ % lowest order bit is 0
  $\mid \mathsf{b1}(y') \Rightarrow z \leftarrow \mathit{b2c} \leftarrow y'\ ;$ % convert remaining stream, concurrently
  $x \leftarrow \mathit{bit1} \leftarrow z$ % lowest order bit is 1
  $\mid \$(y') \Rightarrow\ \langle\,\rangle \leftarrow y'\ ;$ % wait for y' to terminate
  $x \leftarrow \mathit{zero}\ )$ % convert empty bit stream to zero

F A Simple Client for *map*

In Sec. 5.2 we showed a program for mapping a shared process over a sequence of messages. Here we show a simple parametric client. We consider a process id_A that behaves as an identity and map it over a sequence.

$\vdash id :: (f_{\mathsf{U}} : \uparrow(A \multimap A))$

$f_{\mathsf{U}} \leftarrow id =$
  $\mathsf{case}\ f_{\mathsf{U}}\ (\mathsf{shift}(f') \Rightarrow$ % obtain fresh linear channel f'
    $\mathsf{case}\ f'\ (\langle x, f'' \rangle \Rightarrow$ % receive x and continuation f'' along f'
      $f'' \leftarrow x))$ % forward x to f'' and terminate

$k : list_A \vdash copy :: (l : list_A)$

$l \leftarrow copy \leftarrow k =$
  $f_{\mathsf{U}} \leftarrow id$ % start persistent identity process
  $l \leftarrow map \leftarrow f_{\mathsf{U}}, k$ % map f_{U} over k to obtain l

G Implementation in Futures

We present a translation of our shared-memory language into a functional language with futures. We have not rigorously proven the correctness of this translation because the target language has only non-linear types and therefore leaves behind persistent memory subject to garbage collection. Also, for simplicity we have ignored for now recursive types and fixed point expressions. In future work, we plan to design a mixed linear/non-linear language which presents a more faithful target and formally prove the correctness of the translation. Nevertheless, we believe the translation is intuitive and sheds some light on our shared-memory semantics, which is the main point of this appendix.

We take as our target language a standard extension of the simply-typed λ-calculus with a variety of type constructors to match the variety of types available in the linear setting, to which we add futures (see, for example, Harper's textbook [18, Ch. 38])

In particular, we include pairs, $(\langle a, b \rangle : A \times B)$, a unit type $(\langle\,\rangle : \mathbf{1})$, labelled sums $(i(y) : +\{\ell : A^\ell\}_{\ell \in L})$, and lazy records $(\langle\!\langle \ell \Rightarrow e_\ell \rangle\!\rangle_{\ell \in L} : \wedge\{\ell : A^\ell\}_{\ell \in L})$, and, of course, futures $(\mathsf{fut}(e) : \mathsf{fut}(A))$. We formulate our semantics not with an explicit store data structure, but following the idea of destination-passing style [11] and substructural operational semantics [34] where variables are always instantiated by destinations that represent store locations. We also restrict values to "small values" which cannot contain other values, using destinations instead. The advantage for us of this approach is that it aligns better with our shared-memory system, making the correspondence more apparent.

The types for this language are shown in Fig. 7, and the syntax of the language itself is given in Fig. 8.

We now present the semantics of this target language. The typing (Fig. 9) is standard, but included for clarity, and the dynamic semantics (Fig. 10) are given in destination-passing style. We have semantic objects of the form

$$A, B ::= +\{\ell : A^\ell\}_{\ell \in L} \mid \wedge\{\ell : A^\ell\}_{\ell \in L} \mid \mathbf{1} \mid A \times B \mid A \to B \mid \mathsf{fut}(A)$$

Fig. 7. Types for functional target language with futures

Expressions $e ::= d$			destinations
$\mid x$			variables
$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$			let-binding
$\mid i(e)$	$\mid \mathsf{case}\ e\ (\ell(x) \Rightarrow e_\ell)_{\ell \in L}$		disjoint sums ($+\{\,\}$)
$\mid \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L}$	$\mid \pi_i(e)$		lazy records ($\wedge\{\,\}$)
$\mid \langle\,\rangle$	$\mid \mathsf{let}\ \langle\,\rangle = e_1\ \mathsf{in}\ e_2$		unit ($\mathbf{1}$)
$\mid \langle v, w \rangle$	$\mid \mathsf{let}\ \langle x, y \rangle = e_1\ \mathsf{in}\ e_2$		pairs ($\times$)
$\mid \lambda x.\ e$	$\mid e_1\ e_2$		functions ($\to$)
$\mid \mathsf{fut}(e)$	$\mid \mathsf{touch}(e)$		futures ($\mathsf{fut}()$)

Values $v, w ::= d \mid x \mid i(d) \mid \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L} \mid \langle\,\rangle \mid \langle d_1, d_2 \rangle \mid \lambda x.\ e$

Fig. 8. Target language

$\mathsf{eval}\ e\ d$**:** Evaluate expression e with destination d.
$\mathsf{cont}\ d_1.k\ d$**:** Wait for a value in destination d_1 and then execute continuation k with destination d.
$!\mathsf{retn}\ v\ d$**:** Return value v to destination d.

All returns are persistent, both for simplicity and because of the absence of linear typing in the target language. The return objects therefore represent the state of the store.

Now, we define a translation from our language to this target language, first on types (see Fig. 11). We extend this translation in the natural (pointwise) manner to contexts Ψ. Similarly, we will write $\mathsf{fut}(\llbracket\Psi\rrbracket)$ for the context obtained by wrapping each type in $\llbracket\Psi\rrbracket$ in a future.

We now go on to define a translation from process terms to expressions in our target language (see Fig. 12). The guiding typing principle of this translation is that if $\Psi \vdash P :: (z : A_m)$, then we should have that $\mathsf{fut}(\llbracket\Psi\rrbracket) \vdash \llbracket P\rrbracket^z : \llbracket A_m\rrbracket$. Intuitively, the variable z on the translation indicates what channel P provides, and therefore allows us to treat the provided channel differently from the channels used by P. In particular, this lets us disambiguate between the process terms for $\oplus$ and $\&$ or $\otimes$ and $\multimap$, which are identical other than in which channel is provided.

Theorem 7 (Type-correctness of the translation). *If* $\Psi \vdash P :: (z : A_m)$, *then* $\mathsf{fut}(\llbracket\Psi\rrbracket) \vdash \llbracket P\rrbracket^z : \llbracket A_m\rrbracket$.

Proof (sketch). This follows from a straightforward induction over the derivation of $\Psi \vdash P :: (z : A_m)$. In each case, we simply apply the translation and appeal to the inductive hypothesis. There is then only one typing rule for the target language which is applicable, and applying it (along with a future introduction or

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathsf{var} \qquad \frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2, (x : A) \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C}\ \mathsf{let}$$

$$\frac{i \in L \quad \Gamma \vdash e : A^i}{\Gamma \vdash i(e) : +\{\ell : A^\ell\}_{\ell \in L}}\ {+}I \qquad \frac{\Gamma_1 \vdash e : +\{\ell : A^\ell\}_{\ell \in L} \quad \Gamma_2, x : A^\ell \vdash e_\ell : C \ \text{ for all } \ell \in L}{\Gamma_1, \Gamma_2 \vdash \mathsf{case}\ e\ (\ell(x) \Rightarrow e_\ell)_{\ell \in L} : C}\ {+}E$$

$$\frac{\Gamma \vdash e_\ell : A^\ell \quad \text{for all } \ell \in L}{\Gamma \vdash \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L} : \wedge\{\ell : A^\ell\}_{\ell \in L}}\ {\wedge}I \qquad \frac{i \in L \quad \Gamma \vdash e : \wedge\{\ell : A^\ell\}_{\ell \in L}}{\Gamma \vdash \pi_i(e) : A^i}\ {\wedge}E$$

$$\frac{}{\Gamma \vdash \langle\,\rangle : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{1} \quad \Gamma_2 \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle\,\rangle = e_1\ \mathsf{in}\ e_2 : C}\ \mathbf{1}E$$

$$\frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash \langle e_1, e_2 \rangle : A \times B}\ {\times}I \qquad \frac{\Gamma_1 \vdash e_1 : A \times B \quad \Gamma_2, x : A, y : B \vdash e_2 : C}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ \langle x, y \rangle = e_1\ \mathsf{in}\ e_2 : C}\ {\times}E$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ {\to}I \qquad \frac{\Gamma_1 \vdash e_1 : A \to B \quad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1\, e_2 : B}\ {\to}E$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{fut}(e) : \mathsf{fut}(A)}\ \mathsf{fut}I \qquad \frac{\Gamma \vdash e : \mathsf{fut}(A)}{\Gamma \vdash \mathsf{touch}(e) : A}\ \mathsf{fut}E$$

Fig. 9. Target typing

elimination rule) using the results of the inductive hypothesis yields the desired result.

Now, we note that this translation does not exercise the full target language. In fact, its image is the fragment of the target language which is in A-normal form [33]. We give here the syntax for this fragment, as well as a streamlined semantics which takes multiple steps at once in order to keep all expressions produced by evaluation inside this fragment.

We also note that the rules in Fig. 14 are remarkably similar to those of our shared-memory interpretation, as given in Fig. 6. In fact, if we ignore the persistence of the all !retn() objects each rule of this fragment of our target language corresponds exactly to a rule of our shared memory system and vice versa, provided we expand the write^+ and write^- rules into separate cases for each connective. As noted before, we do not rigorously prove this correspondence because the future semantics is imprecise (it does not handle linearity) and we believe a better, more precise semantics based on a mixed linear/non-linear λ-calculus with futures is just around the corner.

$$
\begin{array}{l}
\mathsf{eval}\ (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)\ d \longrightarrow \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1.(\mathsf{let}\ x = d_1\ \mathsf{in}\ e_2)\ d \quad (d_1\ \text{fresh}) \\
!\mathsf{retn}\ v\ d_1, \mathsf{cont}\ d_1.(\mathsf{let}\ x = d_1\ \mathsf{in}\ e_2)\ d \longrightarrow \mathsf{eval}\ ([d_1/x]e_2)\ d \\
\mathsf{eval}\ d_1\ d \longrightarrow \mathsf{cont}\ d_1.(d_1)\ d \\
!\mathsf{retn}\ v\ d_1, \mathsf{cont}\ d_1.(d_1)\ d \longrightarrow !\mathsf{retn}\ v\ d \\
\hline
\mathsf{eval}\ (\mathsf{fut}(e))\ d \longrightarrow \mathsf{eval}\ e\ d_1, !\mathsf{retn}\ d_1\ d \quad (d_1\ \text{fresh}) \\
\mathsf{eval}\ (\mathsf{touch}(e))\ d \longrightarrow \mathsf{eval}\ e\ d_1, \mathsf{cont}\ d_1.(\mathsf{touch}(d_1))\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(\mathsf{touch}(d_1))\ d, !\mathsf{retn}\ v\ d_1 \longrightarrow !\mathsf{retn}\ v\ d \\
\hline
\mathsf{eval}\ (i(e))\ d \longrightarrow \mathsf{eval}\ e\ d_1, \mathsf{cont}\ d_1.(i(d_1))\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(i(d_1))\ d, !\mathsf{retn}\ v\ d_1 \longrightarrow !\mathsf{retn}\ (i(d_1))\ d \\
\mathsf{eval}\ \langle\!\langle \ell : e_\ell \rangle\!\rangle\ d \longrightarrow !\mathsf{retn}\ \langle\!\langle \ell : e_\ell \rangle\!\rangle\ d \\
\mathsf{eval}\ \langle\,\rangle\ d \longrightarrow !\mathsf{retn}\ \langle\,\rangle\ d \\
\mathsf{eval}\ \langle e_1, e_2 \rangle\ d \longrightarrow \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1.(\langle d_1, e_2 \rangle)\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(\langle d_1, e_2 \rangle)\ d, !\mathsf{retn}\ v\ d_1 \longrightarrow \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2.(\langle v, d_2 \rangle)\ d \quad (d_2\ \text{fresh}) \\
\mathsf{cont}\ d_2.(\langle d_1, d_2 \rangle)\ d, !\mathsf{retn}\ w\ d_2 \longrightarrow !\mathsf{retn}\ \langle d_1, d_2 \rangle\ d \\
\mathsf{eval}\ (\lambda x.\ e)\ d \longrightarrow !\mathsf{retn}\ (\lambda x.\ e)\ d \\
\hline
\mathsf{eval}\ (\mathsf{case}\ e\ (\ell(x) \Rightarrow e_\ell)_{\ell \in L})\ d \longrightarrow \mathsf{eval}\ e\ d_1, \mathsf{cont}\ d_1.(\mathsf{case}\ d_1\ (\ell(x) \Rightarrow e_\ell)_{\ell \in L})\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(\mathsf{case}\ d_1\ (\ell(x) \Rightarrow e_\ell)_{\ell \in L})\ d, !\mathsf{retn}\ (i(d_2))\ d_1 \longrightarrow \mathsf{eval}\ ([d_2/x]e_i)\ d \\
\mathsf{eval}\ (\pi_i(e))\ d \longrightarrow \mathsf{eval}\ e\ d_1, \mathsf{cont}\ d_1.(\pi_i(d_1)) \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(\pi_i(d_1))\ d, !\mathsf{retn}\ \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L}\ d_1 \longrightarrow \mathsf{eval}\ e_i\ d \\
\mathsf{eval}\ (\mathsf{let}\ \langle\,\rangle = e_1\ \mathsf{in}\ e_2)\ d \longrightarrow \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1.(\mathsf{let}\ \langle\,\rangle = d_1\ \mathsf{in}\ e_2)\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(\mathsf{let}\ \langle\,\rangle = d_1\ \mathsf{in}\ e_2)\ d, !\mathsf{retn}\ \langle\,\rangle\ d_1 \longrightarrow \mathsf{eval}\ e_2\ d \\
\mathsf{eval}\ (\mathsf{let}\ \langle x, y \rangle = e_1\ \mathsf{in}\ e_2)\ d \longrightarrow \mathsf{eval}\ d_1\ e_1, \mathsf{cont}\ d_1.(\mathsf{let}\ \langle x, y \rangle = d_1\ \mathsf{in}\ e_2)\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(\mathsf{let}\ \langle x, y \rangle = d_1\ \mathsf{in}\ e_2)\ d, !\mathsf{retn}\ \langle d_2, d_3 \rangle\ d_1 \longrightarrow \mathsf{eval}\ ([d_2/x][d_3/y]e_2)\ d \\
\mathsf{eval}\ (e_1\ e_2)\ d \longrightarrow \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1.(d_1\ e_2)\ d \quad (d_1\ \text{fresh}) \\
\mathsf{cont}\ d_1.(d_1\ e_2)\ d, !\mathsf{retn}\ (\lambda x.\ e)\ d_1 \longrightarrow \mathsf{cont}\ d_2.(d_1\ d_2)\ d \quad (d_2\ \text{fresh}) \\
\mathsf{cont}\ d_2.(d_1\ d_2)\ d, !\mathsf{retn}\ (\lambda x.\ e)\ d_1, !\mathsf{retn}\ v\ d_2 \longrightarrow \mathsf{eval}\ ([v/x]e)\ d
\end{array}
$$

Fig. 10. Dynamic semantics of target language

$$
\begin{aligned}
[\![\oplus\{\ell : A^\ell_m\}_{\ell \in L}]\!] &= +\{\ell : \mathsf{fut}([\![A^\ell_m]\!])\}_{\ell \in L} \\
[\![\&\{\ell : A^\ell_m\}_{\ell \in L}]\!] &= \wedge\{\ell : \mathsf{fut}([\![A^\ell_m]\!])\}_{\ell \in L} \\
[\![\mathbf{1}]\!] &= \mathbf{1} \\
[\![A_m \otimes B_m]\!] &= \mathsf{fut}([\![A_m]\!]) \times \mathsf{fut}([\![B_m]\!]) \\
[\![A_m \multimap B_m]\!] &= \mathsf{fut}([\![A_m]\!]) \to \mathsf{fut}([\![B_m]\!]) \\
[\![{\downarrow}A_{\mathsf{U}}]\!] &= +\{\mathsf{shift} : \mathsf{fut}([\![A_{\mathsf{U}}]\!])\} \\
[\![{\uparrow}A_{\mathsf{L}}]\!] &= \wedge\{\mathsf{shift} : \mathsf{fut}([\![A_{\mathsf{L}}]\!])\}
\end{aligned}
$$

Fig. 11. Type translation

$$
\begin{array}{lcl}
[\![x \leftarrow P[x]; Q[x]]\!]^z & = & \mathsf{let}\ x = \mathsf{fut}([\![P[x]]\!]^x)\ \mathsf{in}\ [\![Q[x]]\!]^z \\
[\![z \leftarrow x]\!]^z & = & \mathsf{touch}(x) \\
\hline
\multicolumn{3}{c}{\text{Positive connectives}} \\
[\![z.i(y)]\!]^z & = & i(y) \\
[\![\mathsf{case}\ x\ (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}]\!]^z & = & \mathsf{case}\ \mathsf{touch}(x)\ (\ell(y) \Rightarrow [\![Q_\ell[y]]\!]^z) \\
[\![z.\langle\,\rangle]\!]^z & = & \langle\,\rangle \\
[\![\mathsf{case}\, x\, \{\langle\,\rangle \Rightarrow P\}]\!]^z & = & \mathsf{let}\ \langle\,\rangle = \mathsf{touch}(x)\ \mathsf{in}\ [\![P]\!]^z \\
[\![z.\langle x, y\rangle]\!]^z & = & \langle x, y\rangle \\
[\![\mathsf{case}\, w\, \{\langle x, y\rangle \Rightarrow P[x, y]\}]\!]^z & = & \mathsf{let}\ \langle x, y\rangle = \mathsf{touch}(w)\ \mathsf{in}\ [\![P[x, y]]\!]^z \\
[\![z.\mathsf{shift}(y)]\!]^z & = & \mathsf{shift}(y) \\
[\![\mathsf{case}\, x\, \{\mathsf{shift}(y) \Rightarrow P[y]\}]\!]^z & = & \mathsf{case}\ \mathsf{touch}(x)\ (\mathsf{shift}(y) \Rightarrow [\![P[y]]\!]^z) \\
\hline
\multicolumn{3}{c}{\text{Negative connectives}} \\
[\![\mathsf{case}\ x\ (\ell(y) \Rightarrow Q_\ell[y])_{\ell \in L}]\!]^x & = & \langle\!\langle \ell : [\![Q_\ell[y]]\!]^y \rangle\!\rangle_{\ell \in L} \\
[\![z.i(y)]\!]^y & = & \pi_i(\mathsf{touch}(z)) \\
[\![\mathsf{case}\, w\, \{\langle x, y\rangle \Rightarrow P[x, y]\}]\!]^w & = & \lambda x.\ [\![P[x, y]]\!]^y \\
[\![z.\langle x, y\rangle]\!]^y & = & \mathsf{touch}(z)\, x \\
[\![\mathsf{case}\, x\, \{\mathsf{shift}(y) \Rightarrow P[y]\}]\!]^y & = & \langle\!\langle \mathsf{shift} \Rightarrow [\![P[y]]\!]^y \rangle\!\rangle \\
[\![z.\mathsf{shift}(y)]\!]^y & = & \pi_{\mathsf{shift}}(\mathsf{touch}(z))
\end{array}
$$

Fig. 12. Translation from process terms to expressions

$$
\begin{array}{rll}
\text{Expressions } e ::= & \mathsf{touch}(x) & \mid \mathsf{let}\ x = \mathsf{fut}(e_1)\ \mathsf{in}\ e_2 \\
\mid & i(x) & \mid \mathsf{case}\ \mathsf{touch}(x)\ (\ell(y) \Rightarrow e_\ell)_{\ell \in L} \\
\mid & \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L} & \mid \pi_i(\mathsf{touch}(x)) \\
\mid & \langle\,\rangle & \mid \mathsf{let}\ \langle\,\rangle = \mathsf{touch}(x)\ \mathsf{in}\ e_2 \\
\mid & \langle x, y\rangle & \mid \mathsf{let}\ \langle x, y\rangle = \mathsf{touch}(z)\ \mathsf{in}\ e_2 \\
\mid & \lambda x.\ e & \mid \mathsf{touch}(x)\, y
\end{array}
$$

$$
\text{Values } v, w ::= d \mid x \mid i(d) \mid \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L} \mid \langle\,\rangle \mid \langle d_1, d_2\rangle \mid \lambda x.\ e
$$

Fig. 13. Target language (fragment)

$$\mathsf{eval}\ (\mathsf{let}\ x = \mathsf{fut}(e_1)\ \mathsf{in}\ e_2)\ d \longrightarrow \mathsf{eval}\ e_1\ d_1, \mathsf{eval}\ ([d_1/x]e_2)\ d \quad (d_1\ \text{fresh})$$
$$\mathsf{eval}\ (\mathsf{touch}(d_1))\ d, !\mathsf{retn}\ v\ d_1 \longrightarrow !\mathsf{retn}\ v\ d$$

$$\mathsf{eval}\ (i(d_1))\ d \longrightarrow !\mathsf{retn}\ (i(d_1))\ d$$
$$\mathsf{eval}\ \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L}\ d \longrightarrow !\mathsf{retn}\ \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L}\ d$$
$$\mathsf{eval}\ \langle\,\rangle\ d \longrightarrow !\mathsf{retn}\ \langle\,\rangle\ d$$
$$\mathsf{eval}\ \langle d_1, d_2 \rangle\ d \longrightarrow !\mathsf{retn}\ \langle d_1, d_2 \rangle\ d$$
$$\mathsf{eval}\ (\lambda x.\ e)\ d \longrightarrow !\mathsf{retn}\ (\lambda x.\ e)\ d$$

$$\mathsf{eval}\ (\mathsf{case}\ \mathsf{touch}(d_1)\ (\ell(x) \Rightarrow e_\ell)_{\ell \in L})\ d, !\mathsf{retn}\ (i(d_2))\ d_1 \longrightarrow \mathsf{eval}\ ([d_2/x]e_i)\ d$$
$$\mathsf{eval}\ (\pi_i(\mathsf{touch}(d_1)))\ d, !\mathsf{retn}\ \langle\!\langle \ell : e_\ell \rangle\!\rangle_{\ell \in L}\ d_1 \longrightarrow \mathsf{eval}\ e_i\ d$$
$$\mathsf{eval}\ (\mathsf{let}\ \langle\,\rangle = \mathsf{touch}(d_1)\ \mathsf{in}\ e)\ d, !\mathsf{retn}\ \langle\,\rangle\ d_1 \longrightarrow \mathsf{eval}\ e\ d$$
$$\mathsf{eval}\ (\mathsf{let}\ \langle x, y \rangle = \mathsf{touch}(d_1)\ \mathsf{in}\ e)\ d, !\mathsf{retn}\ \langle d_2, d_3 \rangle\ d_1 \longrightarrow \mathsf{eval}\ ([d_2/x][d_3/y]e)\ d$$
$$\mathsf{eval}\ (\mathsf{touch}(d_1)\ d_2)\ d, !\mathsf{retn}\ (\lambda x.\ e)\ d_1, !\mathsf{retn}\ v\ d_2 \longrightarrow \mathsf{eval}\ ([d_2/x]e)\ d$$

Fig. 14. Semantics of the fragment