Polarized Substructural Session Types

Frank Pfenning[1] and Dennis Griffith[2]

[1] Carnegie Mellon University
fp@cs.cmu.edu
[2] University of Illinois at Urbana-Champaign
dgriffi3@illinois.edu

Abstract. The deep connection between session-typed concurrency and linear logic is embodied in the language SILL that integrates functional and message-passing concurrent programming. The exacting nature of linear typing provides strong guarantees, such as global progress, absence of deadlock, and race freedom, but it also requires explicit resource management by the programmer. This burden is alleviated in an affine type system where resources need not be used, relying on a simple form of garbage collection.
In this paper we show how to effectively support both linear and affine typing in a single language, in addition to the already present unrestricted (intuitionistic) types. The approach, based on Benton's adjoint construction, suggests that the usual distinction between synchronous and asynchronous communication can be viewed through the lens of modal logic. We show how polarizing the propositions into positive and negative connectives allows us to elegantly express synchronization in the type instead of encoding it by extra-logical means.

1 Introduction

Session types prescribe the communication behavior of concurrent message-passing processes [13, 14]. Anticipated with some analogies for some time [11, 23], session types have recently been placed upon the firm foundation of linear logic via a Curry-Howard interpretation of linear propositions as types, proofs as processes, and cut reduction as communication. Variations apply for both intuitionistic [5, 6] and classical [24] linear logic. This has enabled the application of proof-theoretic techniques in this domain, for example, developing logical relations [17], corecursion [22], and parametricity and behavioral polymorphism [4]. It has also given rise to the design of SILL, a modular extension of an underlying functional language with session-typed concurrency [21].

Practical experience with a SILL prototype has led to a number of new questions. For example, should the type system really be *linear*, where all resources must be fully accounted for by the programmer, or should it be *affine*[16], where resources may be reclaimed by a form of garbage collection? Another question concerns the underlying model of communication: should it be synchronous or asynchronous? The proof theory does not provide a definitive answer to this question, supporting both. The purpose of this paper is to show that we can have our cake and eat it, too, in both cases. First, we combine linear and affine types in an elegant and proof-theoretically justified way, slightly reformulating unrestricted types along the way. Second, we show how to support synchronous and asynchronous communication patterns in a single language, again taking our inspiration from proof theory.

The central idea behind the first step is to generalize Benton's LNL [3] in the spirit of Reed's adjoint logic [18]. This stratifies the propositions into linear, affine, and unrestricted ones, with modal operators *shifting* between the strata. For example, the familiar exponential of linear logic $!A$ is decomposed into two shifting modalities, one going from A (which is linear) into the unrestricted layer, and one going from the unrestricted layer back to the linear one. Similar modalities connect the linear and affine layers of the language.

The main idea behind the second step is to *polarize* the presentation of linear logic [15], segregating positive (sending) connectives from negative (receiving) connectives. Surprisingly, the two sublanguages of propositions can be connected by new versions of the shift modalities, fully consistent with the adjoint construction, leading to a pleasantly coherent language.

In the rest of this note we walk through these steps, taking small liberties with previously published notations for the sake of consistency.

2 Linear Logic and Session Types

We give here only the briefest review of linear logic and its deep connection to session types. The interested reader is referred to [5, 6, 21] for further background.

The key idea of linear logic [12] is to view logical propositions as resources: they must be used exactly once in a proof. We adopt the intuitionistic version [2], which is defined via a *linear hypothetical judgment* [8]

$$A_1, \ldots, A_n \vdash A$$

where the hypotheses $A_1, \ldots, A_n$ must be used exactly once in the proof of the conclusion A. We do not care about the order of the assumptions, treating them like a multiset, and use Δ to denote such a multiset. The judgmental rules (sometimes called structural rules) explain the meaning of the hypothetical judgment itself and are independent of any particular propositions. In a sequent calculus, there are two such rules: cut, which states that if we can prove A we are justified to use A as a resource, and identity, which says that we can use a resource A to prove A.

$$\frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut} \qquad\qquad \frac{}{A \vdash A}\ \mathsf{id}$$

Under the Curry-Howard isomorphism for intuitionistic logic, propositions are related to types, proofs to programs, and proof reduction to computation. Here, linear logic propositions are related to session types, proofs to concurrent programs, and cut reduction in proofs to computation. For this correspondence, each hypothesis is labeled by a *channel* (rather than a variable). In addition, we also label the conclusion by a channel. This is because, unlike functional programming, we do not reduce a process to a value but we interact with it. For such interaction to take place in the concurrent setting, we need a channel to communicate along.

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash P :: (x : A)$$

Here, $x_1, \ldots, x_n$ and x are distinct channels, and $A_1, \ldots, A_n$ and A are their respective session types. We say that process P *provides* A along channel x and *uses* channels $x_1, \ldots, x_n$.

The rule of cut now is a form of *process composition*, connecting a client (here Q) to a provider (here P).

$$\frac{\Delta \vdash P_x :: (x : A) \quad \Delta, x{:}A \vdash Q_x :: (z : C)}{\Delta, \Delta' \vdash (x \leftarrow P_x \,;\, Q_x) :: (z : C)}\;\mathsf{cut}$$

We use syntactic forms for processes, rather than π-calculus terms, to emphasize the interpretation of proofs as programs. Because every (well-typed) process P offers a session along exactly one channel, and each channel is provided by exactly one process, we can think of channels as unique process identifiers. Under this interpretation, suggested by the intuitionistic formulation of linear logic, we can see that the cut rule *spawns* P as a new process. More precisely, the process identified by z executing $(x \leftarrow P_x \,;\, Q_x)$ creates a fresh channel a, spawns a process executing P_a that provides session A along a, and continues as Q_a. Because a is fresh, this channel will be a private channel between P_a and Q_a.

We can express this in a *substructural operational semantics* [19] which is based on *multiset rewriting* [7]. The notation is again borrowed from linear logic, but it should not be confused with the use of linear logic propositions as session types.

$$\mathsf{cut} : \mathsf{proc}_c(x \leftarrow P_x \,;\, Q_x) \multimap \{\exists a.\ \mathsf{proc}_a(P_a) \otimes \mathsf{proc}_c(Q_a)\}$$

In this formalization $\mathsf{proc}_c(P)$ is the state of a process executing program P, offering along channel c. The multiplicative conjunction $(\otimes)$ combines processes in the same state, linear implication $(\multimap)$ expresses a state transition from left to right, and the existential quantification corresponds to generation of a fresh channel. The curly braces $\{\cdots\}$ indicate a monad which essentially forces the rule above to be interpreted as a multiset rewriting rule.

The identity rule instead *forwards* between its client and the process that it uses, which must be of the same type.

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

There are several ways to describe this action operationally. A straightforward one globally identifies the channels x and y, while the forwarding process itself terminates.

$$\mathsf{id} : \mathsf{proc}_c(c \leftarrow d) \multimap \{c = d\}$$

This could be implemented in the substrate of the network or operating system. Or it could be implemented more explicitly by sending a message along c asking the client to use d for subsequent interactions. For now, we abstract over such lower level details.

Assigning process expressions to each rule of linear logic yields the following interpretation of propositions.

$A, B, C ::=$	$\mathbf{1}$	send end and terminate
$\mid$	$A \otimes B$	send channel of type A and continue as B
$\mid$	$A \oplus B$	send inl or inr and continue as A or B, respectively
$\mid$	$\tau \wedge B$	send value v of type τ and continue as B
$\mid$	$A \multimap B$	receive channel of type A
$\mid$	$A \mathbin{\&} B$	receive inl or inr and continue as A or B, respectively
$\mid$	$\tau \supset B$	receive value V of type τ and continue as B

Here, we wrote $\tau \wedge B$ as a special case of $\exists x{:}\tau.\, B$ where x does not appear in B, and $\tau \supset B$ is a special case of $\forall x{:}\tau.\, B$. The syntactic simplification is justified because in this paper we do not consider propositions that depend on terms.

Below is a summary of the process expressions, with the sending construct followed by the matching receiving construct. For the purpose of the examples we generalize the binary choice constructs $A \mathbin{\&} B$ and $A \oplus B$ to n-ary choice $\&\{lab_i : A_i\}_i$ and $\oplus\{lab_i : A_i\}_i$, respectively. We have as a special case $A \mathbin{\&} B = \&\{\mathsf{inl} : A, \mathsf{inr} : B\}$ and $A \oplus B = \oplus\{\mathsf{inl} : A, \mathsf{inr} : B\}$.

$P, Q, R ::=$	$x \leftarrow P_x \; ; Q_x$	cut (spawn)
$\mid$	$c \leftarrow d$	id (forward)
$\mid$	$\mathsf{close}\ c \mid \mathsf{wait}\ c$	$\mathbf{1}$
$\mid$	$\mathsf{send}\ c\ (y \leftarrow P_y) \; ; Q \mid x \leftarrow \mathsf{recv}\ c \; ; R_x$	$A \otimes B, A \multimap B$
$\mid$	$\mathsf{send}\ c\ d$	derived form $A \otimes B$, $A \multimap B$
$\mid$	$\mathsf{send}\ c\ M \; ; P \mid n \leftarrow \mathsf{recv}\ c \; ; Q_n$	$A \wedge B, A \supset B$
$\mid$	$c.lab \; ; P \mid \mathsf{case}\ c\ \{lab_i \to Q_i\}_i$	$\&\{lab_i : A_i\}_i, \oplus\{lab_i : A_i\}_i$

As a running example in this paper we will use variations of an implementation of polymorphic queues. We begin with the purely linear version. The interface specifies that a queue presents an external choice between enqueue and dequeue operations. When the client selects to enqueue, we input a channel of type A (to be stored in the queue), and recurse. When the client selects to dequeue, we either indicate that the queue is empty and terminate, or we indicate that there is some element in the queue, send the first element (removing it in the process), and recurse. The "recursion" here is an instance of an *equirecursive session type* [10]; some logical underpinnings are available for *coinductive types* [22]. We also use polymorphism intuitively; a formal development can be found in [4].

First, the specification of the queue interface.

$$\mathsf{queue}\, A = \&\{\mathsf{enq} : A \multimap \mathsf{queue}\, A, \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{queue}\, A\}\}$$

We implement queues with two forms of recursive processes, *empty* for the empty queue and *elem* for a process holding exactly one element. For processes with

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash P :: (x : A)$$

we write $P : \{A \leftarrow A_1, \ldots, A_n\}$ to specify its typing and $x \leftarrow P \leftarrow x_1, \ldots, x_n$ to provide its interface.

```
$empty : \{\mathsf{queue}\,A\}$

$c \leftarrow empty =$
  case $c$ of
  | enq → $x \leftarrow$ recv $c$ ;
          $e \leftarrow empty$ ;
          $c \leftarrow elem \leftarrow x, e$
  | deq → $c$.none ;
          close $c$
```

```
$elem : \{\mathsf{queue}\,A \leftarrow A, \mathsf{queue}\,A\}$

$c \leftarrow elem \leftarrow x, d =$
  case $c$ of
  | enq → $y \leftarrow$ recv $c$ ;
          $d$.enq ; send $d\ y$ ;
          $c \leftarrow elem \leftarrow x, d$
  | deq → $c$.some ;
          send $c\ x$ ;
          $c \leftarrow d$
```

From the perspective of the client, this implementation has constant time enqueue and dequeue operations. For dequeue this is obvious. For enqueue, the process at the front of the queue passes the element down the queue and is immediately available to serve another request while the element travels to the end of the queue.

3 Categorical Truth

The linear logic proposition $!A$ allows A to be used arbitrarily often in a proof—it functions as an unrestricted resource. In the intuitionistic reconstruction of linear logic [8], $!A$ internalizes a *categorical judgment*. We say that A is *valid* if it is true, and its proof does not depend on any assumptions about the truth of other propositions. Since we are working with a linear hypothetical judgment, this means that the proof of A does not depend on any resources. We further allow hypotheses Γ that are assumed to be valid (rather than merely true), and these are allowed in a proof A *valid*.

$$\Gamma \,;\, \Delta \vdash C$$

The meaning of validity is captured in the following two judgmental rules, where '$\cdot$' stands for an empty context:

$$\frac{\Gamma \,;\, \cdot \vdash A \quad (\Gamma, A) \,;\, \Delta \vdash C}{\Gamma \,;\, \Delta \vdash C}\,\mathsf{cut!} \qquad \frac{(\Gamma, A) \,;\, \Delta, A \vdash C}{(\Gamma, A) \,;\, \Delta \vdash C}\,\mathsf{copy}$$

The first, cut!, states that we are justified in assuming that A is valid if we can prove it without using any resources. The second, copy, states that we are justified in assuming a copy of the resource A if A is known to be valid. All the purely linear rules are generalized by adding an unrestricted context Γ which is propagated to all premises.

How do we think of these in terms of processes? We introduce a new form of channel, called a *shared channel* (denoted by u, w) which can be used arbitrarily often in a client, and by arbitrarily many clients. It is offered by a *persistent process*. Operationally, a persistent process offering along $w : A$ inputs a fresh linear channel c and spawns a new process P that offers A along c.

We have the following typing rules, first at the level of judgments.

$$\frac{\Gamma, u{:}A \; ; \; \Delta, x{:}A \vdash P_x :: (z{:}C)}{\Gamma, u{:}A \; ; \; \Delta \vdash (x \leftarrow \mathsf{send}\ u \; ; \; P_x) :: (z{:}C)}\ \mathsf{copy}$$

$$\frac{\Gamma \; ; \; \cdot \vdash P_y :: (y{:}A) \quad \Gamma, u{:}A \; ; \; \Delta \vdash Q_u :: (z{:}C)}{\Gamma \; ; \; \Delta \vdash (u \leftarrow !(y \leftarrow \mathsf{recv}\ u \; ; \; P_y) \; ; \; Q_u) :: (z{:}C)}\ \mathsf{cut!}$$

The copy rule has a slightly strange process expression,

$$x \leftarrow \mathsf{send}\ u \; ; \; P_x$$

It expresses that we send a *new* channel x along u. The continuation P refers to x so it can communicate along this new channel. This pattern will be common for sending fresh channels in a variety of constructs in this paper.

We see that the cut! rule incorporates two steps: creating a new shared channel u and then immediately receiving a linear channel y along u. There is no simple way to avoid this, since P in the first premise offers along a linear channel y. We will see alternatives in later sections.

In the operational semantics we write $!\mathsf{proc}_w(P)$ for a persistent process, offering along shared channel w. In the language of substructural specification, $!\mathsf{proc}_w(P)$ on the left-hand side of a rule means that it has to match a persistent proposition. We therefore do not need to repeat it on the right-hand side: it will continue to appear in the state. In this notation, the operational semantics is as follows:

$$\begin{aligned}\mathsf{copy} : \; & !\mathsf{proc}_w(y \leftarrow \mathsf{recv}\ w \; ; \; P_y) \otimes \mathsf{proc}_c(x \leftarrow \mathsf{send}\ w \; ; \; Q_x) \\ & \multimap \{\exists a.\ \mathsf{proc}_a(P_a) \otimes \mathsf{proc}_c(Q_a)\} \\[1em] \mathsf{cut!} \; : \; & \mathsf{proc}_c(u \leftarrow !(y \leftarrow \mathsf{recv}\ u \; ; \; P_y) \; ; \; Q_u) \\ & \multimap \{\exists w.\ !\mathsf{proc}_w(y \leftarrow \mathsf{recv}\ w \; ; \; P_y) \otimes \mathsf{proc}_c(Q_w)\}\end{aligned}$$

The validity judgment realized by persistent processes offering along unrestricted channels can be internalized as a proposition $!A$ with the following rules. Note that the linear context must be empty in the !R rule, since validity is a categorical judgment. Allowing dependence on linear channels would violate their linearity.

$$\frac{\Gamma \; ; \; \cdot \vdash P_y :: (y{:}A)}{\Gamma \; ; \; \cdot \vdash (u \leftarrow \mathsf{send}\ x \; ; \; !(y \leftarrow \mathsf{recv}\ u \; ; \; P_y)) :: (x{:}!A)}\ !\mathsf{R}$$

$$\frac{\Gamma, u{:}A \; ; \; \Delta \vdash Q_u :: (z{:}C)}{\Gamma \; ; \; \Delta, x{:}!A \vdash (u \leftarrow \mathsf{recv}\ x \; ; \; Q_u) :: (z{:}C)}\ !\mathsf{L}$$

Again the !R rule combines two steps: sending a new persistent channel u along x and then receiving a linear channel y along u. Operationally:

$$\begin{aligned}\mathsf{bang} : {}& \mathsf{proc}_c(u \leftarrow \mathsf{recv}\ a\ ;\ Q_u) \otimes \mathsf{proc}_a(u \leftarrow \mathsf{send}\ a\ ;\ !(y \leftarrow \mathsf{recv}\ u\ ;\ P_y)) \\ & \multimap \{\exists w.\ \mathsf{proc}_c(Q_w) \otimes !\mathsf{proc}_w(y \leftarrow \mathsf{recv}\ w\ ;\ P_y)\}\end{aligned}$$

As expected, the persistent process spawned by the bang computation rule has exactly the same form as the one spawned by cut!, because a linear cut for a proposition $!A$ becomes a persistent cut for a proposition A.

Let's analyze the two-step rule in more detail.

$$\frac{\Gamma\ ;\ \cdot \vdash P_y :: (y{:}A)}{\Gamma\ ;\ \cdot \vdash (u \leftarrow \mathsf{send}\ x\ !(y \leftarrow \mathsf{recv}\ u\ ;\ P_y)) :: (x{:}!A)}\ !\mathsf{R}$$

The judgment A *valid* (corresponding to an unrestricted hypothesis $u{:}A$) is elided on the right-hand side: we jump directly from the truth of $!A$ to the truth of A. Writing it out as an intermediate step appears entirely reasonable. We do not even mention the linear hypotheses in the intermediate step, since the validity of A depends only on assumptions of validity in Γ.

$$\frac{\dfrac{\Gamma\ ;\ \cdot \vdash P_y :: (y{:}A)}{\Gamma \vdash (y \leftarrow \mathsf{recv}\ u\ ;\ P_y) :: (u{:}A)}\ \mathsf{valid}}{\Gamma\ ;\ \cdot \vdash (u \leftarrow \mathsf{send}\ x\ ;\ !(y \leftarrow \mathsf{recv}\ u\ ;\ P_y)) :: (x{:}!A)}\ !\mathsf{R}$$

We emphasize that $!A$ is positive (in the sense of polarized logic), so it corresponds to a send, while $A\,\mathsf{valid}$ is negative as a judgment, so it correspond to a receive. In the next section we elevate this from a judgmental to a first-class logical step.

Revisiting the example, recall that if we are the client of a channel $c : \mathsf{queue}\,A$, we must use this channel. This means we have to explicitly dequeue all its elements. In fact, we have to explicitly consume each of the elements as well, since they are also linear. However, if we know that each element in the queue is in fact unrestricted, we can destroy it recursively with the following program.

```
destroy : {1 ← queue (!A)}

c ← destroy ← q =
  q.deq ;
  case q of
  | none → wait q ; close c
  | some → x ← recv q ;        % obtain element x
           u ← recv x ;        % receive shared channel u, using x
           c ← destroy ← q     % recurse, ignoring u
```

4 Adjoint Logic

Adjoint logic is based on the idea that instead of a modality like $!A$ that remains within a given language of propositions, we have two mutually dependent languages and *two* modalities going back and forth between them. For this to make sense, the operators have to satisfy certain properties that pertain to the semantics of the two languages. We have in fact three language layers, which we call *linear propositions* A_{L}, *affine propositions* A_{F}, and *unrestricted propositions* A_{U}. They are characterized by the structural properties they satisfy: linear propositions are subject to none (they must be used exactly once), affine proposition can be weakened (they can be used at most once), and unrestricted propositions can be contracted and weakened (they can be used arbitrarily often). The order of propositions in the context matters for none of them. The hierarchy of structural properties is reflected in a hierarchy of modes of truth:

$$\mathsf{U} > \mathsf{F} > \mathsf{L}$$

U is *stronger* than F in the sense that unrestricted hypotheses can be used to prove affine conclusions, but not vice versa, and similarly for the other relations. Contexts Ψ combine assumptions with all modes. We write $\geq$ for the reflexive and transitive closure of $>$ and define

$$\Psi \geq k \quad \text{if } m \geq k \text{ for every } B_m \text{ in } \Psi$$

and

$$\Psi \vdash A_k \qquad \text{presupposes } \Psi \geq k$$

We use the notation $\uparrow_k^m A_k$ for an operator going from mode k *up* to mode m, and $\downarrow_k^m A_m$ for an operator going *down* from mode m to mode k. In both cases we presuppose $m > k$.

Taking this approach we obtain the following language:

$$\begin{array}{lrcl}
\text{Modes} & m,k,r & ::= & \mathsf{U} \mid \mathsf{F} \mid \mathsf{L} \\
\text{Propositions} & A_m, B_m & ::= & \mathbf{1}_m \mid A_m \otimes_m B_m \mid A_m \oplus_m B_m \mid \tau \wedge_m B_m \\
 & & \mid & A_m \multimap_m B_m \mid A_m \,\&_m\, B_m \mid \tau \supset_m B_m \\
 & & \mid & \uparrow_k^m A_k \qquad (m > k) \\
 & & \mid & \downarrow_m^r A_r \qquad (r > m)
\end{array}$$

Because both $!A$ and A are linear propositions, the exponential $!A$ decomposes into two modalities:

$$!A = \downarrow_{\mathsf{L}}^{\mathsf{U}} \uparrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{L}}$$

Because linear and affine propositions behave essentially the same way except that affine channels need not be used, we reuse all the same syntax (both for propositions and for process expressions) at these two layers. Unrestricted propositions would behave quite differently in ways that are outside the scope of this note, so we specify that there are no unrestricted propositions besides $\uparrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{L}}$ and $\uparrow_{\mathsf{F}}^{\mathsf{U}} A_{\mathsf{F}}$.

In the following logical rules we always presuppose that the sequent in the conclusion is well-formed and add enough conditions to verify the presupposition in the premises.

$$\frac{\Psi \vdash A_k}{\Psi \vdash {\uparrow}_k^m A_k}\ {\uparrow}\mathsf{R} \qquad \frac{k \geq r \quad \Psi, A_k \vdash C_r}{\Psi, {\uparrow}_k^m A_k \vdash C_r}\ {\uparrow}\mathsf{L}$$

$$\frac{\Psi_{\geq m} \vdash A_m}{\Psi \vdash {\downarrow}_k^m A_m}\ {\downarrow}\mathsf{R} \qquad \frac{\Psi, A_m \vdash C_r}{\Psi, {\downarrow}_k^m A_m \vdash C_r}\ {\downarrow}\mathsf{L}$$

Here $\Psi_{\geq m}$ is the restriction of Ψ to propositions A_k with $k \geq m$. The rule does not apply if this would erase a linear proposition A_{L} since only affine and unrestricted propositions are subject to weakening.

The rules with no condition on the modes are invertible, while the others are not invertible. This means ${\uparrow}A$ is *negative* while ${\downarrow}A$ is *positive* (in the terminology of polarized logic [15]). We already noted that processes offering a negative type receive, while processes offering a positive type send. But what do we send or receive? Thinking of channels as intrinsically linear, affine, or shared suggests that we should send and receive fresh channels of different modes. Following this reasoning we obtain:

$$\frac{\Psi \vdash P_{x_k} :: (x_k{:}A_k)}{\Psi \vdash (x_k \leftarrow \mathsf{recv}\, x_m \ ; P_{x_k}) :: (x_m{:}{\uparrow}_k^m A_k)}\ {\uparrow}\mathsf{R}$$

$$\frac{k \geq r \quad \Psi, x_k{:}A_k \vdash Q_{x_k} :: (z_r{:}C_r)}{\Psi, x_m{:}{\uparrow}_k^m A_k \vdash (x_k \leftarrow \mathsf{send}\ x_m \ ; Q_{x_k}) :: (z_r{:}C_r)}\ {\uparrow}\mathsf{L}$$

For clarity, we annotate each channel with its mode, although it may not be strictly necessary. Operationally:

$$\begin{aligned} \mathsf{up}_k^m : {} & \mathsf{proc}_{a_r}(x_k \leftarrow \mathsf{send}\, c_m \ ; Q_{x_k}) \otimes \mathsf{proc}_{c_m}(y_k \leftarrow \mathsf{recv}\ c_m \ ; P_{y_k}) \\ & \multimap \{\exists c_k.\ \mathsf{proc}_{a_r}(Q_{c_k}) \otimes \mathsf{proc}_{c_k}(P_{c_k})\} \end{aligned}$$

And for the other modality:

$$\frac{\Psi \geq m \quad \Psi \vdash Q_{x_m} :: (x_m{:}A_m)}{\Psi \vdash (x_m \leftarrow \mathsf{send}\ x_k \ ; Q_{x_m}) :: (x_k{:}{\downarrow}_k^m A_m)}\ {\downarrow}\mathsf{R}$$

$$\frac{\Psi, x_m{:}A_m \vdash P_{x_m} :: (z_r{:}C_r)}{\Psi, x_k{:}{\downarrow}_k^m A_m \vdash (x_m \leftarrow \mathsf{recv}\ x_k \ ; P_{x_m}) :: (z_r{:}C_r)}\ {\downarrow}\mathsf{L}$$

Operationally:

$$\begin{aligned} \mathsf{down}_k^m : {} & \mathsf{proc}_{a_r}(y_m \leftarrow \mathsf{recv}\ c_k \ ; P_{y_m}) \otimes \mathsf{proc}_{c_k}(x_m \leftarrow \mathsf{send}\ c_k \ ; Q_{x_m}) \\ & \multimap \{\exists c_m.\ \mathsf{proc}_{a_r}(P_{c_m}) \otimes \mathsf{proc}_{c_m}(Q_{c_m})\} \end{aligned}$$

Since processes offering along unrestricted channels are persistent, we use here the (admittedly dangerous) notational convention that all processes offering along unrestricted channels c_{U} are implicitly marked persistent. In particular, we should read $\mathsf{up}^{\mathsf{U}}_k$ and $\mathsf{down}^{\mathsf{U}}_k$ as

$$\begin{array}{ll}\mathsf{up}^{\mathsf{U}}_k & : \mathsf{proc}_{a_r}(x_k \leftarrow \mathsf{send}\, c_{\mathsf{U}} \;;\; Q_{x_k}) \otimes !\mathsf{proc}_{c_{\mathsf{U}}}(x_k \leftarrow \mathsf{recv}\; c_{\mathsf{U}} \;;\; P_{x_k}) \\ & \quad \multimap \{\exists c_k.\, \mathsf{proc}_{a_r}(Q_{c_k}) \otimes \mathsf{proc}_{c_k}(P_{c_k})\}\end{array}$$

$$\begin{array}{ll}\mathsf{down}^{\mathsf{U}}_k & : \mathsf{proc}_{a_r}(x_{\mathsf{U}} \leftarrow \mathsf{recv}\; c_k \;;\; P_{x_{\mathsf{U}}}) \otimes \mathsf{proc}_{c_k}(x_{\mathsf{U}} \leftarrow \mathsf{send}\; c_k \;;\; Q_{x_{\mathsf{U}}}) \\ & \quad \multimap \{\exists c_{\mathsf{U}}.\, \mathsf{proc}_{a_r}(P_{c_{\mathsf{U}}}) \otimes !\mathsf{proc}_{c_{\mathsf{U}}}(Q_{c_{\mathsf{U}}})\}\end{array}$$

At this point we have achieved that every logical connective, including the up and down modalities, correspond to exactly one matching send and receive action. Moreover, as we can check, the compound rules for $!A$ decompose into individual steps.

Returning to our example, we can now specify that our queue is supposed to be affine, that is, that we can decide to ignore it. We annotate defined types and type variables with their mode (U, F, or L), but we overload the logical connectives since their meanings, when defined, are consistent. The elements of an affine queue should also be affine. If we make them linear, as in

$$\begin{array}{ll}\mathsf{queue}_{\mathsf{F}}\, A_{\mathsf{L}} = \&\{ & \mathsf{enq} : \uparrow^{\mathsf{F}}_{\mathsf{L}} A_{\mathsf{L}} \multimap \mathsf{queue}_{\mathsf{F}}\, A_{\mathsf{L}}, \\ & \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : \uparrow^{\mathsf{F}}_{\mathsf{L}} A_{\mathsf{L}} \otimes \mathsf{queue}_{\mathsf{F}}\, A_{\mathsf{L}}\}\ \}\end{array}$$

then we could never use $x : \uparrow^{\mathsf{F}}_{\mathsf{L}} A_{\mathsf{L}}$ in a process offering an affine service (rule $\uparrow L$) since $\mathsf{L} \not\geq \mathsf{F}$. So instead we should define an affine queue as

$$\begin{array}{ll}\mathsf{queue}_{\mathsf{F}}\, A_{\mathsf{F}} = \&\{ & \mathsf{enq} : A_{\mathsf{F}} \multimap \mathsf{queue}_{\mathsf{F}}\, A_{\mathsf{F}}, \\ & \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A_{\mathsf{F}} \otimes \mathsf{queue}_{\mathsf{F}}\, A_{\mathsf{F}}\}\ \}\end{array}$$

so that all types in the definition (including A_{F}) are affine. Now we no longer need to explicitly destroy a queue, we can just abandon it and the runtime system will deallocate it by a form of garbage collection.

If we want to enforce a linear discipline, destroying a queue with linear elements will have to rely on a consumer for the elements of the queue. This consumer must be unrestricted because it is used for each element. Channels are linear by default, so in the example we only annotate affine and unrestricted channels with their mode.

```
$destroy : \{\mathbf{1} \leftarrow \mathsf{queue}_{\mathsf{L}}\, A_{\mathsf{L}}, \uparrow^{\mathsf{U}}_{\mathsf{L}}(A_{\mathsf{L}} \multimap \mathbf{1})\}$

$c \leftarrow destroy \leftarrow q, u_{\mathsf{U}} =$
    $q$.deq ;
    case $q$ of
    | none → wait $q$ ; close $c$
    | some → $x$ ← recv $q$ ;
              $d$ ← send $u_{\mathsf{U}}$ ;            % obtain instance $d$ of $u_{\mathsf{U}}$
              send $d$ $x$ ; wait $d$ ;        % use $d$ to consume $x$
              $c \leftarrow destroy \leftarrow q, u_{\mathsf{U}}$         % recurse, reusing $u_{\mathsf{U}}$
```

5 Polarized Logic

We now take a step in a different direction by introducing asynchronous communication, postponing discussion of the modalities for now. In asynchronous communication each linear channel contains a message queue [11], which can be related directly to the proof system via continuation channels [9]. Sending adds to the queue on one end and receiving takes from the other. Because session-based communication goes in both directions, the queue switches direction at certain times. Moreover, the queue must maintain some information on the direction of the queue so that a process that performs a send followed by a receive does not incorrectly read its own message. Fortunately, session typing guarantees that there is no send/receive mismatch.

A simple way to maintain the direction of a queue is to set a flag when enqueuing a message. We write just q when the direction of q does not matter, and $\overleftarrow{q}$ and $\overrightarrow{q}$ for the two directions. Our convention is that $\overleftarrow{q}$ corresponds to messages from a provider to its client, and $\overrightarrow{q}$ for messages from a client to the provider. The reasons for this convention is that in $\mathsf{proc}_c(P)$, the channel c is to the left of P, which is in turn derived from $c \leftarrow P$ for a process expression P offering a service along c.

We have a predicate $\mathsf{queue}(c, q, d)$ for a queue q connecting a process Q using c with one providing d. Here are two example rules for sending and receiving data values.

$$\begin{aligned}
\mathsf{and\_s} : {} & \mathsf{queue}(c, q, d) \otimes \mathsf{proc}_d(\mathsf{send}\ d\ v\ ;\ P) \\
& \multimap \{\mathsf{queue}(c, \overleftarrow{q \cdot v}, d) \otimes \mathsf{proc}_d(P)\} \\
\mathsf{and\_r} : {} & \mathsf{proc}_c(x \leftarrow \mathsf{recv}\ c\ ;\ Q_x) \otimes \mathsf{queue}(c, \overleftarrow{v \cdot q}, d) \\
& \multimap \{\mathsf{proc}_c(Q_v) \otimes \mathsf{queue}(c, q, d)\}
\end{aligned}$$

We see some difficulty in the second rule, where the direction of q is unclear. It should be $\overleftarrow{q}$ unless q is empty, it which case it is unknown. This ambiguity is also present in forwarding.

$$\begin{aligned}
\mathsf{fwd} : {} & \mathsf{queue}(c, p, d) \otimes \mathsf{proc}_d(d \leftarrow e) \otimes \mathsf{queue}(d, q, e) \\
& \multimap \{\mathsf{queue}(c, p \cdot q, e)\}
\end{aligned}$$

We won't go into detail why there are some difficulties implementing this, but we see that there are multiple possibilities for p and q pointing left, right, or being empty.

Next we note that the *polarity* of each connective determines the direction of communication. From the perspective of the service provider, if we have $P ::$ $(x{:}A)$ for a *positive* A then the action of P along x will be a send, if A is *negative* it will be receive. Intuitively this is because the right rules for negative connectives are invertible and therefore carry no information: any information has to come from the outside. Conversely, the right rules for positive connectives involve some choice and can therefore communicate the essence of that information. We can make this explicit by *polarizing* the logic, dividing the propositions into positive and negative propositions with explicit *shift* operators connecting them. Omitting other modalities, the syntax of polarized logic is:

$$\begin{array}{lrll}
\text{Positive propositions} & A^+, B^+ ::= & \mathbf{1} & \text{send } \mathsf{end} \text{ and terminate} \\
 & \mid & A^+ \otimes B^+ & \text{send channel of type } A^+ \\
 & \mid & A^+ \oplus B^+ & \text{send } \mathsf{inl} \text{ or } \mathsf{inr} \\
 & \mid & \tau \wedge B^+ & \text{send value of type } \tau \\
 & \mid & {\downarrow}A^- & \text{send } \mathsf{shift}\text{, then receive} \\
\text{Negative propositions} & A^-, B^- ::= & A^+ \multimap B^- & \text{receive channel of type } A^+ \\
 & \mid & A^- \mathbin{\&} B^- & \text{receive } \mathsf{inl} \text{ or } \mathsf{inr} \\
 & \mid & \tau \supset B^- & \text{receive value of type } \tau \\
 & \mid & {\uparrow}A^+ & \text{receive } \mathsf{shift}\text{, then send}
\end{array}$$

Note that a process that sends along a channel will continue to do so until it sends a shift and then it starts receiving. Conversely, a process that receives continues to do so until it receives a shift after which it starts sending. The new constructs are:

$$\begin{array}{rll}
P, Q, R ::= & \mathsf{send}\ c\ \mathsf{shift}\ ;\ P & \text{send } \mathsf{shift}\text{, then receive along } c \text{ in } P \\
\mid & \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q & \text{receive } \mathsf{shift}\text{, then send along } c \text{ in } Q
\end{array}$$

We have already annotated the shifts with their expected operational semantics. Queues now always have a definite direction and there can be no further messages following a shift. We write m for messages other than shift, such as data values, labels, and channels and treat $\cdot$ as an associative concatenation operator with the empty queue as its unit.

$$\begin{array}{lll}
\text{Queue filled by provider} & \overleftarrow{q} ::= & \overleftarrow{\cdot} \mid \overleftarrow{m \cdot q} \mid \overleftarrow{\mathsf{end}} \mid \overleftarrow{\mathsf{shift}} \\
\text{Queue filled by client} & \overrightarrow{q} ::= & \overrightarrow{\mathsf{shift}} \mid \overrightarrow{q \cdot m} \mid \overrightarrow{\cdot}
\end{array}$$

In the polarized setting, we just need to initialize the direction correctly when a new channel is created, after which the direction is maintained correctly throughout. When receiving, the direction needs to be checked. When sending, the direction will always be correct by invariant.

$$\begin{array}{l}
\mathsf{and\_s} : \mathsf{queue}(c, \overleftarrow{q}, d) \otimes \mathsf{proc}_d(\mathsf{send}\ d\ v\ ;\ P) \\
\qquad \multimap \{\mathsf{queue}(c, \overleftarrow{q \cdot v}, d) \otimes \mathsf{proc}_d(P)\} \\
\mathsf{and\_r} : \mathsf{proc}_c(n \leftarrow \mathsf{recv}\ c\ ;\ Q_n) \otimes \mathsf{queue}(c, \overleftarrow{v \cdot q}, d) \\
\qquad \multimap \{\mathsf{proc}_c(Q_v) \otimes \mathsf{queue}(c, \overleftarrow{q}, d)\}
\end{array}$$

The shift reverses direction when received.

$$\begin{array}{l}
\mathsf{shift\_s} : \mathsf{queue}(c, \overleftarrow{q}, d) \otimes \mathsf{proc}_d(\mathsf{send}\ d\ \mathsf{shift}\ ;\ P) \\
\qquad \multimap \{\mathsf{queue}(c, \overleftarrow{q \cdot \mathsf{shift}}, d) \otimes \mathsf{proc}_d(P)\} \\
\mathsf{shift\_r} : \mathsf{proc}_a(\mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q) \otimes \mathsf{queue}(c, \overleftarrow{\mathsf{shift}}, d) \\
\qquad \multimap \{\mathsf{proc}_a(Q) \otimes \mathsf{queue}(c, \overrightarrow{\cdot}, d)\}
\end{array}$$

There are symmetric rules for $\overrightarrow{\mathsf{shift}}$, which we elide here.

In our running example, the natural polarization would interpret *queue* as a negative type, since it offers an external choice. We have to switch to positive when we send a response to the dequeue request, and then switch again before we recurse. The type parameter A is most naturally positive, since both occurrences in the type are in fact positive.

$$
\begin{aligned}
\mathsf{queue}^-\ A^+ = \&\{\ & \mathsf{enq} : A^+ \multimap \mathsf{queue}^-\ A^+, \\
& \mathsf{deq} : \uparrow \oplus \{\mathsf{none} : \mathbf{1}, \mathsf{some} : A^+ \otimes \downarrow \mathsf{queue}^-\ A^+\}\ \}
\end{aligned}
$$

The code requires some minimal changes: we have to insert three shift operators.

```
empty : {queue⁻ A⁺}

c ← empty =
  case c of
  | enq → x ← recv c ;
          e ← empty ;
          c ← elem ← x, e
  | deq → shift ← recv c ;
          c.none ;
          close c
```

```
elem : {queue⁻ A⁺ ← A⁺, queue⁻ A⁺}

c ← elem ← x, d =
  case c of
  | enq → y ← recv c ;
          d.enq ; send d y ;
          c ← elem ← x, d
  | deq → shift ← recv c ;    % shift c to send
          c.some ; send c x ;
          send c shift ;      % shift c to recv
          c ← d
```

6 Recovering Synchronous Communication

We obtain maximally asynchronous communication by inserting shifts in a bare (unpolarized) session type only where necessary.

$$
\begin{array}{lll}
(1)^+ & = 1 & \\
(A \otimes B)^+ & = (A)^+ \otimes (B)^+ & \\
(A \oplus B)^+ & = (A)^+ \oplus (B)^+ & \\
(\tau \wedge B)^+ & = \tau \wedge (B)^+ & \\
(A)^+ & = {\downarrow}(A)^- & \text{for other propositions } A \\
(A \multimap B)^- & = (A)^+ \multimap (B)^- & \\
(A \mathbin{\&} B)^- & = (A)^- \mathbin{\&} (B)^- & \\
(\tau \supset B)^- & = \tau \supset (B)^- & \\
(A)^- & = {\uparrow}(A)^+ & \text{for other propositions } A
\end{array}
$$

As a provider, we can send asynchronously at a positive session type until we shift explicitly to perform an input because we are now at a negative proposition. A client behaves dually.

In order to simulate synchronous communication, we insert additional shifts to prevent two consecutive send operations on the same channel. Here, the down shift after a send switches to a mode where we wait for an acknowledgment, which is implicit in the next receive. If this is another shift, it acts as a pure acknowledgment, otherwise it is already the next message.

$$\begin{array}{lll}
(1)^+ & = \mathbf{1} & \\
(A \otimes B)^+ & = (A)^+ \otimes {\downarrow}(B)^- & \\
(A \oplus B)^+ & = {\downarrow}(A)^- \oplus {\downarrow}(B)^- & \\
(\tau \wedge B)^+ & = \tau \wedge {\downarrow}(B)^- & \\
(A)^+ & = {\downarrow}(A)^- & \text{for other propositions } A \\
(A \multimap B)^- & = (A)^+ \multimap {\uparrow}(B)^+ & \\
(A \mathbin{\&} B)^- & = {\uparrow}(A)^+ \mathbin{\&} {\uparrow}(B)^+ & \\
(\tau \supset B)^- & = \tau \supset {\uparrow}(B)^+ & \\
(A)^- & = {\uparrow}(A)^+ & \text{for other propositions } A
\end{array}$$

If we want to bound the size of message queues then we can insert shift in session types which would otherwise allow an unbounded number of consecutive sends.

In our running example, a client of a queue can perform an unbounded number of enqueue operations in the asynchronous operational semantics before the queue implementation must react. This is because this portion of the queue type is entirely negative. In order to force synchronization, we can change the type of the enqueue operation before we recurse.

$$\begin{array}{l}
\mathsf{queue}^-\, A^+ = \&\{\ \mathsf{enq} : A^+ \multimap {\uparrow}{\downarrow}\mathsf{queue}^-\, A^+, \\
\qquad\qquad\qquad\quad \mathsf{deq} : {\uparrow}\oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A^+ \otimes {\downarrow}\mathsf{queue}^-\, A^+\}\ \}
\end{array}$$

Now the maximal size of the queue will be 3 in one direction ($\mathsf{shift} \cdot x \cdot \mathsf{enq}$) and also 3 in the other direction ($\mathsf{some} \cdot x \cdot \mathsf{shift}$). In a slightly different language, boundedness calculations for queues in asynchronous session-typed communication can be found in [11], so we do not repeat a more formal analysis here.

7 Synthesis in Polarized Adjoint Logic

Now we are ready to combine the ideas from adjoint logic in Sec. 4 with polarization in Sec. 5. Amazingly, they are fully consistent. The two differences to the polarized presentation are that (a) the modalities go between positive and negative propositions (already anticipated by the fact that $\downarrow$ is positive and $\uparrow$ is negative), and (b) the modalities ${\downarrow}^m_k A$ and ${\uparrow}^m_k$ allow $m \geq k$ rather than presupposing $m > k$ as before. We no longer index the connectives, overloading their meaning at the different layers.

$$\begin{array}{llll}
\text{Pos. propositions } A^+_m, B^+_m & ::= & \mathbf{1} & \text{send } \mathsf{end} \text{ and terminate} \\
& \mid & A^+_m \otimes B^+_m & \text{send channel of type } A^+_m \\
& \mid & A^+_m \oplus B^+_m & \text{send } \mathsf{inl} \text{ or } \mathsf{inr} \\
& \mid & \tau \wedge B^+_m & \text{send value of type } \tau \\
& \mid & {\downarrow}^r_m A^-_r & (r \geq m), \text{send } \mathsf{shift}\text{, then receive} \\
\text{Neg. propositions } A^-_m, B^-_m & ::= & A^+_m \multimap B^-_m & \text{receive channel of type } A^+_m \\
& \mid & A^-_m \mathbin{\&} B^-_m & \text{receive } \mathsf{inl} \text{ or } \mathsf{inr} \\
& \mid & \tau \supset B^-_m & \text{receive value of type } \tau \\
& \mid & {\uparrow}^m_k A^+_k & (m \geq k), \text{receive } \mathsf{shift}\text{, then send}
\end{array}$$

A shift staying at the same level just changes the polarity but is otherwise not subject to any restrictions. We can see this from the rules, now annotated with a polarity: if $m = k$ in $\uparrow$L, then $k \geq r$ by presupposition since $(\Psi, \uparrow_k^m A_k^+) \geq r$. Similarly, in $\downarrow$R, $\Psi \geq m$ by presupposition if $m = k$.

$$\frac{\Psi \vdash A_k^+}{\Psi \vdash \uparrow_k^m A_k^+}\ \uparrow\mathsf{R} \qquad \frac{k \geq r \quad \Psi, A_k^+ \vdash C_r}{\Psi, \uparrow_k^m A_k^+ \vdash C_r}\ \uparrow\mathsf{L}$$

$$\frac{\Psi_{\geq m} \vdash A_m^-}{\Psi \vdash \downarrow_k^m A_m^-}\ \downarrow\mathsf{R} \qquad \frac{\Psi, A_m^- \vdash C_r}{\Psi, \downarrow_k^m A_m^- \vdash C_r}\ \downarrow\mathsf{L}$$

Adding process expressions in a straightforward manner generalizes the shift to carry a fresh channel because there may now be a change in modes associated with the shift. We have the following new syntax

$$\begin{array}{rll} P, Q ::= & \mathsf{shift}\ x_k \leftarrow \mathsf{send}\ c_m\ ;\ P_{x_k} & \text{send fresh } \mathsf{shift}\ x_k \text{, then recv. along } x_k \text{ in } P \\ \mid & \mathsf{shift}\ x_k \leftarrow \mathsf{recv}\ c_m\ ;\ Q_{x_k} & \text{receive } \mathsf{shift}\ x_k \text{, then send along } x_k \text{ in } Q \end{array}$$

and the modified rules

$$\frac{\Psi \vdash P_{x_k} :: (x_k{:}A_k^+)}{\Psi \vdash (\mathsf{shift}\ x_k \leftarrow \mathsf{recv}\ x_m\ ;\ P_{x_k}) :: (x_m{:}\uparrow_k^m A_k^+)}\ \uparrow\mathsf{R}$$

$$\frac{k \geq r \quad \Psi, x_k{:}A_k^+ \vdash Q_{x_k} :: (z_r{:}C_r)}{\Psi, x_m{:}\uparrow_k^m A_k^+ \vdash (\mathsf{shift}\ x_k \leftarrow \mathsf{send}\ x_m\ ;\ Q_{x_k}) :: (z_r{:}C_r)}\ \uparrow\mathsf{L}$$

$$\frac{\Psi_{\geq m} \vdash Q_{x_m} :: (x_m{:}A_m^-)}{\Psi \vdash (\mathsf{shift}\ x_m \leftarrow \mathsf{send}\ x_k\ ;\ Q_{x_m}) :: (x_k{:}\downarrow_k^m A_m^-)}\ \downarrow\mathsf{R}$$

$$\frac{\Psi, x_m{:}A_m^- \vdash P_{x_m} :: (z_r{:}C_r)}{\Psi, x_k{:}\downarrow_k^m A_m^- \vdash (\mathsf{shift}\ x_m \leftarrow \mathsf{recv}\ x_k\ ;\ P_{x_m}) :: (z_r{:}C_r)}\ \downarrow\mathsf{L}$$

Operationally:

$$\begin{array}{ll} \mathsf{up}_k^m\mathsf{\_s} & : \mathsf{proc}_{a_r}(\mathsf{shift}\ x_k \leftarrow \mathsf{send}\ c_m\ ;\ Q_{x_k}) \otimes \mathsf{queue}(c_m, \overrightarrow{q}, d_m) \\ & \quad \multimap \{\exists c_k.\ \exists d_k.\ \mathsf{proc}_{a_r}(Q_{c_k}) \otimes \mathsf{queue}(c_k, \overrightarrow{\mathsf{shift}\ d_k \cdot q}, d_m)\} \\ \mathsf{up}_k^m\mathsf{\_r} & : \mathsf{queue}(c_k, \overrightarrow{\mathsf{shift}\ d_k}, d_m) \otimes \mathsf{proc}_{d_m}(\mathsf{shift}\ x_k \leftarrow \mathsf{recv}\ d_m\ ;\ P_{x_k}) \\ & \quad \multimap \{\mathsf{queue}(c_k, \overleftarrow{\cdot}, d_k) \otimes \mathsf{proc}_{d_k}(P_{d_k})\} \\ & \\ \mathsf{down}_k^m\mathsf{\_s} & : \mathsf{queue}(c_k, \overleftarrow{q}, d_k) \otimes \mathsf{proc}_{d_k}(\mathsf{shift}\ x_m \leftarrow \mathsf{send}\ d_k\ ;\ Q_{x_m}) \\ & \quad \multimap \{\exists c_m.\ \exists d_m.\ \mathsf{queue}(c_k, \overleftarrow{q \cdot c_m}, d_m) \otimes \mathsf{proc}_{d_m}(Q_{d_m})\} \\ \mathsf{down}_k^m\mathsf{\_r} & : \mathsf{proc}_{a_r}(\mathsf{shift}\ x_m \leftarrow \mathsf{recv}\ c_k\ ;\ P_{x_m}) \otimes \mathsf{queue}(c_k, \overleftarrow{\mathsf{shift}\ c_m}, d_m) \\ & \quad \multimap \{\mathsf{proc}_{a_r}(P_{c_m}) \otimes \mathsf{queue}(c_m, \overrightarrow{\cdot}, d_m)\} \end{array}$$

As pointed out in Sec. 4, we have to assume that processes that offer along an unrestricted channel c_{U} are persistent. Also, this formulation introduces a new channel even when $m = k$, a slight redundancy best avoided in the syntax and semantics of a real implementation. Even when going between linear and affine channels, creating new channels might be avoided in favor of just changing some channel property.

Returning to forwarding, the earlier agnostic formulation will work more elegantly, since both queues to be appended are guaranteed to go into the same direction.

$$\begin{aligned}\mathsf{fwd} : \mathsf{queue}(c, p, d) \otimes \mathsf{proc}_d(d \leftarrow e) \otimes \mathsf{queue}(d, q, e)\\ \multimap \{\mathsf{queue}(c, p \cdot q, e)\}\end{aligned}$$

If implementation or other considerations suggest forwarding as an explicit message, we can also implement this, taking advantage of the direction information that is always available. Here we write $x \leftarrow \mathsf{recv}\ c$ as a generic receive operation along channel c, which is turned into a receive along the forwarded channel e.

$$\begin{aligned}\mathsf{fwd\_s} : \mathsf{queue}(c, \overleftarrow{p}, d) \otimes \mathsf{proc}_d(d \leftarrow e)\\ \multimap \{\mathsf{queue}(c, \overleftarrow{p \cdot \mathsf{fwd}}, e)\}\end{aligned}$$

$$\begin{aligned}\mathsf{fwd\_r} : \mathsf{proc}_a(x \leftarrow \mathsf{recv}\ c \ ; P_x) \otimes \mathsf{queue}(c, \overleftarrow{\mathsf{fwd}}, e)\\ \multimap \{\mathsf{proc}_a(x \leftarrow \mathsf{recv}\ e \ ; P_x)\}\end{aligned}$$

We elide the symmetric version of the rules pointing to the right. The reason we would forward in the direction of the current communication is so that send remains fully asynchronous and does not have to check if a forwarding message may be present on the channel.

Once again rewriting the linear version of the example, forcing synchronization.

$$\begin{aligned}\mathsf{queue}^-\ A^+ = \&\{\ & \mathsf{enq} : A^+ \multimap {\uparrow}{\downarrow}\mathsf{queue}^-\ A^+,\\ & \mathsf{deq} : {\uparrow}\oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A^+ \otimes {\downarrow}\mathsf{queue}^-\ A^+\}\ \}\end{aligned}$$

```
empty : {queue⁻ A⁺}

c ← empty =
  case c of
  | enq → x ← recv c ;
          shift c ← recv c
          shift c ← send c
          e ← empty ;
          c ← elem ← x, e
  | deq → shift c ← recv c
          c.none ;
          close c
```

```
elem : {queue⁻ A⁺ ← A⁺, queue⁻ A⁺}

c ← elem ← x, d =
  case c of
  | enq → y ← recv c ;
          shift c ← recv c ;    % shift to send
          shift c ← send c ;    % send ack
          d.enq ; send d y ;
          shift d ← send d ;    % shift to recv
          shift d ← recv d ;    % recv ack
          c ← elem ← x, d
  | deq → shift c ← recv c ;    % shift to send
          c.some ; send c x ;
          shift c ← send c ;    % shift to recv
          c ← d
```

And destroying a linear queue with affine elements:

```
$destroy : \{\mathbf{1} \leftarrow \mathsf{queue}\,(\downarrow^{\mathsf{F}}_{\mathsf{L}} A_{\mathsf{F}})\}$

$c \leftarrow destroy \leftarrow q =$
  $q$.deq ;
  shift $q \leftarrow$ send $q$ ;           % shift to recv
  case $q$ of
  | none $\rightarrow$ wait $q$ ; close $c$
  | some $\rightarrow$ $x \leftarrow$ recv $q$ ;          % obtain element $x$
             shift $a_{\mathsf{F}} \leftarrow$ recv $x$ ;   % obtain affine $a_{\mathsf{F}}$, consuming $x$
             shift $q \leftarrow$ send $q$ ;    % shift to recv
             $c \leftarrow destroy \leftarrow q$      % recurse, ignoring $a_{\mathsf{F}}$
```

8 Sequent Calculus for Polarized Adjoint Logic

We summarize the sequent calculus rules for polarized adjoint logic in Fig. 1, omitting the uninteresting rules for existential and universal quantification. However, we have added in atomic propositions p_m^+ and p_m^- (corresponding to session type variables) and removed the stipulation that the only unrestricted propositions are $\uparrow_m^{\mathsf{U}} A_m^+$, thereby making our theorem slightly more general at the expense of a nonstandard notation for intuitionistic connectives such as $A_{\mathsf{U}} \multimap_{\mathsf{U}} B_{\mathsf{U}}$ for $A \supset B$.

We have the following theorem.

Theorem 1.

1. *Cut is admissible in the system without cut.*
2. *Identity is admissible for arbitrary propositions in the system with the identity restricted to atomic propositions and without cut.*

Proof. The admissibility of cut follows by a nested structural induction, first on the cut formula A, second simultaneously on the proofs of the left and right premise. We liberally use a lemma which states that we can weaken a proof with affine and unrestricted hypotheses without changing its structure and we exploit the transitivity of $\geq$. See [8, 18] for analogous proofs.

The admissibility of identity at A follows by a simple structural induction on the proposition A, exploiting the reflexivity of $\geq$ in one critical case. □

A simple corollary is *cut elimination*, stating that every provable sequent has a cut-free proof. Cut elimination of the logic is the central reason why the session-typed processes assigned to these rules satisfy the by now expected properties of *session fidelity* (processes are guaranteed to follow the behavior prescribed by the session type) and *global progress* (a closed process network of type $c_0 : \mathbf{1}$ can either take a step will send end along c_0). In addition, we also have *productivity* (processes will eventually perform the action prescribed by the session type) and *termination* if recursive processes are appropriately restricted. The proofs of these properties closely follow those in the literature for related systems [6, 22, 20], so we do not formally state or prove them here.

$$
\begin{array}{lll}
m, k, r & ::= \mathsf{U} \mid \mathsf{F} \mid \mathsf{L} \quad \text{with } \mathsf{U} > \mathsf{F} > \mathsf{L} & \\
A_m^+, B_m^+ & ::= p_m^+ \mid \mathbf{1}_m \mid A_m^+ \otimes_m B_m^+ \mid A_m^+ \oplus_m B_m^+ \mid {\downarrow}_m^r A_r^- & (r \geq m) \\
A_m^-, B_m^- & ::= p_m^- \mid A_m^+ \multimap_m B_m^- \mid A_m^- \mathbin{\&}_m B_m^- \mid {\uparrow}_k^m A_k^+ & (m \geq k) \\
A_m, B_m, C_m & ::= A_m^+ \mid A_m^- &
\end{array}
$$

$$
\frac{\Psi \geq \mathsf{F}}{\Psi, A_m \vdash A_m}\ \mathsf{id}
\qquad\qquad
\frac{\Psi \geq m \geq r \quad \Psi \vdash A_m \quad \Psi', A_m \vdash C_r}{\Psi, \Psi' \vdash C_r}\ \mathsf{cut}
$$

$$
\frac{\Psi \vdash A_k^+}{\Psi \vdash {\uparrow}_k^m A_k^+}\ {\uparrow}\mathsf{R}
\qquad\qquad
\frac{k \geq r \quad \Psi, A_k^+ \vdash C_r}{\Psi, {\uparrow}_k^m A_k^+ \vdash C_r}\ {\uparrow}\mathsf{L}
$$

$$
\frac{\Psi_{\geq m} \vdash A_m^-}{\Psi \vdash {\downarrow}_k^m A_m^-}\ {\downarrow}\mathsf{R}
\qquad\qquad
\frac{\Psi, A_m^- \vdash C_r}{\Psi, {\downarrow}_k^m A_m^- \vdash C_r}\ {\downarrow}\mathsf{L}
$$

$$
\frac{\Psi \geq \mathsf{F}}{\Psi \vdash \mathbf{1}_m}\ \mathbf{1}R
\qquad\qquad
\frac{\Psi \vdash C_r}{\Psi, \mathbf{1}_m \vdash C_r}\ \mathbf{1}L
$$

$$
\frac{\Psi \vdash A_m^+ \quad \Psi' \vdash B_m^+}{\Psi, \Psi' \vdash A_m^+ \otimes_m B_m^+}\ \otimes R
\qquad\qquad
\frac{\Psi, A_m^+, B_m^+ \vdash C_r}{\Psi, A_m^+ \otimes_m B_m^+ \vdash C_r}\ \otimes L
$$

$$
\frac{\Psi, A_m^+ \vdash B_m^-}{\Psi \vdash A_m^+ \multimap_m B_m^-}\ {\multimap} R
\qquad\qquad
\frac{\Psi \geq m \quad \Psi \vdash A_m^+ \quad \Psi', B_m^- \vdash C_r}{\Psi, \Psi', A_m^+ \multimap_m B_m^- \vdash C_r}\ {\multimap} L
$$

$$
\frac{\Psi \vdash A_m^- \quad \Psi \vdash B_m^-}{\Psi \vdash A_m^- \mathbin{\&}_m B_m^-}\ \& R
\qquad
\frac{\Psi, A_m^- \vdash C_r}{\Psi, A_m^- \mathbin{\&}_m B_m^- \vdash C_r}\ \& L_1
\qquad
\frac{\Psi, B_m^- \vdash C_r}{\Psi, A_m^- \mathbin{\&}_m B_m^- \vdash C_r}\ \& L_2
$$

$$
\frac{\Psi \vdash A_m^+}{\Psi \vdash A_m^+ \oplus_m B_m^+}\ \oplus R_1
\qquad
\frac{\Psi \vdash B_m^+}{\Psi \vdash A_m^+ \oplus_m B_m^+}\ \oplus R_2
\qquad
\frac{\Psi, A_m^+ \vdash C_r \quad \Psi, B_m^+ \vdash C_r}{\Psi, A_m^+ \oplus_m B_m^+ \vdash C_r}\ \oplus L
$$

All judgments $\Psi \vdash A_m$ presuppose $\Psi \geq m$.
Ψ, Ψ' allows contraction of unrestricted A_{U} shared between Ψ and Ψ'

Fig. 1. Polarized Adjoint Logic

9 Conclusion

We have developed a language which uniformly integrates linear, affine, and unrestricted types, allowing the programmer to vary the degree of precision with which resources are managed. At the same time, the programmer has fine-grained control over which communications are synchronous or asynchronous, and these decisions are reflected in the type in a logically motivated manner.

On the pragmatic side, we should decide to what extent the constructs here are exposed to the programmer or inferred during type checking, and develop a concise and intuitive concrete syntax for those that are explicitly available in types and process expressions.

Finally, our language is polarized, but deductions are not focused [1]. This is perhaps somewhat unexpected since the two are closely connected and historically tied to each other. It suggests that some further benefits from proof-theoretic concepts are still to be discovered, continuing the current line of investigation into the foundation of session-typed concurrency.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1423168 and by the Department of Energy under Grant No. SNL 1488651. The authors would like to acknowledge helpful discussions with Stephanie Balzer, Luís Caires, Henry DeYoung, Hannah Gommerstadt, Elsa Gunter, Limin Jia, Rokhini Prabhu, Bernardo Toninho, and Max Willsey.

References

1. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation 2(3), 197–347 (1992)
2. Barber, A.: Dual intuitionistic linear logic. Tech. Rep. ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh (Sep 1996)
3. 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). Springer LNCS 933, Kazimierz, Poland (Sep 1994), an extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge
4. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: M.Felleisen, P.Gardner (eds.) Proceedings of the European Symposium on Programming (ESOP'13). pp. 330–349. Springer LNCS 7792, Rome, Italy (Mar 2013)
5. 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)
6. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science (2013), to appear. Special Issue on Behavioural Types.
7. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation 207(10), 1044–1077 (Oct 2009)

8. Chang, B.Y.E., Chaudhuri, K., Pfenning, F.: A judgmental analysis of linear logic. Tech. Rep. CMU-CS-03-131R, Carnegie Mellon University, Department of Computer Science (Dec 2003)
9. 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)
10. Gay, S.J., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica 42(2–3), 191–225 (2005)
11. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. Journal of Functional Programming 20(1), 19–50 (Jan 2010)
12. Girard, J.Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
13. Honda, K.: Types for dyadic interaction. In: 4th International Conference on Concurrency Theory. pp. 509–523. CONCUR'93, Springer LNCS 715 (1993)
14. 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)
15. Laurent, O.: Polarized proof-nets: Proof-nets for LC. In: Girard, J.Y. (ed.) Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999). pp. 213–227. Springer LNCS 1581, L'Aquila, Italy (Apr 1999)
16. Mostrous, D., Vasconcelos, V.: Affine sessions. In: Kühn, E., Pugliese, R. (eds.) 16th International Conference on Coordination Models and Languages. pp. 115–130. Springer LNCS 8459, Berlin, Germany (Jun 2014)
17. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Information and Computation 239, 254–302 (2014)
18. Reed, J.: A judgmental deconstruction of modal logic (2009), unpublished manuscript
19. Simmons, R.J.: Substructural Logical Specifications. Ph.D. thesis, Carnegie Mellon University (Nov 2012), available as Technical Report CMU-CS-12-142
20. Toninho, B.: A Logical Foundation for Session-based Concurrent Computation. Ph.D. thesis, Carnegie Mellon University and New University of Lisbon (2015), in preparation
21. 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)
22. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014). Rome, Italy (Sep 2014), to appear
23. Vasconcelos, V.T.: Fundamentals of session types. Information and Computation 217, 52–70 (2012)
24. 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)