A Message-Passing Interpretation of Adjoint Logic

Klaas Pruiksma
Carnegie Mellon University
Computer Science Department
kpruiksm@cs.cmu.edu

Frank Pfenning
Carnegie Mellon University
Computer Science Department
fp@cs.cmu.edu

We present a system of session types based on *adjoint logic* which generalize standard binary session types [14]. Our system allows us to uniformly capture several new behaviors in the space of asynchronous message-passing communication, including *multicast*, where a process sends a single message to multiple clients, *replicable services*, which have multiple clients and replicate themselves on-demand to handle requests from those clients, and *cancellation*, where a process discards a channel without communicating along it. We provide session fidelity and deadlock-freedom results for this system, from which we then derive a logically justified form of garbage collection.

1 Introduction

Binary session types [14] were designed to specify the communication behavior between two message-passing processes. But there are patterns of communication that do not fall into this category. One example is one provider of a *replicable service* with multiple clients. Another is a *multicast*, that is, a process sending one message to multiple recipients. A third one is a client that no longer wishes to use a service, a form of *cancellation*. In this paper we provide a uniform language and operational semantics rooted in logic that captures such patterns of asynchronous communication. It generalizes the usual binary session types by supporting multiple *modes of communication*. In each of these modes every channel has a unique *provider* (which may send or receive), and possibly multiple clients. We identify the following modes: *linear* (a unique client that must communicate, as with the usual binary session types), *affine* (a unique client that may communicate or cancel), *strict* (multiple clients, each of which must communicate), and *unrestricted* (multiple clients, each of which may or may not communicate, which captures both replicable services and multicast).

A type system that uniformly integrates all of these patterns is not obvious if we want to preserve the desirable properties of session fidelity and deadlock freedom that we obtain from binary session types. Underlying our approach is *adjoint logic* [24, 15, 23], which generalizes *intuitionistic linear logic* [12, 11] and LNL [2] by decomposing the usual exponential modality $!A$ into two adjoint modal operators and also affords individual control over the structural rules of weakening and contraction. We provide a formulation of adjoint logic in which cut reduction corresponds to asynchronous communication, and from which session fidelity and deadlock freedom derive. Moreover, our formulation uses a form of explicit structural rules embedded in a multicut, where weakening corresponds to cancellation and contraction corresponds to duplication of a message or service.

Some of these patterns have been previously addressed with varying degrees of proximity to an underlying logic. A replicable service with multiple clients can be achieved with *access points* [9] or *persistent services* of type $!A$ [4]. Cancellation can be addressed with affine types [17, 25, 19] further developed for asynchronous communication and general handling of failure [8]. Cancellation can also be handled with modalities used to label cancellable types [3]. This approach differs from ours in a few respects — first, Caires and Pérez work in a purely synchronous setting, without multicast, and second,

Submitted to:
PLACES 2019

they focus heavily on introducing nondeterminism, which we believe to be orthogonal to (our form of) cancellation. Closest to the present proposal is a polarized formulation of asynchronous communication in adjoint logic [21] which had several shortcomings that are addressed here. Specifically, the mode hierarchy was fixed to have only three modes (linear, affine, and unrestricted), and the unrestricted mode only allowed a single kind of proposition $\uparrow^{\mathsf{U}}_{m} A_m$. This meant that, for example, multicast was not representable. Also, the rules left weakening and contraction implicit, which means that there is no explicit cancellation or distributed garbage collection, which is only briefly hinted at as a possibility [13].

The Curry-Howard correspondence relates propositions to types, proofs to programs, and proof reduction to computation. Cut reductions in a pure sequent calculcus for linear logic [4, 27] naturally correspond to synchronous communication because both premises of the cut are reduced at the same time. We reformulate adjoint logic with a nonstandard sequent calculus in which noninvertible rules are presented as axioms, that is, rules with no premises. As our operational interpretation shows, an axiom can be seen as a message and cut reduction in this sequent calculus corresponds to asynchronous communication. Another unusual aspect of our sequent calculus is that we generalize cut to a sound rule of multicut [10, 18], which operationally allows one provider to connect with multiple clients. Two further consequences of this reformulation are that (a) no explicit rules are needed for weakening and contraction, and yet (b) channels and resources are tracked with sufficient precision that computation in a network of processes "leaves no garbage" (see section 4). This is the concurrent realization of the early observation by Girard and Lafont [11] that functional computation based on intuitionistic linear logic does not require a garbage collector. Cancellation [17, 8] is a natural consequence, without requiring any special mechanism, but our system goes beyond it in the sense that processes with multiple clients will also terminate once no clients are left.

We begin with a brief discussion of our type system (section 2), deferring discussion of the underlying logic to appendix A, in order to focus on the programming system. We then present an operational semantics (section 3): our first major contribution. It models a variety of asynchronous communication behaviors, uniformly generalizing previous systems. We close by briefly presenting our results on session fidelity and deadlock-freedom, along with a brief discussion of the "garbage-collection" result that follows from them (section 4).

2 Language and Typing

Our typing judgment for processes P is based on *intuitionistic sequents* of the form

$$(x^1 : A^1) \cdots (x^n : A^n) \vdash P :: (x : A)$$

where each of the x^i are *channels* that P *uses* and x is a channel that P *provides*. All of these channels must be distinct and we abbreviate the collection of antecedents as Ψ. The *session types* A^i and A specify the communication behavior that the process P must follow along each of the channels.

Such sequents are standard for the intuitionistic approach to understanding binary session types (e.g., [4]) where the channels are *linear* in that every channel in a network of processes has exactly one provider and exactly one client. In the closely related formulation based on classical linear logic [27] all channels are on the right-hand side of the turnstile, but each linear channel still has exactly two endpoints.

We generalize this significantly by assigning to each channel an intrinsic *mode* m. Each mode m is assigned a set of structural properties $\sigma(m)$ among W (for weakening) and C (for contraction). Separating

$$\frac{}{(a:A_m)\vdash c\leftarrow a :: (c:A_m)}\ \mathsf{id} \qquad \frac{\Psi\geq m\geq k \quad |S|\sim m \quad \Psi\vdash P::(x:A_m) \quad (S:A_m)\ \Psi'\vdash Q::(c:C_k)}{\Psi\ \Psi'\vdash S\leftarrow(\nu x)P;Q::(c:C_k)}\ \mathsf{cut}(S)$$

$$\frac{\ell\in I}{(a:A^\ell_m)\vdash c.\ell(a)::(c:\oplus_{i\in I}A^i_m)}\ \oplus R^0_\ell \qquad \frac{\Psi\ (x_i:A^i_m)\vdash P_i::(c:C_k)\ \text{for each } i\in I}{\Psi\ (a:\oplus_{i\in I}A^i_m)\vdash \mathsf{case}\ a\ (i(x_i)\Rightarrow P_i)_{i\in I}::(c:C_k)}\ \oplus L$$

$$\frac{\Psi\vdash P_i::(x_i:A^i_m)\ \text{for each } i\in I}{\Psi\vdash \mathsf{case}\ c\ (i(x_i)\Rightarrow P_i)_{i\in I}::(c:\&_{i\in I}A^j_m)}\ \&R \qquad \frac{\ell\in I}{(a:\&_{i\in I}A^i_m)\vdash a.\ell(c)::(c:A^\ell_m)}\ \&L^0_\ell$$

$$\frac{}{(a:A_m)\ (b:B_m)\vdash c.\langle a,b\rangle::(c:A_m\otimes B_m)}\ \otimes R^0 \qquad \frac{\Psi\ (x:A_m)\ (y:B_m)\vdash P::(c:C_k)}{\Psi\ (a:A_m\otimes B_m)\vdash \mathsf{case}\ a(\langle x,y\rangle\Rightarrow P)::(c:C_k)}\ \otimes L$$

$$\frac{}{\cdot\vdash c.\langle\rangle::(c:\mathbf{1}_m)}\ \mathbf{1}R \qquad \frac{\Psi\vdash P::(c:C_k)}{\Psi\ (a:\mathbf{1}_m)\vdash \mathsf{case}\ a(\langle\rangle\Rightarrow P)::(c:C_k)}\ \mathbf{1}L$$

$$\frac{(x:A_m)\ \Psi\vdash P::(y:B_m)}{\Psi\vdash \mathsf{case}\ c(\langle x,y\rangle\Rightarrow P)::(c:A_m\multimap B_m)}\ \multimap R \qquad \frac{}{(a:A_m)\ (c:A_m\multimap B_m)\vdash c.\langle a,b\rangle::(b:B_m)}\ \multimap L^0$$

$$\frac{\Psi\vdash P::(x:A_k)}{\Psi\vdash \mathsf{case}\ c(\mathsf{shift}(x)\Rightarrow P)::(c:\uparrow^m_k A_k)}\ \uparrow R \qquad \frac{}{(a:\uparrow^m_k A_k)\vdash a.\mathsf{shift}(c)::(c:A_k)}\ \uparrow L^0$$

$$\frac{}{(a:A_m)\vdash c.\mathsf{shift}(a)::(c:\downarrow^m_k A_m)}\ \downarrow R^0 \qquad \frac{\Psi\ (x:A_m)\vdash P::(c:C_\ell)}{\Psi\ (a:\downarrow^m_k A_m)\vdash \mathsf{case}\ a(\mathsf{shift}(c)\Rightarrow P)::(c:C_\ell)}\ \downarrow L$$

Figure 1: Process Assignment for Asynchronous Adjoint Logic

m from $\sigma(m)$ allows us to have multiple modes with the same set of structural properties.[1] No matter which structural properties are available for a channel, each active channel will still have *exactly one provider*. Beyond that, a channel x_m with $\mathrm{W}\in\sigma(m)$ may not have any clients. Furthermore, a channel x_m with $\mathrm{C}\in\sigma(m)$ may have multiple clients. All other properties of our system of session types for processes derive systematically from these simple principles.

The modes are organized into a preorder where $m\geq k$ requires that $\sigma(m)\supseteq\sigma(k)$, that is, m must allow more structural properties than k. In order to guarantee session fidelity and deadlock freedom, for any sequent $\Psi\vdash P::(x_m:A_m)$ is must be the case that for every $y_k:B_k\in\Psi$ we have $k\geq m$. For example, if m permits contraction and therefore P may have multiple clients, then for any y_k in Ψ, mode k must also permit contraction because (intuitively) if x_m is referenced multiple times then, indirectly, so is y_k. If $k\geq m$ then this is ensured. We express this with the *presupposition* that

$$\Psi\vdash P::(x_m:A_m) \quad \text{requires} \quad \Psi\geq m$$

where $\Psi\geq m$ simply means $k\geq m$ for every $y_k:A_k\in\Psi$. We will only consider sequents satisfying this presupposition, so our rules, when they are used to break down a conclusion into the premises, must preserve this fundamental property which we call *the declaration of independence*.

In our formulation, channels x_m as well as types A_m are endowed with modes which must always be consistent between a channel and its type $(x_m:A_m)$. We therefore often omit redundant mode annotations on channels.

[1] This allows us, for example, to model the modal logic S4 or lax logic (the logical origins of comonadic and monadic programming), each with two modes both satisfying weakening and contraction, as well as linear analogues of these constructions.

The complete set of rules for the typing judgment are given in Fig. 1. We first examine the judgmental rules that explain the meaning of identity and composition. Identity (rule id) is straightforward: a process $c \leftarrow a$ providing c defers to the provider of a, which is possible as long as a and c have the same type and mode. This is usually called *forwarding* or *identification* of the channels a and c.

The usual logical rule of cut corresponds to the parallel composition of two processes with a single private channel for communication between them. However, ordinary cut is insufficiently general to describe the situation where a single provider of a channel x_m may have multiple clients ($C \in \sigma(m)$) or no clients ($W \in \sigma(m)$). We therefore generalize it to a form of multicut, [2] where the channel x_m provided by P is known by multiple aliases in the set of channels S in Q as long as the multiplicity of the aliases is permitted by the mode. This is expressed as $|S| \sim m$ and is sufficient for static typing. Formally, we define this condition by $0 \sim m$ if $\mathsf{W} \in \sigma(m)$, $1 \sim m$ always, and $k \sim m$ for $k \geq 2$ if $\mathsf{C} \in \sigma(m)$. When processes execute we will have an even more general situation where one provider has multiple separate client processes, which is captured in the typing judgment for process configurations (section 3).

Next we come to the various session types. From the logical perspective, these are the propositions of adjoint logic.

$$A_m, B_m \quad ::= \quad p_m \mid A_m \multimap_m B_m \mid A_m \otimes_m B_m \mid \mathbf{1}_m \mid \oplus_m \{ i : A^i_m \}_{i \in I} \mid \&_m \{ i : A^i_m \}_{i \in I} \mid \uparrow^m_k A_k \mid \downarrow^\ell_m A_\ell$$

Here, p_m stands for atomic propositions at mode m. The other connectives, other than $\uparrow^m_k$ and $\downarrow^\ell_m$, are standard linear logic connectives, except that they are only allowed to combine types (propositions) at the same mode. Since the mode of a connective can be inferred from the modes of the types it connects (other than for shifts), we omit subscripts on connectives. Note also that $\&$ and $\oplus$ have been generalized to n-ary forms from the usual binary forms. This is convenient for programming. We will use a label set $I = \{\pi_1, \pi_2\}$ when working with the binary forms $A_m \,\&\, B_m$ and $A_m \oplus B_m$, where π_1 selects the left-hand type and π_2 selects the right-hand type. The operational meaning of these connectives (as discussed further in section 3) is largely similar to that in past work (e.g. [4]), with $\multimap_m$ and $\otimes_m$ sending channels along other channels, $\mathbf{1}_m$ sending an end-of-communication message, and $\oplus_m$ and $\&_m$ sending labels. The shifts send a simple shift message to signal a transition between modes, either *up* ($\uparrow^m_k$) from k to some $m \geq k$ or *down* ($\downarrow^\ell_m$) from ℓ to some $m \leq \ell$.

We provide proof terms for the rules in our sequent calculus, as shown in Figure 1. We can then interpret the proof terms as process expressions, and these rules are used to give the typing judgment for such processes. Table 1 gives the informal meaning of each such process term.

In general, our process syntax represents an intermediate point between a programmer-friendly syntax and a notation in which it is easy to describe the operational semantics and prove progress and preservation. When compared to, for instance, SILL [26], the main revisions are that (1) we make channel continuations explicit in order to facilitate asynchronous communication while preserving message order [7], and (2) we distinguish between an *internal name* for the channel provided by a process and *external names* connecting it to multiple clients.

Some simple examples. We provide here some small examples with their types; additional examples which highlight more interesting behavior can be found in appendix C.

First, we have a process that can be written at any mode m, which witnesses that $\otimes_m$ is commutative.

$(x : A_m \otimes B_m) \vdash \mathsf{case}\, x\, (\langle y, x' \rangle \Rightarrow z.\langle x', y \rangle) :: (z : B_m \otimes A_m)$

[2] The term "multicut" has been used in the literature for different rules. We follow here the proof theory literature [18, Section 5.1], where it refers to a rule that cuts out some number of copies of the *same* proposition A, as in Gentzen's original proof of cut elimination [10], where he calls it "Mischung".

Process term	Meaning
$a \leftarrow c$	Identify channels a and c.
$S \leftarrow (\nu x)P\,;Q$	Spawn a new process P providing channel x with aliases S to be used by Q. Here, x is the *internal name* in P for the channel offered by P, and S is the set of *external names* of the same channel as used in Q.
$c.\ell(a)$	Send the label ℓ and the channel a along c.
$\mathsf{case}\ c(i(x_i) \Rightarrow P_i\}_{i\in I})$	Receive a label i and a channel x_i from c, continue as P_i.
$c.\langle a,b\rangle$	Send the channels a and b along c.
$\mathsf{case}\ c(\langle x,y\rangle \Rightarrow P)$	Receive channels x and y from c to be used in P.
$c.\langle\rangle$	End communication over c by sending a terminal message.
$\mathsf{case}\ c(\langle\rangle \Rightarrow P)$	Wait for c to be closed, continue as P.
$c_m.\mathsf{shift}(a_k)$	Send a shift, from mode m to mode k
$\mathsf{case}\ c_m\ (\mathsf{shift}(x_k) \Rightarrow P)$	Receive a shift from mode m to mode k

Table 1: Informal Meanings of Process Terms

If m is a mode that admits contraction, we can write the following process, which witnesses that $A_m \,\&\, B_m$ proves $A_m \otimes B_m$ in the presence of contraction. '%' starts a comment.

$$
\begin{array}{lll}
(p : A_m \,\&\, B_m) \vdash \{p_1, p_2\} \leftarrow (\nu q)\,(q \leftarrow p); & \% \ \{p_1,p_2\} \leftarrow \mathsf{copy}\ p \\
\qquad\qquad x \leftarrow (\nu a)\,p_1.\pi_1(a); & \\
\qquad\qquad y \leftarrow (\nu b)\,p_2.\pi_2(b); & \\
\qquad\qquad z.\langle x,y\rangle & :: (z : A_m \otimes B_m)
\end{array}
$$

If m is a mode that admits weakening, we can write the following process, which witnesses that $A_m \otimes B_m$ proves $A_m \,\&\, B_m$ in the presence of weakening.

$$
\begin{array}{lll}
(x : A \otimes B) \vdash \mathsf{case}\ p\ \ (\ \pi_1(p_1) \Rightarrow \mathsf{case}\ x\,(\langle y,z\rangle \Rightarrow & & \\
\qquad\qquad \{\,\} \leftarrow (\nu a)\,(a \leftarrow z); & \% \ \mathsf{drop}\ z & \\
\qquad\qquad p_1 \leftarrow y) & & \\
\qquad \mid \pi_2(p_2) \Rightarrow \mathsf{case}\ x\,(\langle y,z\rangle \Rightarrow & & \\
\qquad\qquad \{\,\} \leftarrow (\nu a)\,(a \leftarrow y); & \% \ \mathsf{drop}\ y & \\
\qquad\qquad p_2 \leftarrow z)) & & \\
:: (p : A \,\&\, B) & &
\end{array}
$$

3 Operational Semantics

In order to describe the computational behavior of process expressions, we need to first give some syntax for the computational artifacts, which are running processes $\mathsf{proc}(S,\Delta,a,P)$. Such a process executes P and provides a channel a while using the channels in the channel set Δ. S is a set of aliases for the channel a, which can be referred to by one or more clients. Each alias $c \in S$ is used by at most one client, but one client may use multiple such aliases. Note that as the aliases in S are the only way to interact with the channel a from an external process, the objects $\mathsf{proc}(S,\Delta,a,P)$ and $\mathsf{proc}(S,\Delta,b,P[b/a])$ are equivalent — changing the internal name of a process has no effect on its interactions with other processes.

A *process configuration* is a multiset of processes:

$$\mathscr{C} \quad ::= \quad \mathsf{proc}(S,\Delta,a,P) \mid (\cdot) \mid \mathscr{C}\ \mathscr{C}'$$

where we require that all the aliases or names provided by the processes $\mathsf{proc}(S,\Delta,a,P)$ are distinct, i.e., given objects $\mathsf{proc}(S,\Delta_1,a,P)$ and $\mathsf{proc}(T,\Delta_2,b,Q)$ in the same process configuration, S and T are disjoint. We will specify the operational semantics in the form of *multiset rewriting rules* [6]. That means we show how to rewrite some subset of the configuration while leaving the remainder untouched. This form provides some assurance of the locality of the rules.

It simplifies the description of the operational semantics if for any process $\mathsf{proc}(S,\Delta,a,P)$, Δ consists of exactly the free channels (other than a) in P. This requires that we restrict the labeled internal and external choices, $\bigoplus_{i\in I} A^i_m$ and $\mathop{\&}_{i\in I} A^i_m$ to the case where $I \neq \emptyset$. Since a channel of empty choice type can never carry any messages, this is not a significant restriction in practice.

In order to understand the rules of the operational semantics, it will be helpful to understand the typing of configurations. The judgment has the form $\Psi \vDash \mathscr{C} :: \Psi'$ which expresses that using the channels in Ψ, configuration $\mathscr{C}$ provides the channels in Ψ'. This allows a channel that is not mentioned at all in $\mathscr{C}$ to appear in both Ψ and Ψ'—we think of such a channel as being "passed through" the configuration.

Note that while the configuration typing rules induce an ordering on a configuration, the configuration itself is not inherently ordered. The key rule is the first: for any object $\mathsf{proc}(S,\Delta,a,P)$ we require that P is well-typed on some subset of the available channels while the others are passed through. Here we write $\overline{\Psi}$ for the set of channels declared in Ψ, which must be exactly those used in the typing of P. Moreover, *externally* such a process provides the channels $S = \{a^1_m,\ldots,a^n_m\}$, all of the same type A_m. We use the abbreviation $(S : A_m)$ for $a^1_m : A_m,\ldots,a^n_m : A_m$. Finally, we enforce that the number of clients must be compatible with the mode m of the offered channel, which is exactly that $|S| \sim m$, as defined in section 2.

$$\frac{|S| \sim m \quad \Psi' \vdash P :: (a : A_m)}{\Psi\ \Psi' \vDash \mathsf{proc}(S,\overline{\Psi'},a,P) :: \Psi\ (S : A_m)}\ \mathsf{Proc} \qquad \frac{}{\Psi \vDash (\cdot) :: \Psi}\ \mathsf{Id} \qquad \frac{\Psi \vDash \mathscr{C} :: \Psi' \quad \Psi' \vDash \mathscr{C}' :: \Psi''}{\Psi \vDash \mathscr{C}\ \mathscr{C}' :: \Psi''}\ \mathsf{Comp}$$

The identity and composition rules are straightforward. The empty context $(\cdot)$ provides Ψ if given Ψ, since it does not use any channels in Ψ or provide any additional channels. Composition just connects configurations with compatible interfaces: what is provided by $\mathscr{C}$ is used by $\mathscr{C}'$.

The computation rules we discuss in this section can be found in Figure 2. Remarkably, the computation rules do not depend on the modes, although some of the rules will naturally only apply at modes satisfying certain structural properties.

Judgmental rules. The identity rule (written as $\xRightarrow{\mathsf{id}}$) describes how an identity process (for instance, $\mathsf{proc}(S,\{c\},a,a \leftarrow c)$) may interact with other processes. We think of such a process as connecting the provider of c to clients in S, and therefore sometimes call it a *forwarding process*. A forwarding process interacts with the provider of c, telling it to replace c with S in its set of clients. In adding S to the set of clients, the forwarding process accomplishes its goal of connecting the provider of c to S, and so it can terminate.

The cut rule steps by spawning a new process which offers along a fresh set of channels S', all of which are used in Q, the continuation of the original process. Here we write Δ_P and Δ_Q for the set of free channels in P and Q, respectively.

Structural rules. A process with no clients can terminate (rule $\xRightarrow{\mathsf{drop}}$), but must notify all of the processes it uses that they should also terminate. It does so by sending each one a forwarding message,

$$\mathsf{proc}(T \cup \{c\}, \Delta, x, P),\ \mathsf{proc}(S, \{c\}, y, y \leftarrow c) \overset{\mathsf{id}}{\Longrightarrow} \mathsf{proc}(T \cup S, \Delta, x, P)$$

$$\mathsf{proc}(T, \Delta_P \cup \Delta_Q, y, S \leftarrow (\nu x)P; Q) \overset{\mathsf{cut}(S)}{\Longrightarrow} \mathsf{proc}(S', \Delta_P, x, P),\ \mathsf{proc}(T, \Delta_Q \cup \{S'\}, y, Q[S'/S])$$
(S' a fresh set of channels matching S)

(P not an identity) $\quad \mathsf{proc}(\emptyset, \Delta, x, P) \overset{\mathsf{drop}}{\Longrightarrow} \mathsf{proc}(\emptyset, \{b\}, y, y \leftarrow b)_{b \in \Delta}$

$$\mathsf{proc}(S \cup T, \Delta, x, P) \overset{\mathsf{copy}}{\Longrightarrow} \mathsf{proc}(\{b', b''\}, \{b\}, y, y \leftarrow b)_{b \in \Delta},\ \mathsf{proc}(S, \{b'\}_{b \in \Delta}, x, P[b'/b]),\ \mathsf{proc}(T, \{b''\}_{b \in \Delta}, x, P[b''/b])$$
(P not an identity and S, T non-empty)

$$\mathsf{proc}(\{b\}, \{c\}, x, x.\ell(c)),\ \mathsf{proc}(S, \Delta \cup \{b\}, z, \mathsf{case}\ b(i(y_i) \Rightarrow P_i)_{i \in I}) \overset{\oplus C}{\Longrightarrow} \mathsf{proc}(S, \Delta \cup \{c\}, z, P_\ell[c/y_\ell])$$

$$\mathsf{proc}(\{b\}, \Delta, x, \mathsf{case}\ x(i(y_i) \Rightarrow P_i)_{i \in I}),\ \mathsf{proc}(\{c\}, \{b\}, z, b.\ell(z)) \overset{\& C}{\Longrightarrow} \mathsf{proc}(\{c\}, \Delta, z, P_\ell[z/y_\ell])$$

$$\mathsf{proc}(\{b\}, \{c, d\}, w, w.\langle c, d\rangle),\ \mathsf{proc}(S, \Delta \cup \{b\}, z, \mathsf{case}\ b(\langle x, y\rangle \Rightarrow P)) \overset{\otimes C}{\Longrightarrow} \mathsf{proc}(S, \Delta \cup \{c, d\}, z, P[c/x, d/y])$$

$$\mathsf{proc}(\{b\}, \Delta, w, \mathsf{case}\ w(\langle x, y\rangle \Rightarrow P)),\ \mathsf{proc}(\{c\}, \{b, d\}, z, b.\langle d, z\rangle) \overset{\multimap C}{\Longrightarrow} \mathsf{proc}(\{c\}, \Delta \cup \{d\}, z, P[d/x, z/y])$$

$$\mathsf{proc}(\{b\}, \emptyset, x, x.\langle\rangle),\ \mathsf{proc}(S, \Delta \cup \{b\}, y, \mathsf{case}\ b(\langle\rangle \Rightarrow P)) \overset{\mathbf{1}C}{\Longrightarrow} \mathsf{proc}(S, \Delta, y, P)$$

$$\mathsf{proc}(\{b_k\}, \{c_m\}, x_k, x_k.\mathsf{shift}(c_m)),\ \mathsf{proc}(S, \Delta \cup \{b_k\}, y, \mathsf{case}\ b_k(\mathsf{shift}(z_m) \Rightarrow P)) \overset{\downarrow_k^m C}{\Longrightarrow} \mathsf{proc}(S, \Delta \cup \{c_m\}, y, P[c_m/z_m])$$

$$\mathsf{proc}(\{b_m\}, \Delta, x_m, \mathsf{case}\ x_m(\mathsf{shift}(z_k) \Rightarrow P)),\ \mathsf{proc}(\{c_k\}, \{b_m\}, y_k, b_m.\mathsf{shift}(y_k)) \overset{\uparrow_k^m C}{\Longrightarrow} \mathsf{proc}(\{c_k\}, \Delta, y_k, P[y_k/z_k])$$

Figure 2: Computation Rules for Asynchronous Adjoint Logic

effectively embodying a cancellation. In concert with the identity rule this accomplishes cascading cancellation in the distributed setting. Note that the mode m of channel a must admit weakening in order for the process on the left-hand side of the rule to be well-typed.

Similarly, a process with multiple clients can spawn a copy of itself, each with a strictly smaller set of clients (rule $\overset{\mathsf{copy}}{\Longrightarrow}$). If the process P is a replicable service, that is, if it has a negative type $\&$, $\multimap$, $\uparrow_k^m$, then this corresponds to actual process replication. If it has a positive type $\oplus$, $\otimes$, $\mathbf{1}$, $\downarrow_k^m$, this corresponds to duplicating a multicast message into copies for different subsets of recipients. The mode m of the channel a must admit contraction in order for the process on the left-hand side of the rule to be well-typed.

While both the drop and copy rules can be applied to any process with 0 or multiple clients, respectively, this does not cause any problems as long as we forbid them from executing on identity processes. If we apply drop or copy to an identity process, we end up with another process of the same form on the right-hand side of the rule, and so we could repeatedly apply drop or copy and not make any progress. As such, we forbid this use of the drop and copy rules.

For any other type of process, regardless of whether we drop/copy first or execute another communication rule first, we can eventually reach the same state, and so we do not need to make additional

restrictions (though an actual implementation would likely pick either a maximally eager or a maximally lazy strategy for applying these rules).

Additive and multiplicative connectives. In the rule for $\oplus$, the process $\mathsf{proc}(\{b\},\{c\},a,a.\ell(c))$ represents the message 'label ℓ with continuation c'. After this message has been received, the process terminates since b was its only client. The recipient selects the appropriate branch of the case construct and also substitutes the continuation channel c for the continuation variable d_ℓ.

The $\&$ computation rule is largely similar to that for $\oplus$, except that communication proceeds in the opposite direction—messages are sent *to* providers *from* clients, rather than from providers to clients as in the case of $\oplus$.

The multiplicative connectives $\otimes$ and $\multimap$ behave similarly to their additive counterparts, except that rather than sending and receiving labels, they send and receive channels together with a continuation, and so an extra substitution is required when receiving messages, while the $\mathbf{1}$ behaves as a nullary $\otimes$, allowing us to signal that no more communication is forthcoming along a channel, and to wait for such a signal before continuing to compute.

Shifts. We present the computation rules for shifts with modes marked explicitly on the relevant channels. Channels whose modes are unmarked may be at any mode (provided, of course, that the declaration of independence is respected).

Operationally, $\uparrow$ behaves essentially the same as unary $\&$, while $\downarrow$ behaves as unary $\oplus$. Their significance lies in the *mode shift* of the continuation channel that is transmitted, which is required for the configuration to remain well-typed.

The messages $\mathsf{shift}(a_k)$ or $\mathsf{shift}(c_m)$ should be thought of as signaling a transition between modes — to mode k for the former, and to mode m for the latter. Whether the transition is up or down depends on which direction the message is being sent in. As with other messages (in particular, the messages for $\oplus$ and $\&$), the continuation channels are made explicit.

4 Session Fidelity, Deadlock-Freedom, and Garbage Collection

While we can prove cut elimination for the form of adjoint logic presented in appendix A, from a programmer's perspective we are not interested in eliminating all cuts (which would correspond to reducing under λ-abstractions in a functional language) but rather we block when waiting to receive a message, analogous to a λ-abstraction waiting for input before it can reduce. What we prove instead are session fidelity and deadlock-freedom.

Session fidelity. The session fidelity theorem follows from a case analysis on the computation rule used to get that $\mathscr{C} \Rightarrow \mathscr{C}'$. In each case, we break $\mathscr{C}$ down to find the processes on which the computation rule acts, along with some collections of processes which are unaffected by the computation. From these pieces, we build a proof that $\Psi \vDash \mathscr{C}' :: \Psi'$.

Theorem 1 (Session Fidelity)**.** *If* $\Psi \vDash \mathscr{C} :: \Psi'$ *and* $\mathscr{C} \Rightarrow \mathscr{C}'$*, then* $\Psi \vDash \mathscr{C}' :: \Psi'$.

Deadlock-freedom. The progress theorem for a functional language states that an expression is either a value or it can take a step. Here we do not have values, but there is nevertheless a clear analogue between, say, a value $\lambda x.e$ that waits for an argument, and a process $\mathsf{case}\, x\,(\langle y,z\rangle \Rightarrow P)$ that waits for an input. We formalize this in the definition below.

Definition 1. We say that a process $\mathsf{proc}(S,\Delta,a,P)$ is *poised* on a if:

1. it is a process $\mathsf{proc}(S,\Delta,a,P)$ that sends on a — that is, P is of the form $(a.\_)$, or

2. it is a process $\mathsf{proc}(S,\Delta,a,P)$ that receives on a — that is, P is of the form $(\mathsf{case}\ a\ (\_))$.

Intuitively, $\mathsf{proc}(S,\Delta,a,P)$ is poised on a if it is blocked trying to communicate along a. This definition allows us to state the following progress theorem:

Theorem 2 (Deadlock-Freedom)**.** *If $(\cdot) \vDash \mathscr{C} :: \Psi$, then exactly one of the following holds:*

1. *There is a $\mathscr{C}'$ such that $\mathscr{C} \Rightarrow \mathscr{C}'$.*
2. *Every $\mathsf{proc}(S,\Delta,a,P)$ in $\mathscr{C}$ is poised on a.*

In order to prove this theorem, we first prove a lemma allowing us to take advantage of the ordering induced by configuration typing. We note that if object ψ is a client of object ϕ, ψ must occur to the right of ϕ in the ordering, and so if we can analyze a configuration from right to left, we consider each process before (or after, depending on your view of induction) all of its dependencies. To formalize this, we present a second set of rules defining another form of configuration typing (which will turn out to prove the same judgments as the original form).

$$\frac{}{\Psi \vDash' (\cdot) :: \Psi}\ \mathsf{Empty} \qquad \frac{|S| \sim m \quad \Psi \vDash' \mathscr{C} :: \Psi'\ \Psi'' \quad \Psi' \vdash P :: (a : A_m)}{\Psi \vDash' \mathscr{C}\ \mathsf{proc}(S,\overline{\Psi'},a,P) :: \Psi''\ (S : A_m)}\ \mathsf{Extend}$$

It is clear that if $\vDash$ and $\vDash'$ are the same, then we can perform induction using the Empty and Extend rules rather than the Id, Comp, and Proc rules, allowing us to analyze a configuration from right to left. We formalize this as lemma 1.

Lemma 1. *$\Psi \vDash \mathscr{C} :: \Psi'$ if and only if $\Psi \vDash' \mathscr{C} :: \Psi'$.*

This lemma is nearly immediate — all of the rules for $\vDash'$ are derivable from the rules of $\vDash$, and all rules of $\vDash$ but Comp are derivable from the rules of $\vDash'$. We therefore need only show (by an induction over the right-hand premise) that the version of the Comp rule with $\vDash$ replaced by $\vDash'$ is admissible.

The proof of deadlock-freedom then proceeds by an induction on the derivation of $(\cdot) \vDash \mathscr{C} :: \Psi$, using lemma 1 to work right to left. Writing $\mathscr{C} = \mathscr{C}'\ \mathsf{proc}(S,\overline{\Psi'},a.P)$, we see that either $\mathscr{C}'$ can step, in which case so can $\mathscr{C}$, or every process in $\mathscr{C}'$ is poised. Now we carefully distinguish cases on S (empty, singleton, or greater) and apply inversion to the typing of P to see that in each case the process either is poised, can take a step independently, or can interact with provider of a channel in $\overline{\Psi'}$.

Garbage collection. As we can see from the preservation theorem, the interface to a configuration never changes. While new processes may be spawned, they will have clients and are therefore not visible at the interface. This is in contrast to the semantics of shared channels in prior work (for example, in [4, 21]) where shared channels may show up as newly provided channels. Therefore they may be left over at the end of a computation without any clients.

This cannot happen here. Initially, at the top level, we envision starting with the configuration below on the left. Assuming this computation completes, by the progress property and the definition of *poised*, computation could only halt with the configuration on the right. In other words: no garbage!

$$\cdot \vDash \mathsf{proc}(\{c_0\},\cdot,c,P_0) :: (c_0 : \mathbf{1}) \qquad\qquad \cdot \vDash \mathsf{proc}(\{c_0\},\cdot,c,c.\langle\rangle) :: (c_0 : \mathbf{1})$$

One can generalize this to allow nontrivial output by allowing any purely positive type (that is, one which only uses the fragment of the logic with connectives $\oplus$, $\otimes$, $\mathbf{1}$, and $\downarrow$), such as $\oplus\{\mathsf{false} : \mathbf{1}, \mathsf{true} : \mathbf{1}\}$.

We can formalize this intuition by defining an *observable* configuration $\mathscr{C}$ which corresponds to our intuitive notion of garbage-free. We only define what it means for a configuration with purely positive

type to be observable. It is likely that this definition can be extended to encompass negative types as well, but it is not nearly as natural to do so.

A configuration $\mathscr{C}$ for which there is Ψ composed entirely of purely positive types such that $\cdot \vDash \mathscr{C} :: \Psi$ is *observable* at Ψ if, when we repeatedly receive messages from all channels we know about, starting from a state where we only know about Ψ, we eventually receive a message from every object in $\mathscr{C}$. If we do not care about the particular channels in Ψ, we may say simply that $\mathscr{C}$ is *observable*.

Definition 2. We define what it means for a configuration $\mathscr{C}$ to be observable at Ψ (written $\mathscr{C} \rhd \Psi$) inductively over the structure of $\mathscr{C}$.

1. $\mathsf{proc}(\{c\}, \cdot, x, x.\langle\rangle) \rhd (c : \mathbf{1})$.
2. If $\mathscr{C} \rhd \Psi\ (d : A^{\ell}_{m})$, then $\mathscr{C}\ \mathsf{proc}(\{c\}, \{d\}, x, x.\ell(d)) \rhd \Psi\ (c : \bigoplus_{i \in I} A^{i}_{m})$.
3. If $\mathscr{C} \rhd \Psi\ (d : A_m)$, then $\mathscr{C}\ \mathsf{proc}(\{c\}, \{d\}, x, x.\mathsf{shift}(d)) \rhd \Psi\ (c : \downarrow^{m}_{k} A_m)$.
4. If $\mathscr{C} \rhd \Psi\ (d : A_m)\ (e : B_m)$, then $\mathscr{C}\ \mathsf{proc}(\{c\}, \{d, e\}, x, x.\langle d, e\rangle) \rhd \Psi\ (c : A_m \otimes B_m)$.

We can then give the following corollary of our deadlock-freedom theorem:

Corollary 1. *If $\cdot \vDash \mathscr{C} :: \Psi$ for some Ψ consisting entirely of purely positive types and $\mathscr{C}$ cannot take any steps, then $\mathscr{C} \rhd \Psi$.*

This proof proceeds by a simple induction on the derivation of $\cdot \vDash \mathscr{C} :: \Psi$, using (Lemma 1) to work from right to left. At each step, we note that the rightmost process is poised. Because Ψ consists only of purely positive types, the rightmost process must therefore be sending a positive message. Moreover, it can only use channels of purely positive type. Well-typedness of the configuration then lets us apply the inductive hypothesis to the remainder of the configuration, at which point we can simply apply the definition of observability.

5 Conclusion

At this point, our formulation of adjoint logic and its operational semantics seem to provide a good explanation for a variety of patterns of asynchronous communication. The key behaviors which we can model (and importantly, model in a uniform fashion) are cancellation, replication, and multicast. We also obtain a foundation for a system of distributed garbage collection. Moreover, if used linearly, our semantics coincides with the purely linear semantics developed in prior work.

In parallel work we have also provided a shared memory semantics for a closely related formulation of adjoint logic with implicit structural rules [22]. In future work, we plan to investigate if the declaration of independence is sufficient to allow a *modular* combination of different operational interpretations for different modes. Of particular interest here would be the semantics with manifest sharing [1].

Acknowledgments

Supported by NSF Grant No. CCF-1718267: “Enriching Session Types for Practical Concurrent Programming”

References

[1] Stephanie Balzer & Frank Pfenning (2017): *Manifest Sharing with Session Types*. In: *International Conference on Functional Programming (ICFP)*, ACM, pp. 37:1–37:29, doi:10.1145/3110281.

[2] Nick Benton (1994): *A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models*. In Leszek Pacholski & Jerzy Tiuryn, editors: *Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94)*, Springer LNCS 933, Kazimierz, Poland, pp. 121–135, doi:10.1007/BFb0022251. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

[3] Luís Caires & Jorge A Pérez (2017): *Linearity, control effects, and behavioral types*. In: *European Symposium on Programming*, Springer, pp. 229–259, doi:10.1007/978-3-662-54434-1_9.

[4] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, Springer LNCS 6269, Paris, France, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[5] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear Logic Propositions as Session Types*. *Mathematical Structures in Computer Science* 26(3), pp. 367–423, doi:10.1016/j.tcs.2010.01.028.

[6] Iliano Cervesato & Andre Scedrov (2009): *Relating State-Based and Process-Based Concurrency through Linear Logic*. *Information and Computation* 207(10), pp. 1044–1077, doi:10.1016/j.ic.2008.11.006.

[7] Henry DeYoung, Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication*. In P. Cégielski & A. Durand, editors: *Proceedings of the 21st Conference on Computer Science Logic*, CSL 2012, pp. 228–242, doi:10.4230/LIPIcs.CSL.2012.228.

[8] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types*. In: *Proceedings of the 46th Symposium on Programming Languages (POPL 2019)*, ACM, Cascais, Portugal, pp. 28:1–28:29.

[9] Simon J. Gay & Vasco T. Vasconcelos (2010): *Linear Type Theory for Asynchronous Session Types*. *Journal of Functional Programming* 20(1), pp. 19–50, doi:10.1006/inco.1994.1093.

[10] Gerhard Gentzen (1935): *Untersuchungen über das Logische Schließen*. *Mathematische Zeitschrift* 39, pp. 176–210, 405–431, doi:10.1007/BF01201353. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[11] J.-Y. Girard & Y. Lafont (1987): *Linear Logic and Lazy Computation*. In H. Ehrig, R. Kowalski, G. Levi & U. Montanari, editors: *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, 2, Springer-Verlag LNCS 250, Pisa, Italy, pp. 52–66, doi:10.1007/BFb0014972.

[12] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[13] Dennis Griffith (2016): *Polarized Substructural Session Types*. Ph.D. thesis, University of Illinois at Urbana-Champaign.

[14] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *4th International Conference on Concurrency Theory*, CONCUR'93, Springer LNCS 715, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[15] Daniel R. Licata & Michael Shulman (2016): *Adjoint Logic with a 2-Category of Modes*. In: *International Symposium on Logical Foundations of Computer Science (LFCS)*, Springer LNCS 9537, pp. 219–235, doi:10.1007/978-3-319-27683-0_16.

[16] Daniel R. Licata, Michael Shulman & Mitchell Riley (2017): *A Fibrational Framework for Substructural and Modal Logics*. In: *International Conference on Formal Structures for Computation and Deduction*, LIPIcs, Oxford, doi:10.4230/LIPIcs.FSCD.2017.25.

[17] Dimitris Mostrous & Vasco Vasconcelos (2014): *Affine Sessions*. In E. Kühn & R. Pugliese, editors: *16th International Conference on Coordination Models and Languages*, Springer LNCS 8459, Berlin, Germany, pp. 115–130, doi:10.1007/978-3-662-43376-8_8.

[18] Sara Negri & Jan von Plato (2001): *Structural Proof Theory*. Cambridge University Press, doi:10.1017/CBO9780511527340.

[19] Luca Padovani (2017): *A Simple Library Implementation of Binary Sessions*. Journal of Functional Programming 27(e4), doi:10.1016/0304-3975(83)90059-2.

[20] Frank Pfenning (2016): *Law and Order*. Available at `http://www.cs.cmu.edu/~fp/courses/15816-f16/lectures/08-lawandorder.pdf`. Lecture notes on *Substructural Logics*.

[21] Frank Pfenning & Dennis Griffith (2015): *Polarized Substructural Session Types*. In A. Pitts, editor: *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, Springer LNCS 9034, London, England, pp. 3–22, doi:10.1007/978-3-662-46678-0_1. Invited talk.

[22] Frank Pfenning & Klaas Pruiksma (2018): *A Shared Memory Semantics for Session Types*. Invited talk at the Workshop on Linearity/TLLA, Oxford, UK.

[23] Klaas Pruiksma, William Chargin, Frank Pfenning & Jason Reed (2018): *Adjoint Logic*. Available at `http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf`. Unpublished manuscript.

[24] Jason Reed (2009): *A Judgmental Deconstruction of Modal Logic*. Available at `http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf`. Unpublished manuscript.

[25] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In: *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016)*, LICIcs 56, Rome, Italy, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[26] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In M.Felleisen & P.Gardner, editors: *Proceedings of the European Symposium on Programming (ESOP'13)*, Springer LNCS 7792, Rome, Italy, pp. 350–369, doi:10.1007/978-3-642-37036-6_20.

[27] Philip Wadler (2012): *Propositions as Sessions*. In: *Proceedings of the 17th International Conference on Functional Programming*, ICFP 2012, ACM Press, Copenhagen, Denmark, pp. 273–286, doi:10.1145/2364527.2364568.

A Adjoint Logic

We present here a brief overview of the formulation of adjoint logic that we take as a basis for the semantics presented in the main body of the paper. Adjoint logic can be thought of as a schema to define particular logics. The schema is parameterized by a set of modes of truth m, where each proposition and logical connective is indexed by its mode. Furthermore, each mode intrinsically carries a set of structural properties $\sigma(m) \subseteq \{\mathsf{W}, \mathsf{C}\}$ where W stands for *weakening* and C stands for *contraction*. As a concession to simplicity of the presentation, in this paper we always allow exchange, although nothing stands in the way of an even more general framework [20]. In addition, an instance requires a preorder between modes, where $m \geq k$ expresses that the proof of a proposition of mode k may depend on a hypotheses of mode m. This preorder embodies the *declaration of independence*:

A proof of A_k may only depend on hypotheses B_m for $m \geq k$.

The form of a sequent is

$$\Psi \vdash A_k \quad \text{where } \Psi \geq k$$

where Ψ is a collection of *antecedents* of the form $(x_i : B^i_{m_i})$ with each $m_i \geq k$, where all the variables x_i are distinct. This critical presupposition is abbreviated as $\Psi \geq k$. Furthermore, the order of the antecedents does not matter since we always allow exchange.

$$\frac{}{(x:A_m)\vdash A_m}\ \mathsf{id} \qquad \frac{\Psi\geq m\geq k \quad \Psi\vdash A_m \quad (x:A_m)\ \Psi'\vdash C_k}{\Psi\ \Psi'\vdash C_k}\ \mathsf{cut}$$

$$\frac{\mathsf{W}\in\sigma(m) \quad \Psi\vdash C_k}{\Psi\ (x:A_m)\vdash C_k}\ \mathsf{weaken} \qquad \frac{\mathsf{C}\in\sigma(m) \quad \Psi\ (y:A_m)\ (z:A_m)\vdash C_k}{\Psi\ (x:A_m)\vdash C_k}\ \mathsf{contract}$$

$$\frac{\ell\in I \quad \Psi\vdash A^\ell_m}{\Psi\vdash \bigoplus_{i\in I} A^i_m}\ \oplus R_\ell \qquad \frac{\Psi\ (y:A^i_m)\vdash C_k \text{ for each } i\in I}{\Psi\ (x:\bigoplus_{i\in I} A^i_m)\vdash C_k}\ \oplus L$$

$$\frac{\Psi\vdash A^i_m \text{ for each } i\in I}{\Psi\vdash \mathop{\&}_{i\in I} A^i_m}\ \& R \qquad \frac{\ell\in I \quad \Psi\ (y:A^\ell_m)\vdash C_k}{\Psi\ (x:\mathop{\&}_{i\in I} A^i_m)\vdash C_k}\ \& L_\ell$$

$$\frac{\Psi\vdash A_m \quad \Psi'\vdash B_m}{\Psi\ \Psi'\vdash A_m\otimes B_m}\ \otimes R \qquad \frac{\Psi\ (y:A_m)\ (z:B_m)\vdash C_k}{\Psi\ (x:A_m\otimes B_m)\vdash C_k}\ \otimes L \qquad \frac{}{\cdot\vdash \mathbf{1}_m}\ \mathbf{1}R \qquad \frac{\Psi\vdash C_k}{\Psi\ (x:\mathbf{1}_m)\vdash C_k}\ \mathbf{1}L$$

$$\frac{(x:A_m)\ \Psi\vdash B_m}{\Psi\vdash A_m\multimap B_m}\ \multimap R \qquad \frac{\Psi'\geq m \quad \Psi'\vdash A_m \quad \Psi\ (y:B_m)\vdash C_k}{\Psi\ \Psi'\ (x:A_m\multimap B_m)\vdash C_k}\ \multimap L$$

$$\frac{\Psi\vdash A_k}{\Psi\vdash \uparrow^m_k A_k}\ \uparrow R \qquad \frac{k\geq \ell \quad \Psi\ (y:A_k)\vdash C_\ell}{\Psi\ (x:\uparrow^m_k A_k)\vdash C_\ell}\ \uparrow L \qquad \frac{\Psi\geq m \quad \Psi\vdash A_m}{\Psi\vdash \downarrow^m_k A_m}\ \downarrow R \qquad \frac{\Psi\ (y:A_m)\vdash C_\ell}{\Psi\ (x:\downarrow^m_k A_m)\vdash C_\ell}\ \downarrow L$$

Figure 3: Rules of Adjoint Logic

In addition, we require the preorder between modes to be compatible with their structural properties: that is, $m\geq k$ implies $\sigma(m)\supseteq\sigma(k)$. This is necessary to guarantee cut elimination.

Finally, we may define fragments by restricting the set of propositions we consider for a given mode.

The propositions at each mode are constructed uniformly, remaining within the same mode, except for the *shift operators* that move between modes. They are $\uparrow^m_k A_k$ (pronounced *up*), which is a proposition at mode m and requires $m\geq k$; and $\downarrow^\ell_m A_\ell$ (*down*), which is also a proposition at mode m, and which requires $\ell\geq m$.

At this point we can already write out the syntax of propositions.

$$A_m, B_m \quad ::= \quad p_m \mid A_m \multimap_m B_m \mid A_m \otimes_m B_m \mid \mathbf{1}_m \mid \mathop{\oplus_m}_{i\in I} A^i_m \mid \mathop{\&_m}_{i\in I} A^i_m \mid \uparrow^m_k A_k \mid \downarrow^\ell_m A_\ell$$

Here p_m stands for atomic propositions at mode m. Due to the needs of our operational interpretation, we generalize internal and external choice to n-ary constructors parameterized by an index set I. So we write $A^1_m \oplus A^2_m = \bigoplus_{i\in\{1,2\}} A^i_m$.

Remarkably, the right and left rules in the sequent calculus defining the logical connectives are the same for each mode and are complemented by the permissible structural rules.

A.1 Judgmental and structural rules

The rules for adjoint logic can be found in fig. 3, in which we give a more standard presentation of the logic than that used by the operational semantics (fig. 1). We begin with the judgmental rules of identity and cut, which express the connection between antecedents and succedents. Identity says that if we assume A_m we are allowed to conclude A_m. Cut says the opposite: if we can conclude A_m we are allowed to assume A_m *as long as the declaration of independence is respected*.

As is common for the sequent calculus, we read the rules in the direction of bottom-up proof construction. For the cut rule, this means we should assume that the conclusion $\Psi\ \Psi' \vdash C_k$ is well-formed and, in particular, that $\Psi \geq k$ and $\Psi' \geq k$. Therefore, if we check that $m \geq k$, then we know that the second premise, $(x : A_m)\ \Psi' \vdash C_k$, will also be well-formed. For the first premise to be well-formed, we need to check outright that $\Psi \geq m$.

The structural rules of weakening and contraction just need to verify that the mode of the principal formula permits the rule.

A.2 Additive and multiplicative connectives

The logical rules defining the additive and multiplicative connectives are simply the linear rules for all modes, since we have separated out the structural rules. Except in one case, $\multimap L$, the well-formedness of the conclusion implies the well-formedness of all premises.

As for $\multimap L$, we know from the well-formedness of the conclusion that $\Psi \geq k$, $\Psi' \geq k$, and $m \geq k$. These facts by themselves already imply the well-formedness of the second premise, but we need to check that $\Psi' \geq m$ in order for the first premise to be well-formed.

A.3 Shifts

The shifts represent the most interesting aspects of the rules. Recall that in $\uparrow_k^m A_k$ and $\downarrow_k^m A_m$ we require that $m \geq k$. We first consider the two rules for $\uparrow$. We know from the conclusion of the right rule that $\Psi \geq m$ and from the requirement of the shift that $m \geq k$. Therefore, as $\geq$ is transitive, $\Psi \geq k$ and the premise is always well-formed. This also means (although we do not prove it here) that this rule is *invertible*.

From the conclusion of the left rule, we know $\Psi \geq \ell$, $m \geq \ell$, and $m \geq k$. This does not imply that $k \geq \ell$, which we need for the premise to be well-formed and thus needs to be checked. Therefore, this rule is non-invertible.

The downshift rules are constructed analogously, taking only the declaration of independence and properties of the preorder $\leq$ as guidance. Note that in this case the left rule is always applicable (that is, invertible), while the right rule is non-invertible.

A.4 Multicut

Because we have an explicit rule of contraction, cut elimination does not follow by a simple structural induction. However, we can follow Gentzen [10] and allow multiple copies of the same proposition to be removed by the cut, which then allows a structural induction argument. In anticipation of the operational interpretation, we have labeled our antecedents with unique variables, so the generalized form of cut called *multicut* (see, for example, [18]) can remove $n \geq 0$ copies. Of course, such cuts are only legal if the propositions that are removed satisfy the necessary structural rules. For $n = 0$, we require that the mode m support weakening.

$$\frac{\Psi \geq m \geq k \quad \mathsf{W} \in \sigma(m) \quad \Psi \vdash A_m \quad \Psi' \vdash C_k}{\Psi\ \Psi' \vdash C_k}\ \mathsf{cut}(\emptyset)$$

For $n = 1$, we obtain the usual cut rule and no special requirements are needed.

$$\frac{\Psi \geq m \geq k \quad \Psi \vdash A_m \quad (x : A_m)\ \Psi' \vdash C_k}{\Psi\ \Psi' \vdash C_k}\ \mathsf{cut}(\{x\})$$

For $n \geq 2$, the mode of the cut formula must admit contraction.

$$\frac{\begin{array}{l}\mathsf{C} \in \sigma(m) \\ \Psi \geq m \geq k \quad \Psi \vdash A_m \quad (S \cup \{x,y\} : A_m)\ \Psi' \vdash C_k\end{array}}{\Psi\ \Psi' \vdash C_k}\ \mathsf{cut}(S \cup \{x,y\})$$

Here, we have used the abbreviation $(\{x_1,\ldots,x_n\} : A_m)$ to stand for $(x_1 : A_m)\ldots(x_n : A_m)$.

Note that each of these rules has a side condition that can be interpreted informally as stating that the number of antecedents cut must be compatible with the mode m: if there are no antecedents removed, m must admit weakening, and if we remove two or more, m must admit contraction. This is exactly $|S| \sim m$ as defined in section 2.

This allows us to write down a single rule encompassing all three of the above cases for multicut:

$$\frac{\Psi \geq m \geq k \quad |S| \sim m \quad \Psi \vdash A_m \quad (S : A_m)\ \Psi' \vdash C_k}{\Psi\ \Psi' \vdash C_k}\ \mathsf{cut}(S)$$

Note that the standard cut rule is the instance of the multicut rule where $|S| = 1$, and so proving multicut elimination for adjoint logic also yields cut elimination for the standard cut rule.

A.5 Identity Expansion and Cut Elimination

We present standard identity expansion and cut elimination results as evidence for the correctness of the sequent calculus as capturing the meaning of the logical connectives via their inference rules. Cut-free proofs will always decompose propositions when read from conclusion to premise and thus yield a conservative extension result. Finally, the fine detail of the proof is significant because the cut reductions, which constitute the essence of the proof, are the basis for the operational semantics.

Theorem 3 (Identity Expansion)**.** *If $\Psi \vdash A_m$, then there exists a proof that $\Psi \vdash A_m$ using identity rules only at atomic propositions, which is cut-free if the original proof is.*

Proof. We begin by proving that for any formula A_m, there is a cut-free proof that $(x : A_m) \vdash A_m$ using identity rules only at atomic propositions. This follows easily from an induction on A_m.

Now, we arrive at the theorem by induction over the structure of the given proof that $\Psi \vdash A_m$. □

Theorem 4 (Cut Elimination)**.** *If $\Psi \vdash A_m$, then there is a cut-free proof of $\Psi \vdash A_m$.*

Proof. This proof follows the structure of many cut-elimination results. First we prove admissibility of multicut in the cut-free system. This is established by a straightforward nested induction, first on the proposition A_m and then simultaneously on the structure of the deductions $\mathscr{D}$ and $\mathscr{E}$. This is followed by a simple structural induction to prove cut elimination, using the admissibility of (multi)cut when it is encountered. If we ignore the modes, this proof is very similar to the original proof of Gentzen [10]. □

Corollary 2. *Adjoint logic is a conservative extension of each of the logics at a fixed mode. That is, if $\Psi \vdash A_m$ is a sequent purely at mode m (in that every type in Ψ is at mode m and neither A_m nor the types in Ψ make use of shifts), then $\Psi \vdash A_m$ is provable using the rules of adjoint logic iff it is provable using the rules which define the logic at mode m.*

A.6 Adjunction properties

As yet, we have not discussed the meaning of the name "*adjoint logic*". This can be justified by showing that for fixed $k \leq m$, $\downarrow_k^m$ and $\uparrow_k^m$ yield an adjoint pair of functors $\downarrow_k^m \dashv \uparrow_k^m$. Since prior results (see [2] and [16]) already establish this property and we have little new to contribute here, we omit the details here.

B Asynchronous Adjoint Logic

As has been observed before, intuitionistic and classical linear logics can be put into a Curry–Howard correspondence with session-typed communicating processes [4, 27, 5]. A linear logical proposition corresponds to a session type, and a sequent proof to a process expression. The transition rules of the operational semantics derive from the cut reductions.

Under the intuitionistic interpretation a sequent proof[3] of

$$(x_1 : A_{\mathsf{L}}^1) \cdots (x_n : A_{\mathsf{L}}^n) \vdash (x : A_{\mathsf{L}})$$

corresponds to a process P that *provides* channel x and uses channels x_i. The types of the channels prescribe the pattern of communication: in the succedent, positive types ($\oplus, \otimes, \mathbf{1}$) will send and negative types ($\&, \multimap$) will receive. In the antecedent, the roles are reversed. Cut corresponds to parallel composition of two processes, with a private channel between them, while identity simply equates two channels.

B.1 Enforcing Asynchronous Communication

Under this interpretation, a cut of a right rule against a matching left rule allows computation to proceed by mimicking the cut reduction from the proof of Theorem 4. For example, a cut at type $\bigoplus_{i \in I} A_{\mathsf{L}}^i$ is replaced by a cut at type A_{L}^ℓ for some $\ell \in I$. This corresponds to passing a message ('ℓ') from the process *providing* $x : \bigoplus_{i \in I} A_{\mathsf{L}}^i$ to the process *using* x. By its very nature, this form of cut reduction is *synchronous*: both provider and client proceed simultaneously because the channel $x : A_\ell$ connects the two process continuations.

For realistic languages, and also for the paradigm to smoothly extend to the case of adjoint logic where some modes permit weakening and contraction, we would like to prescribe *asynchronous communication* instead.

We observe that 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 similar effect.

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

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

[3] for now on the linear fragment, and also labeling the succedent with a fresh variable

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

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

and in the other direction we use identity

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

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:

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

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 (section 4) for the corresponding process calculus even in the presence of recursive types and processes, which is ultimately what we care about for the resulting concurrent programming language.

B.2 Eliminating Weakening and Contraction

We have introduced multicut entirely with the standard motivation of providing a simple proof of the admissibility of cut using structural induction. Surprisingly, we can streamline the system further by using multicut to eliminate weakening and contraction from the logic altogether, as in the system we use as the basis for our typing rules (fig. 1).

Consider a mode m with $\mathsf{C} \in \sigma(m)$. Then contraction is a simple instance of multicut with an instance of the identity rule.

$$\dfrac{\dfrac{}{(x : A_m) \vdash A_m}\,\mathsf{id} \quad \Psi\ (y : A_m)\ (z : A_m) \vdash C_k}{\Psi\ (x : A_m) \vdash C_k}\,\mathsf{cut}(\{y, z\})$$

Similarly, for a mode m with $\mathsf{W} \in \sigma(m)$, weakening is also an instance of multicut.

$$\frac{\dfrac{}{(x : A_m) \vdash A_m}\ \mathsf{id} \qquad \Psi \vdash C_k}{\Psi\ (x : A_m) \vdash C_k}\ \mathsf{cut}(\emptyset)$$

Cut reductions in the presence of contraction entail many residual contractions, as is evident already from Gentzen's original proof. Under our interpretation of contraction above, these residual contractions simply become multicuts with the identity. The operational interpretation of identities then plays three related roles: with one client, an identity achieves a renaming, redirecting communication; with two or more clients, an identity implements copying; with zero clients, its effect is cancellation or garbage collection. The central role of identities can be seen in full detail in Figure 2, once we have introduced our notation for processes and process configurations.

C Program Examples

In the examples that follow, we will work with two modes, L and U, with $\mathsf{L} < \mathsf{U}$, $\sigma(\mathsf{L}) = \emptyset$, and $\sigma(\mathsf{U}) = \{W, C\}$. In these examples we also use recursively defined types and processes without formally defining these constructs, since they are well-known from the literature and orthogonal to our concerns (see, for example, [26]).

C.1 Example: Circuits

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

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

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

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

This is somewhat verbose, but note that all channels here are shared. For this particular gate they could also be linear because they are neither reused nor canceled. This illustrates that programming can be uniform at different modes, which is a significant advantage of our system over systems of session types based on 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*:

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

This multi-step reduction is shown in full (one step at a time) below. We only show the initial portion of each process term, which is enough to disambiguate where in the program we are, as otherwise process terms become unwieldy and reduce clarity. We also assume the existence of a rule call that lets us invoke a defined process, replacing the call with the process definition, after appropriate substitution. At each step, we have highlighted in red the process(es) that are about to transition.

$$\begin{aligned}
&\mathsf{proc}(\{a\},\{a'\},a,a.A(a')),\mathsf{proc}(\{b\},\{b'\},b,b.B(b')),\mathsf{proc}(S,\{a,b\},c,c \leftarrow nor \leftarrow a,b) && \overset{\mathsf{call}}{\Longrightarrow}\\
&\mathsf{proc}(\{a\},\{a'\},a,a.A(a')),\mathsf{proc}(\{b\},\{b'\},b,b.B(b')),\mathsf{proc}(S,\{a,b\},c,\mathsf{case}\ a\ldots) && \overset{\oplus C}{\Longrightarrow}\\
&\mathsf{proc}(\{b\},\{b'\},b,b.B(b')),\mathsf{proc}(S,\{a',b\},c,\mathsf{case}\ b\ldots) && \overset{\oplus C}{\Longrightarrow}\\
&\mathsf{proc}(S,\{a',b'\},c,z' \leftarrow \ldots) && \overset{\mathsf{cut}(\{z'\})}{\Longrightarrow}\\
&\mathsf{proc}(\{c'\},\{a',b'\},z',z' \leftarrow nor \leftarrow a',b'),\mathsf{proc}(S,\{c'\},c,c.C(c'))
\end{aligned}$$

When we build an or-gate out of a nor-gate we need to exploit sharing to implement simple negation. In the example below, u and u' are both names for the same shared channel. The process invoked as $nor \leftarrow x,y$ will multicast a message to the clients of u and u'.

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

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

An analogous computation to the above is possible, except that at an intermediate stage of the computation, we will also have a shared channel d carrying the (multicast) message $\mathsf{proc}(\{u,u'\},\{d'\},d,d.D(d'))$ with $D = \neg(A \vee B)$.

C.2 Example: Map

Mapping a process over a list allows us to demonstrate the use of replicable services, as well as cancellation. 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 at mode L.

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

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

In this example, f_{U} is a replicable and cancelable service. In the case of a nonempty list, we create two names for the channel f_{U} — one to use immediately and one to pass to the recursive call. Note that the service itself remains a single service with two clients until the message $\mathsf{shift}(f')$ is sent to it, at which point it replicates itself, creating one copy to handle this request and leaving another to deal with future requests. In the case of an empty list, we have no elements to map over, and so we do not need to use f_{U}. As such, we cancel it before continuing.