A Message-Passing Interpretation of Adjoint Logic

Klaas Pruiksma[a,*], Frank Pfenning[a]

[a]*Computer Science Department, Carnegie Mellon University. 5000 Forbes Ave., Pittsburgh, PA 15213, United States*

Abstract

We present a system of session types based on *adjoint logic* which generalizes standard binary session types. 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 [1] were designed to specify communication protocols between two processes along a single private channel between them. As the parties proceed through the protocol with complementary send/receive actions, the type of the channel evolves accordingly. At first, it may seem that this prohibits the interpretation of types as propositions of a logic, but with careful bookkeeping we can exploit the use-once semantics of *linear logic* to capture this behavior exactly [2]. Proofs in the linear sequent calculus then correspond to processes, and synchronous communication to cut reduction steps. Under this interpretation, the exponential modality $!A$ of linear logic corresponds a replicable service, its proofs to replication $!P$ in the π-calculus. It thereby extends the strict binary nature of channels because there may be multiple clients for a channel providing a replicable service.

In this paper we reformulate and extend these correspondences between session types and logic to uniformly capture additional mechanisms of communication: asynchronous communication, cancellation, and multicast. For *asynchronous communication* [3], we exploit earlier observations on *use-once* and *continuation channels* [4, 5]. We accomplish this by a reformulation of the sequent calculus in *semi-axiomatic* form [6]. A second extension is *cancellation*, which arises when we consider session types (and the corresponding propositions) to be *affine* rather than *linear* [7, 8, 9, 10]. Logically, affine hypotheses may be used *at most once*. To allow both linear *and* affine types at the programmer's discretion [11], we move from linear logic to *adjoint logic* [12, 13, 14] (and from linear types to adjoint types) which supports the modular combination of several logics with varying structural properties. The generalization to adjoint types enables our third generalization: A single message may be sent to multiple recipients in a *multicast*. To our knowledge, this has not been previously considered in conjunction with static typing of communication protocols.

Our main contribution, then, is a single uniform type system corresponding to a single uniform logic, in which all of the mentioned patterns of message-passing communication (asynchronous communication, cancellation, multicast, and replicable services) emerge straightforwardly and coexist harmoniously. Naturally, our system includes the usual binary session types, and even synchronous communication can be expressed easily, both operationally and logically [11] (which stands in contrast to the untyped setting [15]). We prove the standard properties of *session fidelity* (well-typed processes will adhere to the specified communication protocols) and *deadlock freedom* (well-typed configurations of communicating processes will not "get stuck").

*Corresponding author

As a secondary contribution we show that the substructural nature of adjoint logic together with our specific form of cancellation eliminates the need for distributed garbage collection.

We now briefly motivate the various patterns and choices made in our approach.

Asynchronous communication seems like the right primitive for three reasons: (1) it is consistent with message-passing implementations in actual distributed systems, (2) synchronous communication is easily modeled by acknowledgments specified via logical modalities ("double shifts") [11], and (3) its meaning extends immediately to patterns such as cancellation or multicast for which synchronous communication may not be easily defined or understood.

Examples for *replicable services* abound, including web servers, service-oriented architectures, or microservices. *Cancellation* also arises frequently, often in tandem with exceptions [7, 8, 9, 10]. Their combination allows the programmer to choose between services they expect must be used and those that might not be used and obtain suitable *static* error messages if these patterns are violated. Studies of message-passing concurrency in real-world Go programs [16] suggest that some errors could be detected in our expressive type system.

Multicast arises in circuits and pipelines where the output of one node (modelled as a process) is used as input to several others. It also can be used as a proxy for *broadcast* if the set of recipients is known, which is the case in many distributed algorithm, for example, for leader election or consensus.

We begin with an overview of adjoint logic (Section 2), which we revise in Section 3 to a form that is better suited to our operational semantics. We then present our type system and the syntax of our language (Section 4) before providing our first major contribution: the operational semantics for the language (Section 5). Our semantics models a variety of asynchronous communication behaviors, uniformly generalizing several previous systems. We provide some examples entirely within our system that demonstrate several of these patterns, and then in Section 6, we step outside of the system, adding recursion in order to write some more computationally complex examples. Finally, we present our results on session fidelity and deadlock-freedom, as well as freedom from the need for garbage collection, which follows as a corollary (Section 7). [1]

2. Adjoint Logic

We present here a brief overview of the formulation of adjoint logic [12, 14], that we take as a basis for the semantics presented in Section 4. The reader uninterested in details of logic may wish to jump forward to that section.

Adjoint logic is a schema to define particular logics. The schema is parameterized by three data:

1. A set $\mathcal{M}$ of modes of truth m, which are used to index propositions and logical connectives.
2. A map $\sigma : \mathcal{M} \to \mathcal{P}(\{\mathsf{W}, \mathsf{C}\})$ assigning to each mode a set of structural properties, where W stands for *weakening* and C stands for *contraction*. Here, we always allow exchange as a concession to simplicity of the presentation.
3. A preorder $\geq$ on $\mathcal{M}$, where $m \geq k$ expresses that the proof of a proposition of mode k may depend on a hypothesis of mode m. 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 for the resulting logic.

In addition, we sometimes consider subsets of the available connectives to obtain a fragment of the logic (as, for example, [11]).

The preorder embodies a key property of adjoint logic, the *declaration of independence*:

[1] This paper is an extended and revised version of a contribution presented at the *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software* (PLACES 2019) [17]. It provides further examples, more discussion of properties of the *shift* modalities, more details on garbage collection, and additional discussion of related work. Additionally, this paper contains material that was relegated to the appendix of the previous version.

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

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

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

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

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

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

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

Figure 1: Rules of Adjoint Logic

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

We emphasize independence here (rather than the allowable dependence on B_m for $m \geq k$) because while a proof *may* depend on some B_m, it *must not* depend on any B_ℓ where $\ell \not\geq k$.

Sequents in adjoint logic have the form:

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

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. The critical presupposition that each $m_i \geq k$ (a restatement of the declaration of independence) is abbreviated as $\Psi \geq k$. Furthermore, the order of the antecedents does not matter since we always allow exchange. We label the antecedents with variables so we can track the fine structure of proofs, which will be important for our operational semantics in Section 5.

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

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

Here p_m stands for atomic propositions at mode m. Most of the connectives are those of intuitionistic linear logic, annotated with modes. Due to the needs of our operational interpretation, we generalize internal and external choice to n-ary constructors parameterized by a finite index set I, recovering the binary choice with $I = \{\pi_1, \pi_2\}$.

Adjoint logic adds two modal operators (*shifts*) to the linear connectives. 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 $\ell \geq m$. Note that these connectives are the only ones that move between modes — all other connectives operate entirely within a single mode 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.

2.1. Judgmental and structural rules

The rules for adjoint logic can be found in Figure 1, in a standard sequent calculus form with explicit rules of weakening and contraction. When we come to the operational semantics (Section 4), we will use a

slightly different formulation (briefly discussed in Section 3).

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.

2.2. 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.

2.3. Shifts

The shifts represent the most interesting aspects of the rules. Recall that in $\uparrow_k^m A_k$ and $\downarrow_m^\ell A_\ell$ we require that $m \geq k$ and $\ell \geq m$. 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 $\geq$ as guidance. Note that in this case the left rule is invertible, while the right rule is not.

A simple but important property of the shifts is that they are well-behaved with respect to the preorder — in particular, we have the following reflexivity and transitivity properties (using the notation $\dashv\vdash$ for interprovability):

$$\uparrow_k^k A_k \dashv\vdash A_k \qquad\qquad \uparrow_m^\ell \uparrow_k^m A_k \dashv\vdash \uparrow_k^\ell A_k$$
$$\downarrow_k^k A_k \dashv\vdash A_k \qquad\qquad \downarrow_k^m \downarrow_m^\ell A_\ell \dashv\vdash \downarrow_k^\ell A_\ell$$

Using these properties, we are able to combine any sequence of up-shifts into either a single shift or no shifts (from a purely logical perspective), and similarly for down-shifts. Note, however, that we cannot combine up-shifts with down-shifts — sequences of up and down can yield modalities that are not captured by just a single shift.

2.4. Multicut

Because we have an explicit rule of contraction, cut elimination does not follow by a simple structural induction. However, we can follow Gentzen [18] 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, [19]) 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.

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

Observe that since $\Psi \geq m$ and $\mathsf{W} \in \sigma(m)$, each antecedent in Ψ must also admit weakening.

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.

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

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. To formulate multicut as a single rule, we define the *multiplicities of* m ($\mu(m) \subseteq \mathbb{N}$)

$$\mu(m) = \{n \mid (n = 0 \wedge \mathsf{W} \in \sigma(m)) \vee n = 1 \vee (n \geq 2 \wedge \mathsf{C} \in \sigma(m))\}$$

and summarize

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

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.

2.5. 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 1 (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 2 (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 $\mathcal{D}$ and $\mathcal{E}$. 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 [18]. □

Corollary 1. *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.*

2.6. 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 [20] and [21]) already establish this property and we have little new to contribute here, we omit the details.

3. The Semi-Axiomatic Sequent Calculus

As has been observed before, intuitionistic and classical linear logics can be put into a Curry–Howard correspondence with session-typed communicating processes [2, 22, 23]. 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.

For the moment, we restrict ourselves to a single mode L with no structural properties ($\sigma(\mathsf{L}) = \{\}$). Each proposition in this instance of adjoint logic corresponds to a *binary* session type, where a proof of the sequent[2]

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

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 equates two channels.

3.1. 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 2. 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 (and omitting variables, for brevity), they become

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

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

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

and in the other direction we use identity

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

[2] also labeling the succedent with a fresh variable

to derive the other rules.

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

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

We see the cut reduction completely eliminates the cut in one step, which corresponds precisely to receiving a message. In this example the message would be π_1 since the axiom $\oplus R_1^0$ was used; for $\oplus R_2^0$ 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. For reference, the rules are those in Figure 2 ignoring the process expressions given in red.

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 except cut matches this conclusion. Fortunately, we can still prove an analogue of cut elimination in which all remaining cuts (called *snips*) satisfy the subformula property. This has been explored in some depth by DeYoung and the authors [6] in a calculus called Sax for (nonlinear) intuitionistic logic. We believe these properties generalize to the setting of adjoint logic, but we do not explore it further here.

Perhaps more importantly, we have session fidelity and deadlock freedom (Section 7) 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.

3.2. 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, which will be convenient when we look to use the logic as a type system (Figure 2).

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.

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

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. The central role of identities can be seen in full detail in Figure 3, once we have introduced our notation for processes and process configurations.

4. Language and Typing

We now introduce our language of process expressions, typed with propositions from adjoint logic as explained in Section 2 and adapted to asynchronous communication in Section 3. We review some of the basic logical components from the viewpoint of types so that this section is more self-contained.

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., [2]) 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 [22] 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 (see Section 2). Each mode m is assigned a set of structural properties $\sigma(m)$ among W (for weakening) and C (for contraction). Separating m from $\sigma(m)$ allows us to have multiple modes with the same set of structural properties.[3] 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 $\mathsf{W} \in \sigma(m)$ may not have any clients and a channel x_m with $\mathsf{C} \in \sigma(m)$ 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 $\Psi \vdash P :: (x_k : A_k)$ it must be the case that for every $y_m : B_m \in \Psi$ we have $m \geq k$. For example, if k permits contraction and therefore P may have multiple clients, then for any y_m in Ψ, mode m must also permit contraction because (intuitively) if x_k is referenced multiple times then, indirectly, so is y_m. If $m \geq k$ then this is ensured. We express this with the *presupposition* that

$$\Psi \vdash P :: (x_k : A_k) \quad \text{requires} \quad \Psi \geq k$$

where $\Psi \geq k$ simply means $m \geq k$ for every $y_m : A_m \in \Psi$. 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.

The complete set of rules for the typing judgment are given in Figure 2, and we informally describe the meanings of process expressions in Table 1. When we ignore the process expressions we obtain the sequent calculus for adjoint logic in its semi-axiomatic presentation (see Section 3). 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 ($\mathsf{C} \in \sigma(m)$) or no clients ($\mathsf{W} \in \sigma(m)$). We therefore generalize it to a form of multicut[4], where the channel x_m provided by P is

[3] 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.

[4] The term "multicut" has been used in the literature for different rules. We follow here the proof theory literature [19, 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 [18], where he calls it "Mischung".

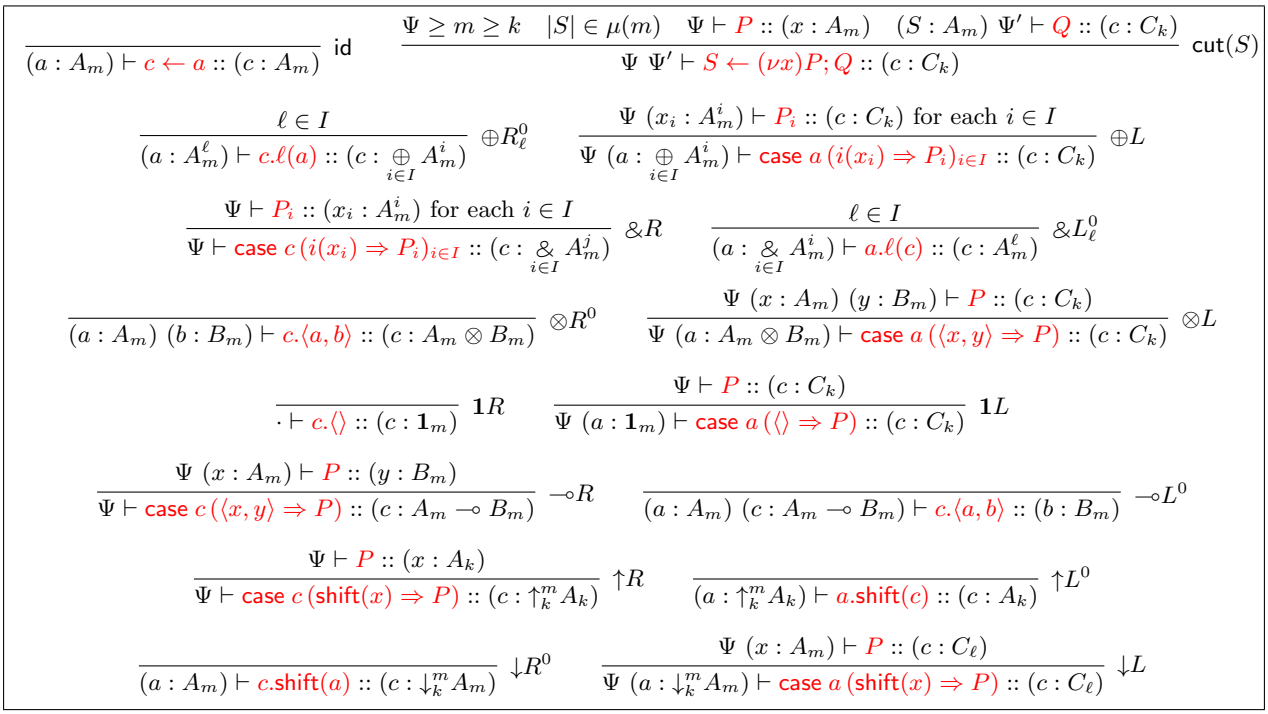

Figure 2: Process Assignment for Adjoint Logic in Semi-Axiomatic Form

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 $|S| \in \mu(m)$ as defined in Section 2. Briefly, $0 \in \mu(m)$ if m admits weakening, $1 \in \mu(m)$ (always), and $i \in \mu(m)$ for $i \geq 2$ if m admits contraction. 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 5).

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

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

Here, p_m stands for atomic propositions at mode m. The other connectives, besides the shifts $\uparrow^m_k$ 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, which is convenient for programming. We will use a label set $I = \{\pi_1, \pi_2\}$ when working with the binary forms $A_m \mathbin{\&} B_m$ and $A_m \oplus B_m$, where π_1 selects the left-hand type and π_2 selects the right-hand type (see, for instance, Example 2). The operational meaning of these connectives (discussed further in Section 5) is largely similar to that in past work (e.g., [2]), 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 $m \leq \ell$. From the perspective of the provider, an up shift then corresponds to a *suspension* (or *quotation*) of its continuation, while the shift message from the client *forces* (or *evaluates*) it. The restriction of the structural properties of the modes prohibit a process of mode k (that is, providing a channel $c_k : A_k$) to be evaluated inside a process of mode m. This is important for the soundness of the operational semantics. For example, assume we have affine and linear modes, F and L, respectively (so that $\mathsf{F} > \mathsf{L}$, $\sigma(\mathsf{F}) = \{\mathsf{W}\}$, and $\sigma(\mathsf{L}) = \{\,\}$). Then an affine process P may be cancelled

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

(by its only client) while a linear process Q may not. But if Q were allowed to execute, providing a linear channel used by P, then the cancellation of P would (implicitly or explicitly) also cancel Q, thereby violating linearity. In combination, the shifts model comonads ($\Box A = \downarrow\uparrow A$) and strong monads ($\Diamond A = \uparrow\downarrow A$), with specific meanings depending on the modes involved.

Remarkably, the operational meaning of all connectives is again uniform, independent of the mode and the structural properties it may admit. For example, sending a label along a linear or affine channel with two endpoints is the same as a multicast. The differences are concentrated in the rules of identity and cut, as explained in more detail in Section 5.

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 [24], the main revisions are that (1) we make channel continuations explicit in order to facilitate asynchronous communication while preserving message order [5], and (2) we distinguish between an *internal name* for the channel provided by a process and *external names* connecting it to multiple clients.

We close the section with some small examples with their types; additional examples which highlight more interesting behavior can be found in Section 6.

First, we have process that witnesses that $\otimes_m$ is commutative at an arbitrary mode.

Example 1. *At any mode m,*

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

The following process exemplifies how messages are represented by small processes ($y.\pi_2(w)$ and ($y.\pi_1(w)$), and how the intuitive operation of sending a message corresponds to a cut.

Example 2. *At any mode m,*

$$\begin{array}{l}(x : A_m \oplus B_m)\,(y : A_m \,\&\, B_m) \vdash P :: (z : A_m \oplus B_m)\\ P \triangleq \mathsf{case}\ x\ \ (\,\pi_1(x') \Rightarrow \{y'\} \leftarrow (\nu w)\,y.\pi_2(w);\\ \qquad\qquad\qquad\qquad z.\langle x', y'\rangle\\ \qquad\qquad |\,\pi_2(x') \Rightarrow \{y'\} \leftarrow (\nu w)\,y.\pi_1(w);\end{array}$$

$z.\langle y', x' \rangle$
)

Our next few examples highlight the differences between modes. The following process witnesses that $A_m \,\&\, B_m$ proves $A_m \otimes B_m$ in the presence of contraction. '%' starts a comment.

Example 3. *For any mode m admitting contraction (i.e., $\mathsf{C} \in \sigma(m)$),*

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

The converse entailment holds in the presence of weakening.

Example 4. *For any mode m admitting weakening (i.e., $\mathsf{W} \in \sigma(m)$),*

$(x : A_m \otimes B_m) \vdash P :: (p : A_m \,\&\, B_m)$
$P \triangleq \mathsf{case}\ p\ \ (\,\pi_1(p_1) \Rightarrow \mathsf{case}\ x\,(\langle y, z \rangle \Rightarrow$
$\{\,\} \leftarrow (\nu a)\,(a \leftarrow z);$ % $\mathsf{drop}\ z$
$p_1 \leftarrow y)$
$|\,\pi_2(p_2) \Rightarrow \mathsf{case}\ x\,(\langle y, z \rangle \Rightarrow$
$\{\,\} \leftarrow (\nu a)\,(a \leftarrow y);$ % $\mathsf{drop}\ y$
$p_2 \leftarrow z\,))$

Examples 3 and 4 show that we can copy and drop propositions provided that their modes allow it. Moreover, we can write processes which can copy and drop *shifted* propositions (e.g. $\downarrow_k^m \uparrow_k^m A_k$) as well, provided only that mode m allows it. In particular, we do not need any condition on the mode k. We can therefore use the shifts to allow a proposition at mode k to behave in some ways like one at mode m. We can use this to model the exponential modality $!A$ of linear logic with two modes, U with $\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$ and L with $\sigma(\mathsf{L}) = \{\,\}$ and $\mathsf{U} > \mathsf{L}$. Then $!A$ corresponds to $\downarrow_{\mathsf{L}}^{\mathsf{U}} \uparrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{L}}$, which means that two messages will be exchanged when moving between shared and linear channels. The direct treatment of $!A$ in prior work [2, 22, 23] is less elegant because it is the only connective requiring two messages in a somewhat asymmetric fashion.

As an example, a considerable generalization of the linear logic equivalence $!(A \,\&\, B) \dashv\vdash (!A) \otimes (!B)$ is then expressed by the following two properties.

- If $k \leq m$ and m admits contraction, then, *whether k admits contraction or not,*

$$\downarrow_k^m \uparrow_k^m (A_k \,\&\, B_k) \vdash (\downarrow_k^m \uparrow_k^m A_k) \otimes_k (\downarrow_k^m \uparrow_k^m B_k)$$

- If $k \leq m$ and m admits weakening, then, *whether k admits weakening or not,*

$$(\downarrow_k^m \uparrow_k^m A_k) \otimes_k (\downarrow_k^m \uparrow_k^m B_k) \vdash \downarrow_k^m \uparrow_k^m (A_k \,\&\, B_k)$$

We will not show these processes explicitly, as they are similar in their overall structure to the previous examples, but require a significant amount of bookkeeping to handle the shifts.

Throughout our next series of examples, we will use comments after (almost) each line of a process to describe the channels (and their session types) that are in scope at that point in the program. We omit these comments after lines which terminate computation, as after these lines, there are no channels left in scope.

The following process witnesses that down shifts distribute over implication.

Example 5. *For modes $k \leq m$ with arbitrary structural properties,*

$(f : \downarrow_k^m (A_m \multimap_m B_m)) \vdash P :: (g : \downarrow_k^m A_m \multimap_k \downarrow_k^m B_m)$
$P \triangleq \mathsf{case}\ g\ (\ \langle x, y\rangle \Rightarrow$ % $(f : \downarrow_k^m (A_m \multimap_m B_m)), (x : \downarrow_k^m A_m) \vdash (y : \downarrow_k^m B_m)$
$\quad \mathsf{case}\ f\ (\ \mathsf{shift}(w) \Rightarrow$ % $(w : A_m \multimap_m B_m), (x : \downarrow_k^m A_m) \vdash (y : \downarrow_k^m B_m)$
$\quad \mathsf{case}\ x\ (\ \mathsf{shift}(v) \Rightarrow$ % $(w : A_m \multimap_m B_m), (v : A_m) \vdash (y : \downarrow_k^m B_m)$
$\quad \{z\} \leftarrow (\nu z)\, y.\mathsf{shift}(z);$ % $(w : A_m \multimap_m B_m), (v : A_m) \vdash (z : B_m)$
$\quad w.\langle v, z\rangle\ ))$

A very similar process shows that upwards shifts distribute over implication. However, we omit this processes here, only showing the sequent which it witnesses, due to this similarity:

$$(f : \uparrow_k^m (A_k \multimap_k B_k)) \vdash (g : \uparrow_k^m A_k \multimap_m \uparrow_k^m B_k)$$

In fact, every well-formed sequence of shifts distributes over implication. We make this precise with the following theorem:

Theorem 3. *Suppose we are given modes k and m, and that $\Updownarrow_k^m$ is a sequence of shifts from k to m:*

$$\Updownarrow_k^m A_k ::= \Updownarrow_\ell^m \uparrow_k^\ell A_k \ \mid\ \Updownarrow_\ell^m \downarrow_\ell^k A_k \ \mid\ A_k$$

where the last case is only permitted if $k = m$.

Then, for any A_k, B_k, $\Updownarrow_k^m (A_k \multimap_k B_k) \vdash (\Updownarrow_k^m A_k) \multimap_m (\Updownarrow_k^m B_k)$.

This theorem follows from a simple induction over the structure of $\Updownarrow_k^m$. The resulting process looks much like the example shown, except that each additional shift adds three communication steps between the initial case and the final send, and the precise nature of those steps depends on whether the shift is an up shift or a down shift.

The final example highlights the symmetry between & and $\oplus$ from the intuitionistic perspective. We will revisit this later example in Section 5, where we will discuss the computation that these examples perform. We provide witnesses that $(x : (A_m \oplus B_m) \multimap C_m) \dashv\vdash (y : (A_m \multimap C_m)\ \&\ (B_m \multimap C_m))$.

Example 6. *For any mode m,*

$(x : (A_m \oplus B_m) \multimap C_m) \vdash P :: (y : (A_m \multimap C_m)\ \&\ (B_m \multimap C_m))$
$P \triangleq \mathsf{case}\ y\ \ (\ \pi_1(ac) \Rightarrow$ % $(x : (A_m \oplus B_m) \multimap C_m) \vdash (ac : A_m \oplus C_m)$
$\quad \mathsf{case}\ ac\ (\ \langle a, c\rangle \Rightarrow$ % $(x : (A_m \oplus B_m) \multimap C_m), (a : A_m) \vdash (c : C_m)$
$\quad \{ab\} \leftarrow (\nu ab)ab.\pi_1(a);$ % $(x : (A_m \oplus B_m) \multimap C_m), (ab : A_m \oplus B_m) \vdash (c : C_m)$
$\quad x.\langle ab, c\rangle)$
$\quad \mid\ \pi_2(bc) \Rightarrow$ % $(x : (A_m \oplus B_m) \multimap C_m) \vdash (bc : B_m \multimap C_m)$
$\quad \{ab\} \leftarrow (\nu ab)ab.\pi_2(b);$ % $(x : (A_m \oplus B_m) \multimap C_m), (ab : A_m \oplus B_m) \vdash (c : C_m)$
$\quad x.\langle ab, c\rangle))$

And in the reverse direction:

$(y : (A_m \multimap C_m)\ \&\ (B_m \multimap C_m)) \vdash Q :: (x : (A_m \oplus B_m) \multimap C_m)$
$Q \triangleq \mathsf{case}\ x\ (\ \langle ab, c\rangle \Rightarrow$ % $(y : (A_m \multimap C_m)\ \&\ (B_m \multimap C_m)), (ab : A_m \oplus B_m) \vdash (c : C_m)$
$\quad \mathsf{case}\ ab\ \ (\pi_1(a) \Rightarrow$ % $(y : (A_m \multimap C_m)\ \&\ (B_m \multimap C_m)), (a : A_m) \vdash (c : C_m)$
$\quad\quad \{ac\} \leftarrow (\nu ac)y.\pi_1(ac);$ % $(ac : (A_m \multimap C_m)), (a : A_m) \vdash (c : C_m)$
$\quad\quad ac.\langle a, c\rangle$
$\quad\quad \mid\ \pi_2(b) \Rightarrow$ % $(y : (A_m \multimap C_m)\ \&\ (B_m \multimap C_m)), (b : B_m) \vdash (c : C_m)$
$\quad\quad \{bc\} \leftarrow (\nu bc)y.\pi_2(bc);$ % $(bc : (B_m \multimap C_m)), (b : A_m) \vdash (c : C_m)$
$\quad\quad bc.\langle b, c\rangle))$

5. Operational Semantics

In order to formally define 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 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 (here a) of a process has no effect on its interactions with other processes. We write processes in this order for practical reasons — the process terms P are the longest part of a running process in typical use cases, and so by placing this last in the tuple, we can both easily abbreviate process terms and easily see the first three elements of a tuple, even when P is very long.[5]

A *process configuration* is a multiset of processes:

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

where we require that all the aliases or names provided by the processes $\mathsf{proc}(S, \Delta, a, P)$ are distinct, i.e., given two 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* [25]. 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 $I \neq \emptyset$. To see why, consider the following process:

$$\Gamma \vdash \mathsf{case}\ x\ () :: (x : \mathop{\&}_{i \in \emptyset} A^i_m)$$

This process uses all channels in Γ, while providing x, but has only x as a free channel. Similar processes can be written using $\bigoplus_{i \in \emptyset} A^i_m$ as well. 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 typing judgment for a configuration $\mathcal{C}$ has the form $\Psi \vDash \mathcal{C} :: \Psi'$ which expresses that using the channels in Ψ, configuration $\mathcal{C}$ provides the channels in Ψ'. This allows a channel that is not mentioned at all in $\mathcal{C}$ to appear in both Ψ and Ψ'— we think of such a channel as being "passed through" the configuration. We define this judgment with the following rules:

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

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 Proc rule: 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 $|S| \in \mu(m)$, as defined in Section 2.4. The identity (Id) and composition (Comp) 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 $\mathcal{C}$ is used by $\mathcal{C}'$.

[5] A side benefit of placing the process term last is that when we look at a running process $\mathsf{proc}(S, \Delta, a, P)$, we first see the interface of the process — it provides channels S and uses channels in Δ. We then see the internal name a, allowing us to read the process term P with the full context of which channels it uses and provides (Δ and a, respectively).

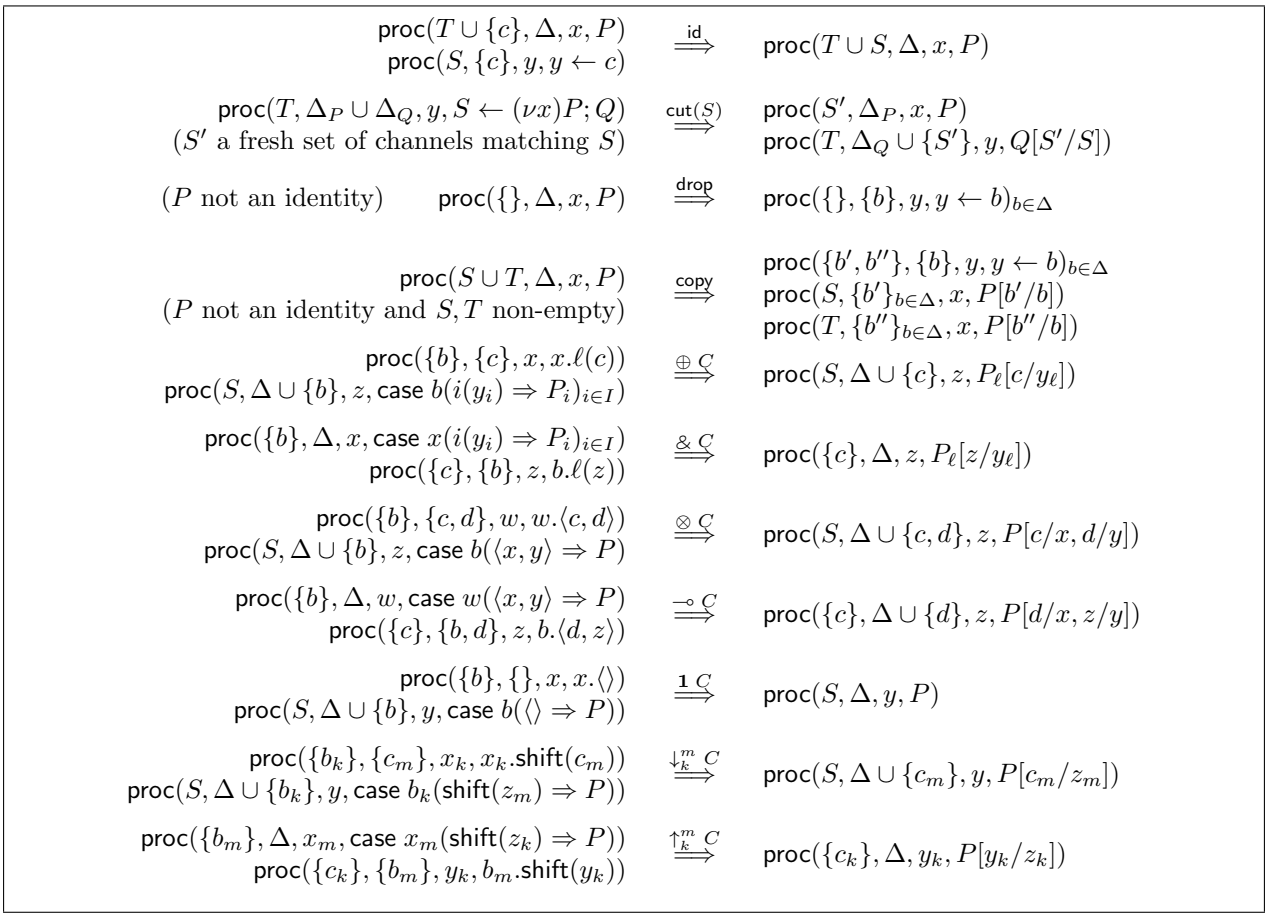

Figure 3: Computation Rules for Asynchronous Adjoint Logic

The computation rules we discuss in this section can be found in Figure 3. 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.

5.1. 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.

5.2. 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, 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 x 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 $\stackrel{\mathsf{copy}}{\Longrightarrow}$). 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 x 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 no 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. We therefore disallow this use of the copy and drop rules to ensure that we can prove a progress theorem. [6]

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).

5.3. Additive and multiplicative connectives

In the rule for $\oplus$, the process $\mathsf{proc}(\{b\}, \{c\}, x, x.\ell(c))$ represents the message 'label ℓ with continuation channel 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 y_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 channel, and so an extra substitution is required when receiving messages. $\mathbf{1}$ 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.

5.4. 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. Example 10 demonstrates how multiple modes, connected with the shifts, can be used to accomplish something useful that cannot be expressed with a single mode — in this case, mapping a *linear* function over a list of elements, which may require replicating or cancelling the function.

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.

5.5. Some Examples, Operationally

We now reexamine two examples in light of the operational interpretation. The first one is reminiscent of a simple (non-recursive) instance of the visitor pattern (as described by Palsberg and Jay [26], for example) of object-oriented programming.

[6] While there are cut reductions in the logic that correspond to applying cut and drop to identities, when giving our semantics, we impose an evaluation strategy by restricting which cut reductions are allowed. Similarly, the so-called "commuting" reductions are not used as evaluation rules in the semantics.

Example 7 (Revisiting Example 6)**.** *For any mode m,*

$$
\begin{array}{l}
(y : (A_m \multimap C_m) \mathbin{\&} (B_m \multimap C_m)) \vdash P :: (x : (A_m \oplus B_m) \multimap C_m) \\
P \triangleq \mathsf{case}\ x\ (\ \langle ab, c\rangle \Rightarrow \qquad\qquad \% \ (y : (A_m \multimap C_m) \mathbin{\&} (B_m \multimap C_m)), (ab : A_m \oplus B_m) \vdash (c : C_m) \\
\quad \mathsf{case}\ ab\ \ (\ \pi_1(a) \Rightarrow \qquad\qquad \% \ (y : (A_m \multimap C_m) \mathbin{\&} (B_m \multimap C_m)), (a : A_m) \vdash (c : C_m) \\
\qquad\qquad \{ac\} \leftarrow (\nu ac) y.\pi_1(ac); \quad \% \ (ac : (A_m \multimap C_m)), (a : A_m) \vdash (c : C_m) \\
\qquad\qquad ac.\langle a, c\rangle \\
\qquad\quad \mid \pi_2(b) \Rightarrow \qquad\qquad \% \ (y : (A_m \multimap C_m) \mathbin{\&} (B_m \multimap C_m)), (b : B_m) \vdash (c : C_m) \\
\qquad\qquad \{bc\} \leftarrow (\nu bc) y.\pi_2(bc); \quad \% \ (bc : (B_m \multimap C_m)), (b : A_m) \vdash (c : C_m) \\
\qquad\qquad bc.\langle b, c\rangle))
\end{array}
$$

If we think of C as an operation to be done, then $x : (A_m \oplus B_m) \multimap C_m$ is a simple visitor which performs the operation (or prepares it to be performed) on either As or Bs. This process then shows how a visitor object can be implemented "piecewise", by specifying how to create a suitable C given an A, and separately, how to create a C from a B. We can then think of $y : (A_m \multimap C_m) \mathbin{\&} (B_m \multimap C_m)$ as an object providing these two methods, and the process above gives us a way to build a visitor out of its component pieces.

We now revisit our example of weakening in order to highlight how cancellation propagates through a configuration of processes.

Example 8 (Revisiting Example 4)**.** *For any mode m admitting weakening,*

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

We will (loosely) step through one possible evaluation of this program. We will be precise about the steps taken by the process P, but will take some liberties with the rest of the configuration — in particular, we will only show the process object corresponding to P, as well as process objects which are imminently relevant to P. This allows us to focus on the computation being done by P. At each step, we have highlighted in red the process(es) that are about to transition, and we abbreviate process terms for brevity.

$$
\begin{array}{lrl}
(1) & \mathsf{proc}(\{x\}, \{y, z\}, x, x.\langle y, z\rangle), \mathsf{proc}(\{b\}, \{x\}, p, \mathsf{case}\ p\ \ldots), \mathsf{proc}(S, \{b\}, b_1, b.\pi_1(b_1)) & \overset{\&\,C}{\Longrightarrow} \\
(2) & \mathsf{proc}(\{x\}, \{y, z\}, x, x.\langle y, z\rangle), \mathsf{proc}(S, \{x\}, b_1, \mathsf{case}\ x\ \ldots) & \overset{\otimes\,C}{\Longrightarrow} \\
(3) & \mathsf{proc}(\{y\}, \Delta_{R_1}, c, R_1), \mathsf{proc}(\{z\}, \Delta_{R_2}, d, R_2), \mathsf{proc}(S, \{y, z\}, b_1, \{\,\} \leftarrow (\nu a)\ \ldots) & \overset{\mathsf{cut}(\{\,\})}{\Longrightarrow} \\
(4) & \mathsf{proc}(\{y\}, \Delta_{R_1}, c, R_1), \mathsf{proc}(\{z\}, \Delta_{R_2}, d, R_2), \mathsf{proc}(\{\,\}, \{z\}, a, a \leftarrow z), \mathsf{proc}(S, \{y\}, b_1, b_1 \leftarrow y) & \overset{\mathsf{id}}{\Longrightarrow} \\
(5) & \mathsf{proc}(\{y\}, \Delta_{R_1}, c, R_1), \mathsf{proc}(\{\,\}, \Delta_{R_2}, d, R_2), \mathsf{proc}(S, \{y\}, b_1, b_1 \leftarrow y) & \overset{\mathsf{drop}}{\Longrightarrow} \\
(6) & \mathsf{proc}(\{y\}, \Delta_{R_1}, c, R_1), \mathsf{proc}(\{\,\}, \{b\}, \hat{b}, \hat{b} \leftarrow b)_{b \in \Delta_{R_2}}, \mathsf{proc}(S, \{y\}, b_1, b_1 \leftarrow y) & \overset{\mathsf{id}}{\Longrightarrow} \\
(7) & \mathsf{proc}(S, \Delta_{R_1}, c, R_1), \mathsf{proc}(\{\,\}, \{b\}, \hat{b}, \hat{b} \leftarrow b)_{b \in \Delta_{R_2}} &
\end{array}
$$

Lines (3), (4), and (5) of this computation trace highlight how cancellation propagates. In line (3), we create a new process whose sole purpose is to mark z as no longer needed by P. Then, in line (4), this process informs the provider of z, R_2, that it may terminate. This triggers, in line (5), the propagation of this cancellation message back along the channels that R_2 uses. While we only show this single step of the propagation, subsequent steps look much the same, alternating between applications of the id *and* drop *rules in order to pass the cancellation backwards through a chain of dependencies. The last step is an application of identity unrelated to cancellation, just forwarding references to channels in S to process R_1.*

6. Examples with Recursion

We now step slightly outside of our system to present some examples involving recursively defined types and processes, which allow for a richer set of programs. We do not formally define recursive types or processes here, as they are well-known from the literature and orthogonal to our concerns (see, for example, [24]).

Since we are already deviating slightly from our system to use recursion, we will, for simplicity of presentation, also write $x \leftarrow P\ ;\ Q$ in place of the (more verbose) $\{x\} \leftarrow (\nu y)P[y/x]\ ;\ Q[y/x]$ in the specific case of cuts with a singleton set of new channels. We will continue to make the new channel explicit in cuts with non-singleton sets.

In these examples, we will work with two modes, L and U, with $\mathsf{L} < \mathsf{U}$, $\sigma(\mathsf{L}) = \{\}$, and $\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$. This is sufficient for the examples we present, but not necessary — some of the examples will use only one of the two modes, and several of the examples using U will not take advantage of both contraction and weakening.

6.1. 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.

Example 9.

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

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

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

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 bi, bj, and bk with $\mathsf{bk} = \neg(\mathsf{bi} \vee \mathsf{bj})$, the following transitions are possible and characterize *nor*:

$$\begin{array}{l}
\mathsf{proc}(\{a\}, \{a'\}, x, x.\mathsf{bi}(a')), \mathsf{proc}(\{b\}, \{b'\}, y, y.\mathsf{bj}(b')), \mathsf{proc}(S, \{a, b\}, z, z \leftarrow \mathit{nor} \leftarrow a, b) \\
\longrightarrow^{*} \mathsf{proc}(c', \{a', b'\}, z', z' \leftarrow \mathit{nor} \leftarrow a', b'), \mathsf{proc}(S, \{c'\}, z, z.\mathsf{bk}(c')) \quad (c' \text{ fresh})
\end{array}$$

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.

$$\mathsf{proc}(\{a\},\{a'\},x,x.\mathsf{bi}(a')),\mathsf{proc}(\{b\},\{b'\},y,y.\mathsf{bj}(b')),\mathsf{proc}(S,\{a,b\},z,z\leftarrow nor\leftarrow a,b) \xRightarrow{\mathsf{call}}$$
$$\mathsf{proc}(\{a\},\{a'\},x,x.\mathsf{bi}(a')),\mathsf{proc}(\{b\},\{b'\},y,y.\mathsf{bj}(b')),\mathsf{proc}(S,\{a,b\},z,\mathsf{case}\ a\ldots) \xRightarrow{\oplus\,\mathsf{bk}}$$
$$\mathsf{proc}(\{b\},\{b'\},y,y.\mathsf{bj}(b')),\mathsf{proc}(S,\{a',b\},z,\mathsf{case}\ b\ldots) \xRightarrow{\oplus\,\mathsf{bk}}$$
$$\mathsf{proc}(S,\{a',b'\},z,z'\leftarrow\ldots) \xRightarrow{\mathsf{cut}(\{z'\})}$$
$$\mathsf{proc}(\{c'\},\{a',b'\},z',z'\leftarrow nor\leftarrow a',b'),\mathsf{proc}(S,\{c'\},z,z.\mathsf{bk}(c'))$$

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 $w\leftarrow nor\leftarrow x\,y$ will multicast a message to the clients of u and u'.

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

$z\leftarrow or\leftarrow x,y=$
$\quad\{u,u'\}\leftarrow(\nu w)(w\leftarrow nor\leftarrow x,y)\ ;$
$\quad z\leftarrow nor\leftarrow u,u'$

An analogous computation to that for *nor* is possible, except that at an intermediate stage of the computation, we will also have a shared channel w carrying the (multicast) message $\mathsf{proc}(\{u,u'\},\{d'\},w,w.\mathsf{bl}(d'))$ with $\mathsf{bl}=\neg(\mathsf{bi}\vee\mathsf{bj})$.

6.2. Example: Map

Mapping a process over a list allows us to demonstrate the use of replicable services, cancellation, and shifts. 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.

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

Note that such a list should not be viewed as a data structure in memory, despite its similarity to a memory-based definition of a list. 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 have type $A\multimap B$. However, in order to use such a process on every element of a list, it must be shared. We therefore obtain the following type and definition for *map*, where all channels not explicitly annotated with a mode subscript are at mode L.

Example 10.

$(\mathsf{f}_\mathsf{U}:\uparrow^\mathsf{U}_\mathsf{L}(A_\mathsf{L}\multimap B_\mathsf{L}))\ (l:list_A)\vdash map::(k:list_B)$
$k\leftarrow map\leftarrow \mathsf{f}_\mathsf{U}\,l=$

$\mathsf{case}\,l\ (\,\mathsf{cons}(l')\Rightarrow$	$\mathsf{case}\,l'(\langle x,l''\rangle\Rightarrow$	$\%$ *receive element* $x:A$ *with continuation* l''
	$\{f'_\mathsf{U},f''_\mathsf{U}\}\leftarrow(\nu a)a\leftarrow \mathsf{f}_\mathsf{U}$	$\%$ *duplicate the channel* f_U
	$f'\leftarrow f'_\mathsf{U}.\mathsf{shift}(f')\ ;$	$\%$ *obtain a fresh linear instance* f' *of* f'_U
	$y\leftarrow f'.\langle x,y\rangle\ ;$	$\%$ *send* x *to* f'*, response will be along fresh* y
	$k'\leftarrow k.\mathsf{cons}(k')\ ;$	$\%$ *select* cons
	$k''\leftarrow k'.\langle y,k''\rangle\ ;$	$\%$ *send* y *with continuation* k''
	$k''\leftarrow map\leftarrow f''_\mathsf{U}\,l'')$	$\%$ *recurse with continuation channels*
$\mid\mathsf{nil}(l')\Rightarrow$	$\{\}\leftarrow(\nu a)a\leftarrow \mathsf{f}_\mathsf{U}$	$\%$ *cancel the channel* f_U
	$k'\leftarrow k.\mathsf{nil}(k')\ ;$	$\%$ *select* nil
	$\mathsf{case}\,l'(\langle\rangle\Rightarrow$	$\%$ *wait for* l' *to close*
	$k'.\langle\rangle\,))$	$\%$ *close* k' *and terminate*

In this example, $f_{\cup}$ is a replicable and cancelable service. In the case of a nonempty list, we create two names for the channel $f_{\cup}$ — 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 $f_{\cup}$. As such, we cancel it before continuing.

7. Session Fidelity, Deadlock-Freedom, and Garbage Collection

While we can prove cut elimination for the form of adjoint logic presented in Section 2, 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.

7.1. Session fidelity

Session fidelity is the message-passing analogue of type preservation under the rules of the operational semantics. Here, it expresses that the interfaces to a configuration remain unchanged as computation proceeds.

Theorem 4 (Session Fidelity)**.** *If* $\Psi \vDash \mathcal{C} :: \Psi'$ *and* $\mathcal{C} \Rightarrow \mathcal{C}'$*, then* $\Psi \vDash \mathcal{C}' :: \Psi'$*.*

Sketch. The proof proceeds by a case analysis on the computation rule used to conclude $\mathcal{C} \Rightarrow \mathcal{C}'$. In each case, we break $\mathcal{C}$ 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 $\Psi \vDash \mathcal{C}' :: \Psi'$.

As a sample case, suppose $\mathcal{C} \overset{\mathsf{drop}}{\Longrightarrow} \mathcal{C}'$. We then know that $\mathcal{C}$ is of the form $\mathcal{C}_1, \mathsf{proc}(\{\}, \Delta, x, P)$, where P is not an identity $x \leftarrow a$, and that $\mathcal{C}'$ has the form $\mathcal{C}_1, \mathsf{proc}(\{\}, \{b\}, y, y \leftarrow b)_{b \in \Delta}$.

Examining the derivation of $\Psi \vDash \mathcal{C} :: \Psi'$, we find the following subtree (where $\overline{\Psi_2} = \Delta$):

$$\frac{0 \in \mu(k) \quad \Psi_2 \vdash P :: (x : A_k)}{\Psi_1 \ \Psi_2 \vDash \mathsf{proc}(\{\}, \overline{\Psi_2}, x, P) :: \Psi_1}\ \mathsf{Proc}$$

In order to show that $\Psi \vDash \mathcal{C}' :: \Psi'$, we can simply take the proof that $\Psi \vDash \mathcal{C} :: \Psi'$ and replace the above segment with a proof that

$$\Psi_1 \ \Psi_2 \vDash \mathsf{proc}(\{\}, \{b\}, y, y \leftarrow b)_{b \in \Delta} :: \Psi_1.$$

Such a proof is easily constructed by composing (with Comp) a collection of proofs of the following form, one for each $b \in \Delta$:

$$\frac{0 \in \mu(m) \quad \dfrac{}{(b : B_m) \vdash y \leftarrow b :: (y : B_m)}\ \mathsf{id}}{\Psi_1 \ (b : B_m) \vDash \mathsf{proc}(\{\}, \{b\}, y, y \leftarrow b) :: \Psi_1}\ \mathsf{Proc}$$

Because $b \in \Delta$, $(b : B_m)$ is in Ψ_2, so we know (by independence) that $m \geq k$ — otherwise, $\Psi_2 \vdash P :: (x : A_k)$ would be ill-formed. Then, as $m \geq k$, $\sigma(m) \supseteq \sigma(k)$, and so $0 \in \mu(m)$ follows from $0 \in \mu(k)$.

While the copy case involves a few more parts (the two copies of P need to be typed as well as all of the newly created identity processes), it follows the same pattern as the drop case. The remaining cases are also similar, but are simplified by dealing with a fixed number of processes, rather than the arbitrary number that the drop and copy rules can create. □

7.2. 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 channel c if:

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

Intuitively, $\mathsf{proc}(S, \Delta, a, P)$ is poised on c if it is blocked trying to communicate along c. Of particular interest is the special case where $\mathsf{proc}(S, \Delta, a, P)$ is poised on the channel a that it provides. Such processes serve as our analogue of values in the following progress theorem:

Theorem 5 (Deadlock-Freedom)**.** *If $(\cdot) \vDash \mathcal{C} :: \Psi$, then exactly one of the following holds:*

1. *There is a $\mathcal{C}'$ such that $\mathcal{C} \Rightarrow \mathcal{C}'$.*
2. *Every $\mathsf{proc}(S, \Delta, a, P)$ in $\mathcal{C}$ is poised* on *a.* [7]

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 a client must occur to the right of the provider 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 the 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).

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

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. *$\Psi \vDash \mathcal{C} :: \Psi'$ if and only if $\Psi \vDash' \mathcal{C} :: \Psi'$.*

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 $(\cdot) \vDash \mathcal{C} :: \Psi$, using Lemma 1 to work right to left. Writing $\mathcal{C} = \mathcal{C}'\ \mathsf{proc}(S, \overline{\Psi'}, a, P)$, we see that either $\mathcal{C}'$ can step, in which case so can $\mathcal{C}$, or every process in $\mathcal{C}'$ 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 $\overline{\Psi'}$.

[7] As such, there can be no cyclic waits — each process is poised on the channel that it provides, and so cannot be waiting as the client of any other process.

7.3. 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 [2, 11]) 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!

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

Some of the prior work on affine session types [10, 7] addresses the problem of garbage collection by using equivalence rules which allow canceled channels to be cleaned up. These approaches, particularly that of Fowler et al. [10], are similar to ours, with a major distinguishing feature being that our system allows for one client of a shared (multi-client) service to cancel its connection to that service. In such affine systems, there is no mechanism for canceling shared channels. Most of the other differences are technical in nature, such as cancellation needing to wait for all prior messages to be received (whereas in our system, cancellation can overtake the processes that serve as messages).

One can generalize the garbage collection property discussed above 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 $\mathcal{C}$ 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.

A configuration $\mathcal{C}$ for which there is Ψ composed entirely of purely positive types such that $\cdot \vDash \mathcal{C} :: \Psi$ 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 process in $\mathcal{C}$. If we do not care about the particular channels in Ψ, we may say simply that $\mathcal{C}$ is *observable*.

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

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

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

Corollary 2. *If $\cdot \vDash \mathcal{C} :: \Psi$ for some Ψ consisting entirely of purely positive types and $\mathcal{C}$ cannot take any steps, then $\mathcal{C} \rhd \Psi$.*

This proof proceeds by a simple induction on the derivation of $\cdot \vDash \mathcal{C} :: \Psi$, 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.

Now, using this theorem, we can give an example of garbage collection for a non-$\mathbf{1}$ but still purely positive type.

Example 11. *Consider the (recursive) type*

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

This is similar to our type of infinite bit-streams from Section 6, but we have added a terminal label e *indicating the end of a bit stream, and we allow the mode m to be arbitrary.*

Now, if we begin with the configuration

$$\cdot \vDash \mathsf{proc}(\{c_0\}, \cdot, c, P) :: (c_0 : \mathsf{bits}_m),$$

and reach a final configuration (one which cannot take any further computation steps), we must end in an observable configuration. Note that while our progress and preservation theorems extend to the setting with recursion, we are not guaranteed termination, and so can say little about what garbage may remain in a non-terminating computation.

Examining the type bits_m *and applying inversion on the definition of observability, we see that such a configuration must be a sequence of messages* $\mathsf{proc}(\{c_n\}, \{c_{n+1}\}, c, c.\mathsf{bi}(c_{n+1}))$*, terminated by a pair of messages of the form* $\mathsf{proc}(\{d\}, \{\}, e, e.\langle\rangle), \mathsf{proc}(\{c_k\}, \{d\}, c, c.\mathsf{e}(d))$ *— in other words, it must be exactly a well-formed bit string, with no left-over garbage.*

8. Related Work

Various items of related work have already been mentioned in the preceding sections either in examples or technical cross-references.

The line of work on using adjoint modal operators to combine logics with different structural properties originated with Benton's LNL [20] that combines (nonlinear) intuitionistic logic and linear logic. This was generalized in unpublished work by Reed [12], who introduces an arbitrary preorder on modes with a uniform logical language and sequent calculus rules. Our formulation is based on a currently unpublished follow-up [14]; a different formulation with categorical semantics without a built-in notion of independence can be found in work by Licata et al. [13, 21].

The formulation of adjoint logic in its *semi-axiomatic form* supporting asynchronous communication originated in the earlier workshop version of this paper [17]. It builds upon earlier work introducing a theory of session-typed asynchronous communication either using explicit message buffers [3] or encoding it in an asynchronous calculus [5, 11]. The intuitionistic (nonlinear) semi-axiomatic sequent calculus SAX has since been studied in more depth from the proof-theoretic perspective, establishing a form of cut elimination and a write-once shared memory semantics [6].

At the core of our work is the theory of binary session types [1, 27]. More recent introductions are provided by Vasconcelos [28] and Honda et al. [29]. A Curry-Howard correspondence between session types and intuitionistic linear logic was established by Caires et al. [2, 23], with classical variants by Wadler [22] and Caires et al. [23]. Processes here are expressed in the the π-calculus, with its natural *synchronous* communication behavior. The interpretation of $!A$ in these languages allows for replicable services, modeling and extending *access points* in prior work on session types.

Affine session types to allow for cancellation go back to Mostrous et al. [7] and have since been used in a number of ways [8, 9, 10]. Several of these systems use cancellation as a way of working with exceptions, which we believe could also be done in our system, but is left for future work. The Curry-Howard correspondence was extended to encompass affine types by the second author and Griffith [11], which also introduced three fixed modes, connected by shifts as in the present paper. However, only the linear mode had a full complement of session types, while the others were populated only by shifts. This has turned out to be tedious for programming in SILL [24, 30, 31] (a language that combines ordinary functional and session-typed concurrent programming) and furthermore did not support multicast.

Caires and Pérez [32] extend earlier work to integrate control effects and, in particular, nondeterminism and failure. As such, their system models a form of affinity and cancellation, controlled by two dual modalities, but is otherwise orthogonal to our concerns. For example, our language does not explicitly

model any nondeterminism and we conjecture that our language remains deterministic.[8] It is an interesting item for future work to investigate if the adjoint approach is compatible with an intuitionistic version of their system.

Using linearity to avoid garbage collection goes back to Girard and Lafont [33] and has since been considered mostly in the case of functional languages (for example, Wadler [34] and Igarashi and Kobayashi [35]). It has recently found its way into Rust, a widely used programming language with *affine types* [36, 37]. Explicit distributed garbage collection was also discussed by Griffith [31] in the context of SILL.

9. 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 avoiding the need for 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 [38, 6]. 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 [39].

Acknowledgments

We wish to thank the anonymous reviewers of a previous version of this paper for their feedback, and William Chargin for his work on earlier versions of adjoint logic and its semantics. This materials is based upon work supported by the National Science Foundation under Grant No. CCF-1718267.

References

[1] K. Honda, Types for dyadic interaction, in: 4th International Conference on Concurrency Theory, CONCUR'93, Springer LNCS 715, 1993, pp. 509–523. doi:10.1007/3-540-57208-2_35.

[2] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010), Springer LNCS 6269, Paris, France, 2010, pp. 222–236. doi:10.1007/978-3-642-15375-4_16.

[3] S. J. Gay, V. T. Vasconcelos, Linear type theory for asynchronous session types, Journal of Functional Programming 20 (1) (2010) 19–50. doi:10.1006/inco.1994.1093.

[4] N. Kobayashi, B. C. Pierce, D. N. Turner, Linearity and the pi-calculus, in: H.-J. Boehm, G. Steele (Eds.), Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL'96), ACM, St. Petersburg Beach, Florida, USA, 1996, pp. 358–371.

[5] H. DeYoung, L. Caires, F. Pfenning, B. Toninho, Cut reduction in linear logic as asynchronous session-typed communication, in: P. Cégielski, A. Durand (Eds.), Proceedings of the 21st Conference on Computer Science Logic, CSL 2012, 2012, pp. 228–242. doi:10.4230/LIPIcs.CSL.2012.228.

[6] H. DeYoung, F. Pfenning, K. Pruiksma, Semi-axiomatic sequent calculus, in: 5th International Conference on Formal Structures for Computation and Deduction, Paris, France, 2020, to appear.

[7] D. Mostrous, V. Vasconcelos, Affine sessions, in: E. Kühn, R. Pugliese (Eds.), 16th International Conference on Coordination Models and Languages, Springer LNCS 8459, Berlin, Germany, 2014, pp. 115–130. doi:10.1007/978-3-662-43376-8_8.

[8] A. Scalas, N. Yoshida, Lightweight session programming in scala, in: Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), LICIcs 56, Rome, Italy, 2016, pp. 21:1–21:28. doi:10.4230/LIPIcs.ECOOP.2016.21.

[9] L. Padovani, A simple library implementation of binary sessions, Journal of Functional Programming 27 (e4) (2017). doi:10.1016/0304-3975(83)90059-2.

[10] S. Fowler, S. Lindley, J. G. Morris, S. Decova, Exceptional asynchronous session types, in: Proceedings of the 46th Symposium on Programming Languages (POPL 2019), ACM, Cascais, Portugal, 2019, pp. 28:1–28:29.

[11] F. Pfenning, D. Griffith, Polarized substructural session types, in: A. Pitts (Ed.), Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015), Springer LNCS 9034, London, England, 2015, pp. 3–22, invited talk. doi:10.1007/978-3-662-46678-0_1.

[8] This property is not immediate due to the possible interactions between cancellation and message receipt.

[12] J. Reed, A judgmental deconstruction of modal logic, unpublished manuscript (May 2009).
URL http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf
[13] D. R. Licata, M. Shulman, Adjoint logic with a 2-category of modes, in: International Symposium on Logical Foundations of Computer Science (LFCS), Springer LNCS 9537, 2016, pp. 219–235. doi:10.1007/978-3-319-27683-0_16.
[14] K. Pruiksma, W. Chargin, F. Pfenning, J. Reed, Adjoint logic, unpublished manuscript (Apr. 2018).
URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf
[15] C. Palamidessi, Comparing the expressive power of the synchronous and the asynchronous π-calculus, Mathematical Structures in Computer Science 13 (5) (2003) 685–719.
[16] T. Tu, X. Liu, L. Song, Y. Zhang, Understanding real-world concurrency bugs in Go, in: Architectural Support for Programming Languages and Operating Systems (ASPLOS'19), ACM, Providence, RI, USA, 2019, pp. 865–878.
[17] K. Pruiksma, F. Pfenning, A message-passing interpretation of adjoint logic, in: F. Martins, D. Orchard (Eds.), Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, Prague, Czech Republic, 7th April 2019, Vol. 291 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Association, 2019, pp. 60–79. doi:10.4204/EPTCS.291.6.
[18] G. Gentzen, Untersuchungen über das logische Schließen, Mathematische Zeitschrift 39 (1935) 176–210, 405–431, english translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969. doi:10.1007/BF01201353.
[19] S. Negri, J. von Plato, Structural Proof Theory, Cambridge University Press, 2001. doi:10.1017/CBO9780511527340.
[20] N. Benton, A mixed linear and non-linear logic: Proofs, terms and models, in: L. Pacholski, J. Tiuryn (Eds.), Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94), Springer LNCS 933, Kazimierz, Poland, 1994, pp. 121–135, an extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge. doi:10.1007/BFb0022251.
[21] D. R. Licata, M. Shulman, M. Riley, A fibrational framework for substructural and modal logics, in: International Conference on Formal Structures for Computation and Deduction, LIPIcs, Oxford, 2017. doi:10.4230/LIPIcs.FSCD.2017.25.
[22] P. Wadler, Propositions as sessions, in: Proceedings of the 17th International Conference on Functional Programming, ICFP 2012, ACM Press, Copenhagen, Denmark, 2012, pp. 273–286. doi:10.1145/2364527.2364568.
[23] L. Caires, F. Pfenning, B. Toninho, Linear logic propositions as session types, Mathematical Structures in Computer Science 26 (3) (2016) 367–423. doi:10.1016/j.tcs.2010.01.028.
[24] B. Toninho, L. Caires, F. Pfenning, Higher-order processes, functions, and sessions: A monadic integration, in: M.Felleisen, P.Gardner (Eds.), Proceedings of the European Symposium on Programming (ESOP'13), Springer LNCS 7792, Rome, Italy, 2013, pp. 350–369. doi:10.1007/978-3-642-37036-6_20.
[25] I. Cervesato, A. Scedrov, Relating state-based and process-based concurrency through linear logic, Information and Computation 207 (10) (2009) 1044–1077. doi:10.1016/j.ic.2008.11.006.
[26] J. Palsberg, C. B. Jay, The essence of the visitor pattern, in: Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98) (Cat. No. 98CB 36241), IEEE, 1998, pp. 9–15.
[27] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: C. Hankin (Ed.), 7th European Symposium on Programming Languages and Systems (ESOP 1998), Springer LNCS 1381, 1998, pp. 122–138.
[28] V. T. Vasconcelos, Fundamentals of session types, Information and Computation 217 (2012) 52–70.
[29] K. Honda, R. Hu, R. Neykova, T.-C. Chen, R. Demangeon, P.-M. Deniélou, N. Yoshida, Structuring communcation with session types, in: Concurrent Objects and Beyond (COB 2014), Springer LNCS 8665, 2014, pp. 105–127.
[30] B. Toninho, A logical foundation for session-based concurrent computation, Ph.D. thesis, Carnegie Mellon University and Universidade Nova de Lisboa, available as Technical Report CMU-CS-15-109 (May 2015).
[31] D. Griffith, Polarized substructural session types, Ph.D. thesis, University of Illinois at Urbana-Champaign (Apr. 2016).
[32] L. Caires, J. A. Pérez, Linearity, control effects, and behavioral types, in: European Symposium on Programming, Springer, 2017, pp. 229–259. doi:10.1007/978-3-662-54434-1_9.
[33] J.-Y. Girard, Y. Lafont, Linear logic and lazy computation, in: H. Ehrig, R. Kowalski, G. Levi, U. Montanari (Eds.), Proceedings of the International Joint Conference on Theory and Practice of Software Development, Vol. 2, Springer-Verlag LNCS 250, Pisa, Italy, 1987, pp. 52–66. doi:10.1007/BFb0014972.
[34] P. Wadler, Linear types can change the world, in: IFIP TC, Vol. 2, 1990, pp. 347–359.
[35] A. Igarashi, N. Kobayashi, Garbage collection based on a linear type system, in: Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC'00), Vol. 152, 2000.
[36] J. A. Tov, R. Pucella, Practical affine types, in: T. Ball, M. Sagiv (Eds.), Proceedings of the 38th Symposium on Principles of Programming Languages (POPL 2011), ACM Press, 2011, pp. 447–458.
[37] Available at www.rust-lang.org.
[38] F. Pfenning, K. Pruiksma, A shared memory semantics for session types, Invited talk at the Workshop on Linearity/TLLA, Oxford, UK (Jul. 2018).
[39] S. Balzer, F. Pfenning, Manifest sharing with session types, in: International Conference on Functional Programming (ICFP), ACM, 2017, pp. 37:1–37:29. doi:10.1145/3110281.