Adjoint Logic with Applications

Klaas Pruiksma

April 29, 2021

Contents

1 Introduction

A wide variety of different logics, with their own sets of connectives and inference rules, have been used in the computer science literature — most prominently, intuitionistic logic, but also intuitionistic linear logic (ILL), various forms of modal logics, and more. It is then natural to consider whether it is possible to express many or all of these in a single framework, leading to the idea of *combining logics*. Several existing systems do this to varying degrees, including ILL with the exponential modality ! [16], Benton's mixed linear and non-linear logic LNL [5], various logics with subexponentials [22, 30, 31], and some earlier forms of adjoint logic [40, 26, 27].

In light of the Curry-Howard correspondence between logics and programming languages (which we will refer to simply as languages for brevity), a framework that can combine several logics or express several logics should also be able to combine multiple languages or language features, or be able to express multiple languages. The work described above, however, focuses heavily on the logical side — while several of the systems, including LNL and some of the subexponential frameworks, give semantics to the logic, they do not comprise full languages, and nor do they seek to describe the mixing of different language features.

Our goal is to develop a framework that successfully embeds a wide variety of language features in a natural way, ideally with minimal overhead needed. Thus far, we have investigated several formulations of adjoint logic and the concurrent semantics that arise naturally by taking proof reduction as the basis of computation. Building on this, we have developed a language based on adjoint logic which allows for a mix of sequential and concurrent execution, and used this language to reconstruct a variety of common concurrency paradigms, such as futures, monadic concurrency, and a form of fork/join.

A key distinction between adjoint logic and some of the previous approaches is its symmetry or uniformity. ILL with ! allows for the encoding of standard (structural or nonlinear) intuitionistic logic, but this is somewhat indirect, because the linear layer is distinguished, having a full set of connectives available, while the nonlinear layer only has propositions $!A$, and so nonlinear connectives needs to be constructed out of ! and the linear connectives. Likewise, subexponential logics have a distinguished base layer, and so have similarly indirect encodings. In our system of adjoint logic, no layer is distinguished, and a primary goal of both the logic and the languages based on it is to treat all layers as uniformly as possible. Some breaking of this symmetry is necessary to allow linear and unrestricted propositions to mix freely without causing problems, but this is limited (and can still be described in a uniform manner).

At a high level, adjoint logic separates propositions into multiple layers, or modes, and provides some constraints on how propositions can be used depending on what mode they live at. Similarly, the adjoint framework for languages separates programs and types into different modes, and constrains what programs may be written at each mode. We propose to investigate modularity and isolation in the context of this adjoint framework for languages. Isolation addresses questions of how well language features are contained at a mode — if we want to allow a specific feature only at some modes, is that possible? Are there constraints on what features can occur at what modes? Looking at this through the lens of modularity, we get slightly different questions — can we replace the portion of the langauge at a given mode with a different language in a modular fashion? To what extent can one mode tell how computation works at other modes? An ideal result would address these questions by giving a description of how languages or semantics can be combined, and (depending on perspective) either what constraints the structure of modes places on these combinations or what constraints the combinations put on the structure of the modes.

In the sections that follow, we will present the existing work, discuss the proposed work, and then briefly touch on a variety of directions for future work. Section 2 presents adjoint logic itself, in three separate forms, and presents some results on the proof theory. Sections 2.1 and 2.2 provide calculi for adjoint logic, one with the structural rules of weakening and contraction made explicit, and one where they are implicit. These are relevant to the programming-related work later on, while Section 2.3, which presents a focused calculus and Section 2.4, which shows how adjoint logic can be used to model various other logics are more proof-theory focused and are not necessary to understand the remaining work. Likewise, Section 2.5, which includes some brief remarks on the proof theory that do not fit elsewhere is not needed later on. Section 3 gives an overview of what changes we will make (and what choices need to be made) in moving from the logics of Section 2 to the languages of Sections 4 and 5. We then present two languages based on different calculi for adjoint logic: A message-passing language similar to the π-calculus or various session-typed process calculi (Section 4), and a shared-memory process calculus with some relations to destination-passing semantics for functional languages (Section 5). In Section 6, we describe the core of the proposed work of the thesis, along with some of the potential stepping stones towards the final goal. Finally, we discuss various related topics and potential extensions for future work in Section 7.

2 Adjoint Logic

Adjoint logic is a schema for describing particular logics with a wide variety of features. In our approach, there are two key ideas that underlie adjoint logic. First, rather than treating propositions uniformly, we stratify them, giving each proposition a *mode of truth* (or just *mode*) m. We think of different modes as potentially allowing different ways of proving or working with propositions. Second, to a more typical set of connectives we add *shifts* $\uparrow_k^m$ and $\downarrow_m^\ell$, pronounced *up from k to m*and *down from ℓ to m*, or just *up* and *down* when the modes are either clear or not particularly important. These are unary connectives that allow us to embed propositions from one mode into another under certain constraints.

With different modes, we also need to think about how they relate to one another. Different prior examples of adjoint logic have used different frameworks here, of varying degrees of complexity. For our purposes, we will treat modes as being drawn from a preordered set, or a set equipped with a transitive and reflexive relation $\leq$. Additionally, we would like to restrict what structural rules may be used at each mode. We do this by using a monotone map σ that takes modes to subsets of the two-element set $\{W, C\}$, where W represents weakening and C contraction. If $W \in \sigma(m)$, we say that m admits weakening, and likewise, if $C \in \sigma(m)$, then m admits contraction. Note that we always allow exchange, as this simplifies the presentation of the logic, but we see no inherent obstacle to a system that restricts exchange as well, as in the system of Licata et al. [27], or that of Kanovich et al. [22].

As has been observed in the past, for example in Benton's LNL [5], allowing propositions at different modes to interact freely leads to problems, such as being able to duplicate or discard supposedly linear propositions (whose mode m satisfies $\sigma(m) = \{\}$). In order to avoid this potential problem, we will enforce globally the following principle, which we call the *declaration of independence*:

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

The declaration of independence means that our sequents will have the form

$$\Psi \vdash A_k \qquad\qquad \text{where } \Psi \geq k.$$

By $\Psi \geq k$, we mean that each antecedent B_m in Ψ satisfies $m \geq k$. This type of sequent generalizes a common dyadic [1] or two-zone presentation of logics with modal operators [4], where proofs of one type of judgment may be restricted to only depend on a certain type of hypotheses. For instance, proofs of validity may be forbidden to depend on hypotheses dealing with truth [35], or proofs of unrestricted (structural) propositions may be forbidden to depend on linear hypotheses. [5]

The propositions at each mode are constructed uniformly, using the syntax of linear logic for connectives, other than the newly added shifts $\uparrow_k^m A_k$ and $\downarrow_m^\ell A_\ell$. Note that $\uparrow_k^m A_k$ requires $k \leq m$ in order to be well-formed, and likewise, $\downarrow_m^\ell A_\ell$ requires $m \leq \ell$. Given this, we can present the syntax of propositions:

$$A_m, B_m \quad ::= \quad p_m \mid A_m \multimap_m B_m \mid A_m \otimes_m B_m \mid \mathbf{1}_m \mid \oplus_{j\in J} A_m^j \mid \&_{j\in J} A_m^j \mid \uparrow_k^m A_k \mid \downarrow_m^\ell A_\ell$$

Here, p_m stands for atomic m-propositions. Anticipating the needs of an operational interpretation (see [37, 38]), we have generalized internal and external choice ($\oplus$ and $\&$) to n-ary constructors parameterized by a finite index set J. With $J = \emptyset$ we recover $\top = \&_{j\in\emptyset}(\,)$ and $\mathbf{0} = \oplus_{j\in\emptyset}(\,)$ in any mode, and it is similarly clear that with a two-element index set, we recover the standard binary $\oplus$ and $\&$. Note also that each mode has the same connectives, whose left and right rules are the same for each mode, with the differences in mode arising from the permissible structural rules, as well as the rules for the shifts. Also note that while we have mode labels on the connectives here, indicating that, for instance, $\multimap_m$ and $\multimap_k$ are different (but related!) connectives that only operate on m-propositions and k-propositions, respectively, we will almost always omit them, as they can be inferred from the modes of the propositions on which they operate.

In order to be able to describe side conditions for rules, we define for contexts Ψ the set $\sigma(\Psi)$.

Definition 1. *We define $\sigma(\Psi)$ inductively as follows:*

$$\begin{array}{rcl} \sigma(\cdot) & = & \{\mathsf{W}, \mathsf{C}\} \\ \sigma(A_m) & = & \sigma(m) \\ \sigma(\Psi_1, \Psi_2) & = & \sigma(\Psi_1) \cap \sigma(\Psi_2) \end{array}$$

Intuitively, $\sigma(\Psi)$ is the largest set of structural properties shared by all propositions in Ψ, so if $\mathsf{W} \in \sigma(\Psi)$, then every proposition in Ψ is weakenable, and if $\mathsf{C} \in \sigma(\Psi)$, then every proposition in Ψ is contractible.

At this point, we are equipped to present our calculi for adjoint logic.

2.1 ADJ^E

The first calculus we examine makes the structural rules of weakening and contraction explicit. This calculus closely matches the calculi used for ILL with the exponential !, where the rules for weakening and contraction of replicated formulae are explicit, and makes it easy to see what the rules look like at any given mode.

This calculus, which we call ADJ^E, can be found in Figure 1. As is common for the sequent calculus, we read the rules in the direction of bottom-up proof construction, and so for each rule, we assume that the conclusion is well-formed (satisfies independence), and add side conditions to enforce that the premises are well-formed as well.

We begin with the judgmental rules of identity and cut, which express the connection between antecedents and succedents. Identity says that given A_m as a hypothesis, we may conclude A_m. Cut says the opposite: if we can conclude A_m, then we are entitled to assume A_m as a hypothesis.

In the cut rule, independence comes into play: if we only assume that the conclusion satisfies independence, we get that $\Psi_1\ \Psi_2 \geq k$, but in order for the premise $\Psi_1 \vdash A_m$ to be well-formed, we need $\Psi_1 \geq m$, and likewise, for $\Psi_2, A_m \vdash C_k$ to be well-formed, we need $m \geq k$. We combine these into the single premise $\Psi_1 \geq m \geq k$.

The structural rules of weakening and contraction are straightforward — they simply need to check that the mode of the principal formula allows the rule to be used.

The logical rules defining the standard multiplicative and additive connectives are the linear rules for those connectives, regardless of what mode they are at, since we have separated out the structural rules. In all but one case — that of $\multimap L$ — 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 $\Psi_1 \geq k$, $\Psi_2 \geq k$, and $m \geq k$. These facts by themselves already imply the well-formedness of the second premise, but we need to check that $\Psi_1 \geq m$ in order for the first premise ($\Psi_1 \vdash A_m$) to be well-formed.

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

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

$$\frac{i \in J \quad \Psi \vdash A_m^i}{\Psi \vdash \oplus_{j \in J} A_m^j}\ \oplus R^i \qquad \frac{\Psi, A_m^j \vdash C_k \quad \text{for all } j \in J}{\Psi, \oplus_{j \in J} A_m^j \vdash C_k}\ \oplus L$$

$$\frac{\Psi \vdash A_m^j \quad \text{for all } j \in J}{\Psi \vdash \&_{j \in J} A_m^j}\ \& R \qquad \frac{i \in J \quad \Psi, A_m^i \vdash C_k}{\Psi, \&_{j \in J} A_m^j \vdash C_k}\ \& L^i$$

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

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

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

$$\frac{\Psi \geq \ell \quad \Psi \vdash A_\ell}{\Psi \vdash {\downarrow}_m^\ell A_\ell}\ {\downarrow}R \qquad \frac{\Psi, A_\ell \vdash C_k}{\Psi, {\downarrow}_m^\ell A_\ell \vdash C_k}\ {\downarrow}L$$

$$\frac{\Psi \vdash A_k}{\Psi \vdash {\uparrow}_k^m A_k}\ {\uparrow}R \qquad \frac{k \geq \ell \quad \Psi, A_k \vdash C_\ell}{\Psi, {\uparrow}_k^m A_k \vdash C_\ell}\ {\uparrow}L$$

Figure 1: Adjoint Logic with Explicit Structural Rules (ADJ^E).
We presuppose that the conclusion of each rule satisfies the declaration of independence and ensure, with conditions on modes, that the premises will, too.

Finally, we reach the rules for the new shift connectives. Recall that in ${\uparrow}_k^m A_k$ and ${\downarrow}_m^\ell A_\ell$ we require that $k \leq m$ and $m \leq \ell$, which provides additional information for well-formedness. We first consider the two rules for $\uparrow$. We know from the conclusion of ${\uparrow}R$ 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 that this rule is *invertible*, an observation integrated into the focusing rules for the system ADJ^F presented in Section 2.3.

From the conclusion of ${\uparrow}L$, 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 $\Psi, A_k \vdash C_\ell$ to be well-formed. As such, we need to add $k \geq \ell$ as a premise to the rule, and the rule is non-invertible.

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

At this point we can prove some simple properties of the shifts as illustrative examples. For instance, shifts distribute over implication — given modes $k \leq m$, we can construct the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{A_m \multimap_m B_m, A_m \geq m \qquad \dfrac{A_m \geq m \quad \dfrac{}{A_m \vdash A_m}\,\mathsf{id} \quad \dfrac{}{B_m \vdash B_m}\,\mathsf{id}}{A_m \multimap_m B_m, A_m \vdash B_m}\,{\multimap} L}{A_m \multimap_m B_m, A_m \vdash {\downarrow}^m_k B_m}\,{\downarrow}R}{A_m \multimap_m B_m, {\downarrow}^m_k A_m \vdash {\downarrow}^m_k B_m}\,{\downarrow}L}{{\downarrow}^m_k(A_m \multimap_m B_m), {\downarrow}^m_k A_m \vdash {\downarrow}^m_k B_m}\,{\downarrow}L}{{\downarrow}^m_k(A_m \multimap_m B_m) \vdash {\downarrow}^m_k A_m \multimap_k {\downarrow}^m_k B_m}\,{\multimap} R$$

A similar proof shows that the upshift also distributes over implication. We can likewise show that the shifts distribute over the other connectives — for instance, the following proof shows that ${\uparrow}^m_k$ distributes over $\&_{j \in J}$. Note that the premises of $\& R$ (and likewise $\oplus L$) need not all have proofs of the same form, but in this example they do (and so we show a generic case):

$$\dfrac{\dfrac{\dfrac{k \geq k \qquad \dfrac{\ell \in J \quad \dfrac{}{A^\ell_k \vdash A^\ell_k}\,\mathsf{id}}{\&_{j\in J} A^j_k \vdash A^\ell_k}\,\& L^\ell}{{\uparrow}^m_k \&_{j \in J} A^j_k \vdash A^\ell_k}\,{\uparrow}L}{{\uparrow}^m_k \&_{j\in J} A^j_k \vdash {\uparrow}^m_k A^\ell_k \quad \text{for all } \ell \in J}\,{\uparrow}R}{{\uparrow}^m_k \&_{j \in J} A^j_k \vdash \&_{j \in J} {\uparrow}^m_k A^j_k}\,\& R$$

2.1.1 Cut Elimination

Because we have an explicit rule of contraction, cut elimination does not follow by a simple structural induction. However, we can follow Gentzen's approach [15] and allow multiple copies of the same proposition to be removed by the cut, which then allows a structural induction argument. To do this, we generalize the rule of cut to a *multicut*,[1] which can remove $n \geq 0$ copies of a proposition *provided that the structural properties of the mode of that proposition allow it.*

[1] The term "multicut" has been used in the literature for several different rules We follow here the proof theory literature [29, 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 [15], where he calls it "Mischung", rather than one of the variants that cuts out several propositions together, for instance.

To simplify the presentation of this rule, we define the *multiplicities of a mode* m ($\mu(m) \subseteq \mathbb{N}$), specifying what numbers of copies of a proposition at mode m can be cut out by a multicut. This is defined as:

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

With this notation, we can write down a simple rule of multicut (where A_m^n denotes n copies of A_m):

$$\frac{\Psi_1 \geq m \geq k \quad n \in \mu(m) \quad \Psi_1 \vdash A_m \quad \Psi_2, A_m^n \vdash C_k}{\Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}(n)$$

As $1 \in \mu(m)$ always, the cut rule that we presented earlier is simply the $n = 1$ case of the $\mathsf{cut}(n)$ rule. As such, proving that the multicut rule is admissible from ADJ^E (without using cut) also proves that cut is admissible, and likewise, if we can eliminate all multicuts from ADJ^E proofs, we can also eliminate standard cuts.

Beyond this, we also observe that weakening and contraction can be derived as instances of multicut where the left-hand premise is an identity, with $n = 0$ and 2, respectively, and that each instance of multicut can be derived from cut along with weakening or contraction. Exchanging cut for multicut therefore does not affect provability (in the presence of weakening and contraction). Moreover, if we replace cut, weakening, and contraction all with multicut, we have a system with the same strength as the original (although we lose cut elimination), something that we will make use of in Section 4.

We now move on to prove admissibility of multicut, writing $\Psi \Vdash_E A_m$ to mean that there is an ADJ^E proof of $\Psi \vdash A_m$ that uses neither cut nor multicut:

Theorem 1 (Admissibility of multicut in ADJ^E)**.** *If* $\Psi_1 \geq m \geq k$, $n \in \mu(m)$, $\Psi_1 \Vdash_E A_m$, *and* $\Psi_2, A_m^n \Vdash_E C_k$, *then* $\Psi_1, \Psi_2 \Vdash_E C_k$.

Proof Sketch. This follows by induction on the (lexicographically ordered) triple $(A_m, \mathcal{D}, \mathcal{E})$, where $\mathcal{D}$ is the proof that $\Psi_1 \Vdash_E A_m$ and $\mathcal{E}$ is the proof that $\Psi_2, A_m^n \Vdash_E C_k$. □

Admissibility of multicut then yields cut elimination in its usual form.

Theorem 2 (Cut elimination for ADJ^E)**.** *If* $\Psi \vdash_E A_m$, *then* $\Psi \Vdash_E A_m$.

Sketch. This follows in the standard way from admissibility of multicut by induction over the proof that $\Psi \vdash_E A_m$, using admissibility of multicut to eliminate each cut as it is encountered. □

2.1.2 Identity expansion

Identity expansion for ADJ^E is standard in its statement and proof.

Theorem 3 (Identity Expansion). *If $\Psi \vdash_E A_m$, then there exists a proof that $\Psi \vdash_E A_m$ using identity rules only at atomic propositions p_m, 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 $A_m \vdash_E A_m$ 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 $\Psi \vdash_E A_m$, applying the above result to remove any identities on non-atomic propositions. □

2.2 ADJ^I

In order to move towards an eventual focused system (Section 2.3), we loosely follow the approach of Andreoli [1], eliminating some of the nondeterminism in proof search by removing the structural rules, instead building weakening and contraction into the other rules. A side benefit of this intermediate system ADJ^I, whose rules can be found in Figure 2, is that it is well-suited to embedding logics whose standard presentations similarly leave structural rules implicit, as many structural modal logics do.

There are a few key differences between ADJ^E and Andreoli's Σ_1. First, we allow for modes to have weakening without contraction and vice versa, and second, we have the shift connectives to deal with. Most of the shift rules are straightforward to turn into this implicit form, but ${\downarrow}R$ requires some thought because of its restriction on the modes allowed in the context. In ADJ^E, we can weaken away any A_m with $\mathsf{W} \in \sigma(m)$ before applying ${\downarrow}R$ in order to satisfy that restriction. In order to match that behavior, in the ${\downarrow}R$ rule of ADJ^I, we split the context into two pieces, Ψ_1 and Ψ_2, and require that $\Psi_1 \geq m$, while $\mathsf{W} \in \sigma(\Psi_2)$. This rule then corresponds to the ADJ^E proof which weakens everything in Ψ_2 and then applies ${\downarrow}R$. Weakening is otherwise easily handled at the leaves of the proof in a similar manner.

Contraction without weakening leads to most of the complication in this system. Were contraction to always imply weakening, we could simply propagate contractible propositions to all branches of the proof and weaken them if they are not needed, as is done in standard intuitionistic logic, or as with Andreoli's second context Θ. Instead, for each multiplicative rule with two premises ($\otimes R$ and $\multimap L$), as well as for the cut rule, we split the context into three parts, sending Ψ_1 to the first premise only, Ψ_3 to the second premise only, and Ψ_2 to both (and, of course, we require that Ψ_2 be contractible). The nondeterminism in how Ψ_2 is chosen allows us to propagate contractible propositions to precisely those premises where they will be needed. Similarly, we allow (but do not require) the principal formula to be preserved in the premises after applying a left rule. To do this, we simply have two versions of each left rule, labelled with $\alpha \in \{0, 1\}$ (or $\alpha, \beta \in \{0, 1\}$ in one case). Each has $(A_m)^\alpha$ in its premise, where $(A_m)^1$ is simply A_m, while $(A_m)^0$ is the empty context. The rule with $\alpha = 1$ thus preserves the principal formula, while the rule with $\alpha = 0$ consumes it. Of course, we must only allow $\alpha = 1$ if $\mathsf{C} \in \sigma(m)$. These changes, along with the removal of the weakening and contraction rules, give us ADJ^I, as shown in Figure 2.

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi, A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Psi_1, \Psi_2 \geq m \geq k \quad \mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash A_m \quad \Psi_2, \Psi_3, A_m \vdash C_k}{\Psi_1, \Psi_2, \Psi_3 \vdash C_k}\ \mathsf{cut}$$

$$\frac{i \in J \quad \Psi \vdash A^i_m}{\Psi \vdash \oplus_{j\in J} A^j_m}\ \oplus R^i \qquad \frac{\Psi, (\oplus_{j\in J} A^j_m)^\alpha, A^j_m \vdash C_k \quad \text{for all } j \in J}{\Psi, \oplus_{j\in J} A^j_m \vdash C_k}\ \oplus L^\alpha$$

$$\frac{\Psi \vdash A^j_m \quad \text{for all } j \in J}{\Psi \vdash \&_{j\in J} A^j_m}\ \& R \qquad \frac{i \in J \quad \Psi, (\&_{j\in J} A^j_m)^\alpha, A^i_m \vdash C_k}{\Psi, \&_{j\in J} A^j_m \vdash C_k}\ \& L^{i,\alpha}$$

$$\frac{\mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash A_m \quad \Psi_2, \Psi_3 \vdash B_m}{\Psi_1, \Psi_2, \Psi_3 \vdash A_m \otimes B_m}\ \otimes R \qquad \frac{\Psi, (A_m \otimes B_m)^\alpha, A_m, B_m \vdash C_k}{\Psi, A_m \otimes B_m \vdash C_k}\ \otimes L^\alpha$$

$$\frac{\Psi, A_m \vdash B_m}{\Psi \vdash A_m \multimap B_m}\ \multimap R$$

$$\frac{\Psi_1, \Psi_2 \geq m \quad \mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2, (A_m \multimap B_m)^\alpha \vdash A_m \quad \Psi_2, \Psi_3, (A_m \multimap B_m)^\beta, B_m \vdash C_k}{\Psi_1, \Psi_2, \Psi_3, A_m \multimap B_m \vdash C_k}\ \multimap L^{\alpha,\beta}$$

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi \vdash \mathbf{1}_m}\ \mathbf{1}R \qquad \frac{\Psi, (\mathbf{1}_m)^\alpha \vdash C_k}{\Psi, \mathbf{1}_m \vdash C_k}\ \mathbf{1}L^\alpha$$

$$\frac{\Psi_1 \geq m \quad \mathsf{W} \in \sigma(\Psi_2) \quad \Psi_1 \vdash A_m}{\Psi_1, \Psi_2 \vdash {\downarrow}^m_k A_m}\ {\downarrow}R \qquad \frac{\Psi, ({\downarrow}^m_k A_m)^\alpha, A_m \vdash C_\ell}{\Psi, {\downarrow}^m_k A_m \vdash C_\ell}\ {\downarrow}L^\alpha$$

$$\frac{\Psi \vdash A_k}{\Psi \vdash {\uparrow}^m_k A_k}\ {\uparrow}R \qquad \frac{k \geq \ell \quad \Psi, ({\uparrow}^m_k A_k)^\alpha, A_k \vdash C_\ell}{\Psi, {\uparrow}^m_k A_k \vdash C_\ell}\ {\uparrow}L^\alpha$$

Figure 2: Adjoint Logic with Implicit Structural Properties (ADJ^I).
We presuppose that the conclusion of each rule satisfies the declaration of independence and ensure, with conditions on modes, that the premises will, too.
α and β range over $\{0, 1\}$. $(A_m)^0$ always denotes the empty context, while $(A_m)^1$ denotes A_m. If α or β are 1, then $\mathsf{C} \in \sigma(m)$ should be treated as an additional premise of the rule.

2.2.1 Equivalence of ADJ^I and ADJ^E

In order to justify that ADJ^I and ADJ^E are simply different presentations of the same logic, we want to show that ADJ^I is sound and complete with respect to ADJ^E. That it is sound is nearly immediate, as each rule of ADJ^I is derivable in ADJ^E. Completeness follows from Lemma 1, as all rules of ADJ^E other than the structural rules of weakening and contraction are derivable in ADJ^I. One interesting (though unsurprising) feature of the translations from the proofs of soundness and completeness is that both take cut-free proofs to cut-free proofs, as well as translating identities at A_m to identities at A_m. As such, we get cut elimination and identity expansion for ADJ^I for free via these translations and our results for ADJ^E.

Theorem 4 (Soundness of ADJ^I). *If $\Psi \vdash_I A_m$, then $\Psi \vdash_E A_m$. Moreover, if $\Psi \Vdash_I A_m$, then $\Psi \Vdash_E A_m$.*

Lemma 1 (Admissibility of weakening and contraction for ADJ^I)**.**

1. *If $\Psi \vdash_I C_k$ and $\mathsf{W} \in \sigma(m)$, then $\Psi, A_m \vdash_I C_k$.*
2. *If $\Psi, A_m, A_m \vdash_I C_k$ and $\mathsf{C} \in \sigma(m)$, then $\Psi, A_m \vdash_I C_k$.*

Theorem 5 (Completeness of ADJ^I). *If $\Psi \vdash_E A_m$ then $\Psi \vdash_I A_m$. Moreover, if $\Psi \Vdash_E A_m$, then $\Psi \Vdash_I A_m$.*

2.2.2 Cut elimination and identity expansion for ADJ^I, directly

While we get cut elimination and identity expansion for free in ADJ^I from the details of soundness and completeness, these results are not very enlightening, due to their indirectness. We will briefly note here that we can provide direct proofs of both cut elimination and identity expansion for ADJ^I. Both follow standard methods, and so we will not dwell on them.

Theorem 6 (Cut elimination for ADJ^I). *If $\Psi \vdash_I A_m$, then $\Psi \Vdash_I A_m$.*

Proof. As is standard, we first prove that the rule of cut is admissible from the other rules of ADJ^I, after which the result follows by induction over the proof that $\Psi \vdash_I A_m$.

Admissibility of cut follows from an induction in the standard manner used to prove admissibility of cut in intuitionistic logic, using cross-cuts to handle contractions. □

Theorem 7 (Identity Expansion). *If $\Psi \vdash_I A_m$, then there exists a proof that $\Psi \vdash_I A_m$ using identity rules only at atomic propositions p_m, which is cut-free if the original proof is.*

Proof. This follows in a standard way by induction over A_m, using paired left and right rules to push the identity up to subformulae of A_m. □

2.3 ADJ^F

As the final presentation of ADJ, we have ADJ^F, a focused calculus for adjoint logic, which is developed using the approach of Simmons [42], using cut elimination and identity expansion for the focused calculus in order to prove focalization.

We begin by assigning polarities to the propositions of ADJ, giving us the following syntax:

$$\begin{array}{lll} \textit{Negative propositions} & A_m^-, B_m^- & ::= \; p_m^- \mid A_m^+ \multimap_m B_m^- \mid \&_{j \in J} {A_m^j}^- \mid {\uparrow}_k^m A_k^+ \\ \textit{Positive propositions} & A_m^+, B_m^+ & ::= \; p_m^+ \mid A_m^+ \otimes_m B_m^+ \mid \mathbf{1}_m^+ \mid \oplus_{j \in J} {A_m^j}^+ \mid {\downarrow}_m^\ell A_\ell^- \end{array}$$

Here, p_m^+ and p_m^- are positive and negative atomic propositions, respectively.

In this polarization, we have chosen to make both $\downarrow$ and $\uparrow$ shifts reverse polarity. We believe that other polarizations are possible, which may streamline some encodings, but as the polarity-reversing shifts are sufficient here, we leave the polarity-preserving shifts to potential future work.

Again closely following Simmons [42], we use the following grammar for the components of our sequents:

$$\begin{array}{llcl} \textit{Stable antecedents} & \Psi & ::= & \cdot \mid A_m^- \mid \langle A_m^+ \rangle \mid \Psi, \Psi' \\ \textit{Inversion antecedents} & \Omega & ::= & \cdot \mid A_m^+ \bullet \Omega \\ \textit{Succedents} & U & ::= & [A_m^+] \mid A_m^+ \mid A_m^- \mid \langle A_m^- \rangle \\ \textit{Ordered antecedents} & L & ::= & \Omega \mid [A_m^-] \end{array}$$

We use a large centered dot $\bullet$ rather than a comma to separate propositions in the (ordered) inversion context Ω to emphasize that those contexts are to be treated as lists rather than as multisets.

Just as we distinguish stable and inversion antecedents, we will refer to the succedents A_m^+ and $\langle A_m^- \rangle$ as stable succedents.

We use square brackets to denote propositions in focus (e.g. $[A_m^+], [A_m^-]$). Likewise, we use angle brackets to denote suspended propositions ($\langle A_m^+ \rangle, \langle A_m^- \rangle$). As atomic propositions are intended to represent arbitrary formulae, we cannot break them down in the inversion phase (as doing so would require knowing more than just their polarity). Instead, in our focused system (Figure 3), we suspend atomic propositions, making it possible for them to appear in stable sequents as antecedents with otherwise positive and in succedents with otherwise negative propositions, respectively. The use of arbitrary suspended propositions is a technical device introduced by Simmons [42] that allows for a structural proof of identity expansion for the focused system, and so we leave this in our system. Moreover, allowing our sequents to contain arbitrary suspended propositions means that we can substitute an arbitrary proposition $A_m^\pm$ for an atomic proposition $p_m^\pm$ while staying within the confines of the system.

Using these parts, we have the following three types of sequents:

$$\begin{array}{lll} \textit{Right focus} & \Psi \vdash_F [A_m^+] & \\ \textit{Inversion} & \Psi \, ; \Omega \vdash_F U & (\text{where } U \neq [A_m^+]) \\ \textit{Left focus} & \Psi \, ; [A_m^-] \vdash_F U & (\text{where } U \text{ is } \textit{stable}) \end{array}$$

Each of these sequents is a special case of the general form $\Psi \, ; \, L \vdash_F U$, but it is useful to separate these cases for some theorem statements and proofs. The constraints on what form U may take in each sequent are standard for intuitionistic focused systems [25, 42], and serve to ensure that at most one formula is in focus at a time, and that if a formula is in focus, then there are no formulae in inversion.

Most of the rules of ADJ^F arise straightfowardly from their ADJ^I counterparts by having the principal formula either in focus or in inversion (depending on its polarity and whether it is on the left or the right — we focus on positive formulae on the right and negative formulae on the left, and likewise, we invert negative formulae on the right, and positive formulae on the left). We also add the $\mathsf{focus}^\pm$ rules, which allow us to bring formulae into focus. The $\mathsf{susp}^\pm$ rules likewise allow us to suspend atomic formulae that we can no longer break down. Finally, the $\mathsf{id}^\pm$ rules bring suspended and focused formulae together, using one to prove the other.

2.3.1 Soundness and Completeness

As before, this calculus is sound and complete with respect to the prior two. In particular, it is easiest to show that it is sound and complete with respect to ADJ^I. In order to state the soundness and completeness theorems (or defocalization and focalization), we must first give a concept of erasure. Informally, we think of $(\cdot)^\bullet$ as removing all focusing and suspension brackets, as well as all polarity information from the proposition or context being erased. This can, of course, be defined formally and extended to sequents as well, but the details are uninteresting, and so are omitted here.

With an erasure operation, we are now equipped to give our soundness and completeness or defocalization and focalization theorems.

Theorem 8 (Defocalization). *If $\Psi \,;\, L \vdash_F U$, then $(\Psi)^\bullet, (L)^\bullet \vdash_I (U)^\bullet$.*

Proof. We prove this by noting that each (erased) rule of the focused system is either a rule of ADJ^I or (in the case of the $\mathsf{focus}^\pm$ and $\mathsf{susp}^\pm$ rules) a no-op. As such, we translate the ADJ^F proof into an ADJ^I proof rule-by-rule, removing the no-op rules. □

Theorem 9 (Cut admissibility for ADJ^F). *Assuming $\mathsf{C} \in \sigma(\Psi_2)$, $\Psi_1, \Psi_2 \geq m$, and that Ψ_1, Ψ_2, Ψ_3, and U contain no non-atomic suspended propositions:*

1. *If $\Psi_1, \Psi_2 \vdash_F [A_m^+]$ and $\Psi_2, \Psi_3 \,;\, A_m^+ \bullet \Omega \vdash_F U$, then $\Psi_1, \Psi_2, \Psi_3 \,;\, \Omega \vdash_F U$.*
2. *If $\Psi_1, \Psi_2 \,;\, \cdot \vdash_F A_m^-$ and $\Psi_2, \Psi_3 \,;\, [A_m^-] \vdash_F U$ and U stable, then $\Psi_1, \Psi_2, \Psi_3 \,;\, \cdot \vdash_F U$.*
3. *If $\Psi_1, \Psi_2 \,;\, L \vdash_F A_m^+$ and $\Psi_2, \Psi_3 \,;\, A_m^+ \vdash_F U$ and U stable, then $\Psi_1, \Psi_2, \Psi_3 \,;\, L \vdash_F U$.*
4. *If $\Psi_1, \Psi_2 \,;\, \cdot \vdash_F A_m^-$ and $\Psi_2, \Psi_3, A_m^- \,;\, L \vdash_F U$ and U stable, then $\Psi_1, \Psi_2, \Psi_3 \,;\, L \vdash_F U$.*

Proof. This proceeds in a relatively standard nested induction, except that cases (3) and (4) depend on cases (1) and (2), respectively. As such, we prove this by induction over the (lexicographically ordered) quadruple $(A_m^\pm, i, \mathcal{D}, \mathcal{E})$, where $A_m^\pm$ is the formula cut out, i is the case number in the theorem statement, $\mathcal{D}$ is the left-hand proof of the cut, and $\mathcal{E}$ is the right-hand proof of the cut.

This proof relies critically on admissibility of weakening and contraction (where permitted by the mode) in ADJ^F, both of which follow from standard structural inductions on the proofs in question. □

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi, \langle A_m^+ \rangle \;;\; \cdot \vdash [A_m^+]}\ \mathsf{id}^+$$

$$\frac{\Psi_1 \geq m \quad \mathsf{W} \in \sigma(\Psi_2) \quad \Psi_1 \;;\; \cdot \vdash A_m^-}{\Psi_1, \Psi_2 \vdash [{\downarrow_k^m} A_m^-]}\ {\downarrow}R \qquad \frac{i \in J \quad \Psi \vdash \left[{A_m^i}^+\right]}{\Psi \vdash \left[\oplus_{j \in J} {A_m^j}^+\right]}\ \oplus R^i$$

$$\frac{\mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash [A_m^+] \quad \Psi_2, \Psi_3 \vdash [B_m^+]}{\Psi_1, \Psi_2, \Psi_3 \vdash [A_m^+ \otimes B_m^+]}\ \otimes R \qquad \frac{\mathsf{W} \in \sigma(\Psi)}{\Psi \vdash [\mathbf{1}_m^+]}\ \mathbf{1}R$$

$$\frac{\Psi \vdash [A_m^+]}{\Psi \;;\; \cdot \vdash A_m^+}\ \mathsf{focus}^+ \qquad \frac{U \text{ stable} \quad \Psi, (A_m^-)^\alpha \;;\; [A_m^-] \vdash U}{\Psi, A_m^- \;;\; \cdot \vdash U}\ (\mathsf{focus}^-)^\alpha$$

$$\frac{\Psi, \langle p^+ \rangle \;;\; \Omega \vdash U}{\Psi \;;\; p^+ \bullet \Omega \vdash U}\ \mathsf{susp}^+ \qquad \frac{\Psi \;;\; \cdot \vdash \langle p^- \rangle}{\Psi \;;\; \cdot \vdash p^-}\ \mathsf{susp}^-$$

$$\frac{\Psi, A_m^- \;;\; \Omega \vdash U}{\Psi \;;\; {\downarrow_k^m} A_m^- \bullet \Omega \vdash U}\ {\downarrow}L \qquad \frac{\Psi \;;\; {A_m^j}^+ \bullet \Omega \vdash U \quad \text{for all } j \in J}{\Psi \;;\; \oplus_{j \in J} {A_m^j}^+ \bullet \Omega \vdash U}\ \oplus L$$

$$\frac{\Psi \;;\; A_m^+ \bullet B_m^+ \bullet \Omega \vdash U}{\Psi \;;\; A_m^+ \otimes B_m^+ \bullet \Omega \vdash U}\ \otimes L \qquad \frac{\Psi \;;\; \Omega \vdash U}{\Psi \;;\; \mathbf{1}_m^+ \bullet \Omega \vdash U}\ \mathbf{1}L$$

$$\frac{\Psi \;;\; \cdot \vdash A_k^+}{\Psi \;;\; \cdot \vdash {\uparrow_k^m} A_k^+}\ {\uparrow}R \qquad \frac{\Psi \;;\; \cdot \vdash {A_m^j}^- \quad \text{for all } j \in J}{\Psi \;;\; \cdot \vdash \&_{j \in J} {A_m^j}^-}\ \& R$$

$$\frac{\Psi \;;\; A_m^+ \vdash B_m^-}{\Psi \;;\; \cdot \vdash A_m^+ \multimap B_m^-}\ \multimap R$$

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi \;;\; [A_m^-] \vdash \langle A_m^- \rangle}\ \mathsf{id}^-$$

$$\frac{k \geq \ell \quad \Psi \;;\; A_k^+ \vdash U_\ell}{\Psi \;;\; [{\uparrow_k^m} A_k^+] \vdash U_\ell}\ {\uparrow}L \qquad \frac{i \in J \quad \Psi \;;\; \left[{A_m^i}^-\right] \vdash U}{\Psi \;;\; \left[\&_{j \in J} {A_m^j}^-\right] \vdash U}\ \& L^i$$

$$\frac{\Psi_1, \Psi_2 \geq m \quad \mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash [A_m^+] \quad \Psi_2, \Psi_3 \;;\; [B_m^-] \vdash U}{\Psi_1, \Psi_2, \Psi_3 \;;\; [A_m^+ \multimap B_m^-] \vdash U}\ \multimap L$$

Figure 3: Focused Adjoint Logic (ADJ^F). "U stable" means that U is either A_m^+ or $\langle p_m^- \rangle$.

Theorem 10 (Admissibility of Identity for ADJ^F). *If* $\mathsf{W} \in \sigma(\Psi)$, *then:*

1. *For any* A_m^+, $\Psi \; ; A_m^+ \vdash_F A_m^+$.
2. *For any* A_m^-, $\Psi, A_m^- \; ; \cdot \vdash_F A_m^-$.

Proof. This result proceeds by first showing that we may suspend arbitrary propositions, rather than only atomic propositions, again via a standard induction. Once we are able to suspend arbitrary propositions, combining this with the $\mathsf{focus}^\pm$ rules and the $\mathsf{id}^\pm$ rules gives the desired result immediately. □

Theorem 11 (Focalization). *If* $(\Psi^-)^\bullet \vdash_I (A_m^+)^\bullet$, *then* $\Psi^- \; ; \cdot \vdash_F A_m^+$

Proof. We first note that cut elimination for ADJ^I allows us to only consider cut-free proofs.

We then proceed by induction over the proof of $(\Psi^-)^\bullet \vdash_I (A_m^+)^\bullet$.

In each case other than the identity cases, we apply the inductive hypothesis to the premise(s) of the last rule used, and then cut the result of this together with the result of applying that rule to identities. This makes use of several lemmas, including cut admissibility for ADJ^F, admissibility of weakening and contraction in ADJ^F, and admissibility of a general identity rule in ADJ^F. □

2.4 Embeddings of other logics

In order to gauge the expressive power of adjoint logic, we will briefly discuss a variety of logics from the computer science literature and how they can be embedded into either ADJ^E or ADJ^I. In the interest of brevity, we will be fairly informal, only sketching how the embeddings work, and avoiding theorems, as the theorem statements alone tend to involve quite a bit of painstaking (but not terribly interesting) details.

Example 1 (Linear logic). *We obtain intuitionistic linear logic [16, 4, 17] by using two modes,* U *(for* unrestricted *or structural) and* L *(for* linear*) with* $\mathsf{U} > \mathsf{L}$. *Moreover,* $\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$ *and* $\sigma(\mathsf{L}) = \{\,\}$, *and the structural layer contains only propositions of the form* $\uparrow_\mathsf{L}^\mathsf{U} A_\mathsf{L}$.

In this representation the exponential modality is decomposed into shift modalities $!A_\mathsf{L} = \downarrow_\mathsf{L}^\mathsf{U} \uparrow_\mathsf{L}^\mathsf{U} A_\mathsf{L}$, *while all other connectives are embedded as themselves.*

Example 2 (LNL). *We obtain Benton's LNL [5] just like linear logic with two modes* $\mathsf{U} > \mathsf{L}$, *but here, rather than only allowing* $\uparrow_\mathsf{L}^\mathsf{U} A_\mathsf{L}$ *at the structural or unrestricted layer, here, we allow the full set of (multiplicative) connectives, where we write* $\times = \otimes_\mathsf{U}$ *and* $\to = \multimap_\mathsf{U}$. *Of course, we can work with additive connectives as well, but LNL avoids them for technical reasons, and so we omit them here.*

Benton's notation for shifts uses F *and* G *for down and up, respectively, and uses two separate contexts to separate linear and unrestricted propositions, whereas we mix both types of propositions into a single context, and rely on the declaration of independence to force that unrestricted succedents depend only on unrestricted antecedents.*

As we have mentioned before, some further examples embed more naturally into ADJ^I, because they are primarily structural in nature, and so are generally also presented with implicit structural rules.

Example 3 (Judgmental S4)**.** *The judgmental modal logic S4 [35] arises from two modes* V *(validity) and* U *(truth) with* $\mathsf{V} > \mathsf{U}$ *and* $\sigma(\mathsf{V}) = \sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$*. The declaration of independence here expresses that validity is categorical with respect to truth—that is, a proof of* A_V *may not depend on any hypotheses of the form* B_U*. Previous calculi enforced this instead by separating the antecedents into two zones, much like the linear and unrestricted contexts of linear logic.*

Analogous to the encoding of linear logic, we only need to allow $\uparrow_\mathsf{U}^\mathsf{V} A_\mathsf{U}$ *in the validity layer. Under that interpretation, we encode* $\Box A_\mathsf{U} = \downarrow_\mathsf{U}^\mathsf{V} \uparrow_\mathsf{U}^\mathsf{V} A_\mathsf{U}$*, which is entirely analogous to the representation of* $!A$ *in linear logic. This type of double shift, first up, then down, allows us to model* comonads*.*

Note that we cannot easily model $\Diamond A$*, which is not a* normal *modality in the technical sense that it does* not *satisfy* $\Diamond(A \multimap B) \multimap (\Diamond A \multimap \Diamond B)$*. Reed [40] provides a less direct, but adequate encoding, while Licata et al. [27] use the* 2*-categorical structure that generalizes our preorder to provide a more elegant representation, so this is expressible in an adjoint framework, if not in the way that we work with it here.*

Example 4 (Lax logic)**.** *Lax logic [12, 35] encodes a weaker form of truth called* lax truth*. We can represent it as a substructural adjoint logic with two modes,* $\mathsf{U} > \mathsf{X}$*, and* $\sigma(\mathsf{U}) = \sigma(\mathsf{X}) = \{\mathsf{W}, \mathsf{C}\}$*. As in the previous examples, we restrict the lax layer* X *to a single form of proposition* $\downarrow_\mathsf{X}^\mathsf{U} A_\mathsf{U}$*, which is sufficient for us to define the lax modality* $\bigcirc A_\mathsf{U} = \uparrow_\mathsf{X}^\mathsf{U} \downarrow_\mathsf{X}^\mathsf{U} A_\mathsf{U}$*. All other connectives are straightforwardly encoded as themselves.*

The double shift, first down, then up, that we use to represent the lax modality is an example of a (strong) monad. In fact, such double shifts will always form strong monads.

2.5 Further remarks in pure logic

Presented with the name "adjoint logic", it is natural to ask what the name means and what relation it has to other notions of adjointness, particularly that of category theory. This can be answered informally by saying that the shifts $\uparrow$ and $\downarrow$ form an adjunction, with $\uparrow$ left adjoint to $\downarrow$. This can be made precise by defining suitable categories and extending the shifts to functors, but such details are unnecessary here. Licata et al. [27] take a more categorical approach to adjoint logic, which, despite its somewhat different presentation, may provide more insight on the adjunctions between the shifts. The work of Benton on LNL [5] is closer to our presentation of adjoint logic, restricted to a specific two-mode case, and also discusses the adjunction between the shifts (which he calls F and G rather than $\uparrow$ and $\downarrow$).

3 From Logics to Languages

In some sense, it is straightforward to turn a logic into a type system, and then on into a language, by simply reinterpreting the propositions as types, writing down some proof terms, and providing a dynamic operational semantics for the terms, often based on some form of proof reduction. However, there are a variety of choices to be made here. The logic on its own does not determine a language fully — for instance, natural deduction presentations of logics tend to be well-suited to lambda calculus-like languages, while Hilbert-style deduction systems yield combinator calculi. Moreover, some of the finer details of the presentation, such as whether we choose to make structural rules explicit or implicit, will affect the resulting language. Finally, even once we have settled on a set of logical rules, our choice of operational semantics will affect the resulting language.

The two versions of the language that we will discuss arise from two different sets of choices of how to present the logic and how to present the dynamics. The first language that we will explore is based on a message-passing interpretation of a form of the logic which makes structural rules explicit as a form of multicut, and the second is based on a shared-memory interpretation of a presentation of the logic with implicit structural rules. We believe that the choice of shared-memory or message-passing is independent from the choice of whether to leave structural rules implicit, to make them explicit, or to fold them into multicut, and so at least four more languages of this nature are possible, but have not yet been explored.

The presentations of adjoint logic in Section 2 are all in the sequent calculus style, which, as in Caires and Pfenning's session-typed π-calculus $\pi\texttt{DILL}$ [8], lends itself well to a synchronous concurrent interpretation, using cut reductions as the basis of computation. We build on this, replacing the sequent $A_1, A_2, \ldots, A_n \vdash B$ with the typing judgment $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash P :: (y : B)$. We can interpret this in several different ways, yielding different sorts of semantics for similar languages. In order to give a message-passing concurrent semantics, we read this as saying that a process P, which may *use* channels x_i and *provides* channel y, along which it may communicate according to protocols A_i and B, respectively. Alternately, we can get a shared-memory semantics by saying instead that P may *read from* addresses x_i, and *writes to* address y, with the data at those addresses having types A_i and B, respectively.

However, in the adjoint setting, synchronous communication makes less sense than in the linear setting that Caires and Pfenning work in. Since a channel may be used by any number of processes (and a memory address may have any number of readers), it is unclear how synchronous communication would work. A channel with zero users, for instance, would cause the provider to get stuck, while a channel with multiple users has several possible behaviors, all of which are problematic. If the provider of the channel sends a message, one can conceivably wait until all users are ready to receive the message before the provider and users can proceed. If instead, however, the provider is receiving a message from the users, this breaks down — would the provider need to wait for all of the users to send messages? What happens if two users send different messages? Rather than trying to address these issues in a synchronous setting, we will instead reformulate our presentations of the logic in order to work with *asynchronous* communication, which we believe to be more natural outside of the linear setting. This also has the benefit that shared memory is most naturally thought of as asynchronous, in that a memory cell is passive, and is only interacted with by one active thread at a time, either to read or to write.

It is somewhat easier to think about asynchrony in the message-passing setting, and so we will focus on it when looking at how to give an asynchronous presentation of adjoint logic. However, the changes to the logic that allow for asynchronous message-passing will also let us give an asynchronous shared-memory semantics. We take inspiration from the asynchronous π-calculus, [7, 20] which replaces the usual output prefix $x\langle y\rangle.P$ with a stand-alone process $x\langle y\rangle$ with no continuation process, which serves as a message. Analogously, we will replace some of our rules (the ones which send, intuitively) with zero-premise axioms, giving a *semi-axiomatic* [11] presentation of the logic.

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

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

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

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

and in the other direction we use identity

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

to derive the other rules.

In the asynchronous π-calculus, instead of explicitly sending a message $x\langle y\rangle.\,P$, we spawn a new process in parallel: $x\langle y\rangle \mid P$. This allows the message to be sent while P continues execution, and in our system corresponds to the cuts used above to construct the usual $\oplus R$ rules from the axiomatic versions.

Receiving a message is then achieved by cut reduction:

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

We see that 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.

As discussed in detail by de Young et al. [11], using a semi-axiomatic calculus loses the standard cut elimination theorem[2], but retains a weaker normalization theorem. A normal proof will only have cuts on subformulae of formulae already in the context, which is sufficient to give the subformula property. The actual restriction for normal forms is a bit stronger, but even this weaker condition is sufficient to give the properties that we want for a logic. Moreover, the languages we will work with based on semi-axiomatic calculi will continue to have progress and preservation results, which is, in the end, what matters for a language.

4 Message Passing

The message-passing language that we will examine here is based on a (semi-axiomatic) presentation of adjoint logic in which structural rules are implemented as multicuts. In the resulting language, management of channels is explicit — a process that wishes to discard ar duplicate a channel can do so with specific forms of multicut. This precise control of unused and reused channels allows for a garbage collection-type result showing that unused processes are automatically removed as part of the computation.

We present the language (Table 1) and its typing rules (Figure 4). The language is broken into three syntactic classes: values V, continuations K, and processes P. Values represent the messages that can be sent, while continuations represent processes waiting to receive a message.

Values and continuations come in matching pairs, with each dual pair of types ($\oplus$ and $\&$, $\otimes$ and $\multimap$, $\downarrow$ and $\uparrow$) having one shape of value and a continuation prepared to receive that shape of value. A key operation in the semantics, used when a message is received, is that of passing a value V to a matching continuation K to get a new process P. Intuitively, a process executing K receives a message V, and then continues as P (although we will see that it is not always quite so simple). This operation is written $V \circ K$ (pass V to K), and is defined as follows:

$$\begin{array}{lcll}
\ell(c) \circ (j(x) \Rightarrow P_j)_{j \in J} & = & P_\ell[c/x] & (\oplus, \&) \\
\langle a, b\rangle \circ (\langle x, y\rangle \Rightarrow P) & = & P[a/x, b/y] & (\otimes, \multimap) \\
\langle\,\rangle \circ (\langle\,\rangle \Rightarrow P) & = & P & (\mathbf{1}) \\
\mathbf{shift}(c_k) \circ (\mathbf{shift}(x_k) \Rightarrow P) & = & P[c_k/x_k] & (\downarrow, \uparrow)
\end{array}$$

The operational semantics presented here will use one type of computational artifact, written $\mathsf{proc}(S, \Delta, a, P)$. Such an object represents a running process executing P, which communicates along the channel a. Δ and S are both sets of endpoints — Δ is the set of endpoints that P uses, while S is the set of endpoints that link to a and may be used by clients. Each endpoint $c \in S$ is used by at most one client, but a single client may use multiple of the endpoints in S. Note that as the endpoints in S are the only way to interact with the channel a from an external process, objects $\mathsf{proc}(S, \Delta, a, P)$ and $\mathsf{proc}(S, \Delta, b, P[b/a])$ are equivalent — changing the internal name of the channel provided by a process has no effect on its interactions with other processes. Note

[2]For example, the sequent $\cdot \vdash \mathbf{1} \oplus \mathbf{1}$ has no cut-free proof since no rule matches this conclusion.

Value V	Meaning
$\langle\rangle$	The terminal value, sent to close a channel
$\ell(c)$	A label ℓ, along with the continuation channel c
$\langle a, b\rangle$	A pair of channels a and b. Here, a is the message, while b is the continuation channel.
$\mathsf{shift}(c_k)$	A shift message indicating a shift to mode k, along with the continuation channel c_k

Continuation K	Meaning
$(\langle\rangle \Rightarrow P)$	Given a terminal value, continue as P
$(j(x) \Rightarrow P_j)_{j\in J}$	Given the value $\ell(c)$ for some $\ell \in J$, continue as $P_\ell[c/x]$.
$(\langle x, y\rangle \Rightarrow P)$	Given the value $\langle a, b\rangle$, continue as $P[a/x, b/y]$.
$(\mathsf{shift}(x_k) \Rightarrow P)$	Given the value $\mathsf{shift}(c_k)$, continue as $P[c_k/x_k]$.

Process term P	Meaning
$c \leftarrow a$	Identify endpoints a and c
$S \leftarrow (\nu x)P \,;\, Q$	Spawn a new process P providing endpoint x, which is linked to the endpoints in the set S, and continue as Q. Here, x is the *internal name* in P for the newly created channel, while the names in S are the *external names* for this new channel, and may be used in Q.
$c.V$	Send value V along the channel with endpoint c.
$\mathsf{case}\, c\, K$	Receive a message from the channel with endpoint c, and pass that message to the continuation K.

Table 1: The values, continuations, and processes of our language.

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

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

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

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

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

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

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

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

Figure 4: Message-passing typing rules

that messages, rather than being distinguished by a separate type of object $\mathsf{msg}(S, \Delta, a, V)$, for example, are simply represented as processes. The process $\mathsf{proc}(S, \Delta, a, a.V)$ which sends a message V along a is itself the message. It is possible to give a version of the semantics that distinguishes these, in order to add an extra step of message-sending, but this complicates the semantics for little benefit.

The state of a running program is then a multiset of these process objects, which we refer to as a *process configuration*. We maintain the invariant that for any two processes $\mathsf{proc}(S_1, \Delta_1, a_1, P_1)$, $\mathsf{proc}(S_2, \Delta_2, a_2, P_2)$, S_1 and S_2 are disjoint, ensuring that no two processes try to provide the same endpoint.

We can then define configurations with the following grammar (and the additional constraint of our invariant):

$$\text{Configurations} \quad \mathcal{C} \quad ::= \quad \cdot \mid \mathsf{proc}(S, \Delta, a, P) \mid \mathcal{C}_1, \mathcal{C}_2$$

We think of the join $\mathcal{C}_1, \mathcal{C}_2$ of two configurations as a commutative and associative operation so that this grammar defines a multiset rather than a list or tree. Additionally, while our configurations are not ordered, we will adopt the convention that the provider of an endpoint appears to the left of any clients of that endpoint.

Our semantics will be given in terms of multiset rewriting rules [10] which can operate on process configurations. A multiset rewriting rule $\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$ can only be applied to a configuration $\mathcal{C}$ which contains $\phi_1, \ldots, \phi_n$. It consumes these objects, and replaces them with $\psi_1, \ldots, \psi_m$. Rules may be applied to any subconfiguration, leaving the remainder of the configuration unchanged.

Putting all of these pieces together, we end up with the following set of rules:

$$\begin{array}{rcl}
\mathsf{proc}(T \cup \{c\}, \Delta, x, P) & & \\
\mathsf{proc}(S, \{c\}, y, y \leftarrow c) & \overset{\mathsf{id}}{\Longrightarrow} & \mathsf{proc}(T \cup S, \Delta, x, P) \\[1ex]
\mathsf{proc}(T, \Delta_P \cup \Delta_Q, y, S \leftarrow (\nu x) P; Q) & \overset{\mathsf{cut}(S)}{\Longrightarrow} & \mathsf{proc}(S', \Delta_P, x, P) \\
(S' \text{ a fresh set of endpoints matching } S) & & \mathsf{proc}(T, \Delta_Q \cup \{S'\}, y, Q[S'/S]) \\[1ex]
(P \text{ not an identity}) \qquad \mathsf{proc}(\{\}, \Delta, x, P) & \overset{\mathsf{drop}}{\Longrightarrow} & \mathsf{proc}(\{\}, \{b\}, y, y \leftarrow b)_{b \in \Delta} \\[1ex]
 & & \mathsf{proc}(\{b', b''\}, \{b\}, y, y \leftarrow b)_{b \in \Delta} \\
\mathsf{proc}(S \cup T, \Delta, x, P) & \overset{\mathsf{copy}}{\Longrightarrow} & \mathsf{proc}(S, \{b'\}_{b \in \Delta}, x, P[b'/b]) \\
(P \text{ not an identity and } S, T \text{ non-empty}) & & \mathsf{proc}(T, \{b''\}_{b \in \Delta}, x, P[b''/b]) \\[1ex]
\mathsf{proc}(\{b\}, \Delta, x, x.V) & \overset{\mathsf{comm}^+}{\Longrightarrow} & \mathsf{proc}(S, \Delta \cup \Delta', z, V \circ K) \\
\mathsf{proc}(S, \Delta' \cup \{b\}, z, \mathsf{case}\ b\ K) & & \\[1ex]
\mathsf{proc}(\{b\}, \Delta, x, \mathsf{case}\ x\ K) & \overset{\mathsf{comm}^-}{\Longrightarrow} & \mathsf{proc}(S, \Delta \cup \Delta', z, V \circ K) \\
\mathsf{proc}(S, \Delta' \cup \{b\}, z, b.V) & &
\end{array}$$

Figure 5: Computation Rules for a Message-Passing Language

The id rule implements forwarding or identification of channels — the forwarding process $y \leftarrow c$ simply hands its clients off to the provider of c. The cut rule implements process spawning — it creates a new channel with new endpoints, spawns a new process to execute P, and continues as Q. The $\mathsf{comm}^{\pm}$ rules are likewise simple, allowing processes to communicate, with one process receiving a message from another — in the rightwards direction in comm^+, and in the leftwards direction in comm^-. It is the drop and copy rules that are somewhat unusual. These two rules take care of propagating duplication or cancellation of an endpoint. If a process P is no longer needed, then P can terminate, but all of the endpoints that P was using must first be cancelled. The drop rule deals with this, turning a cancelled P into several identities that go on to remove P as a client from other processes. Likewise, the copy rule deals with duplicating resources that P relies on when P is itself expected to produce two results.

At this point we can write down some example processes, though our examples for now are very logical in nature, and are not of as much computational interest. Throughout our process examples in this section, we will use comments (beginning with %) after some lines of a process to describe the channels (and their 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.

For a simple example, the following process witnesses that $A_m \mathbin{\&} B_m$ proves $A_m \otimes B_m$ at modes m which admit contraction, making use of multicut:

Example 5. *For any mode m admitting contraction (i.e., $\mathsf{C} \in \sigma(m)$),*

$$
\begin{array}{ll}
(p : A_m \mathbin{\&} B_m) \vdash P :: (z : A_m \otimes B_m) & \\
P = \{p_1, p_2\} \leftarrow (\nu q)\,(q \leftarrow p); & \%(p_1 : A_m \mathbin{\&} B_m), (p_2 : A_m \mathbin{\&} B_m) \\
\quad x \leftarrow (\nu a)\, p_1.\pi_1(a); & \%(x : A_m), (p_2 : A_m \mathbin{\&} B_m) \\
\quad y \leftarrow (\nu b)\, p_2.\pi_2(b); & \%(x : A_m), (y : B_m) \\
\quad z.\langle x, y\rangle &
\end{array}
$$

The following process witnesses that down shifts distribute over implication, and matches the proof used as an example in Section 2.1.

Example 6. *For modes $k \leq m$ with arbitrary structural properties,*

$$
\begin{array}{ll}
(f : {\downarrow}^m_k(A_m \multimap_m B_m)) \vdash P :: (g : {\downarrow}^m_k A_m \multimap_k {\downarrow}^m_k B_m) & \\
P = \mathsf{case}\ g\ (\,\langle x, y\rangle \Rightarrow & \%\ (f : {\downarrow}^m_k(A_m \multimap_m B_m)), (x : {\downarrow}^m_k A_m) \vdash (y : {\downarrow}^m_k B_m) \\
\quad \mathsf{case}\ f\ (\,\mathsf{shift}(w) \Rightarrow & \%\ (w : A_m \multimap_m B_m), (x : {\downarrow}^m_k A_m) \vdash (y : {\downarrow}^m_k B_m) \\
\quad \mathsf{case}\ x\ (\,\mathsf{shift}(v) \Rightarrow & \%\ (w : A_m \multimap_m B_m), (v : A_m) \vdash (y : {\downarrow}^m_k B_m) \\
\quad \{z\} \leftarrow (\nu z)\, y.\mathsf{shift}(z); & \%\ (w : A_m \multimap_m B_m), (v : A_m) \vdash (z : B_m) \\
\quad w.\langle v, z\rangle\,)) &
\end{array}
$$

The behavior of this process is not terribly interesting, but it illustrates how a proof turns into a process.

Under these semantics, we can prove standard session fidelity and deadlock-freedom theorems, the variants of preservation and progress in this session-typed message passing setting. In order to state these theorems, however, we need to extend our notion of typing for processes to one for full 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 untouched. 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 endpoints declared in Ψ', which must be exactly those used in the typing of P. Moreover, *externally* such a process provides the endpoints $S = \{a_m^1, \ldots, a_m^n\}$, all of the same type A_m. We use the abbreviation $(S : A_m)$ for $a_m^1 : A_m, \ldots, a_m^n : A_m$. 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 2.1.1. 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 $\mathcal{C}'$.

4.1 Recursion

The language presented so far is tightly tied to the logic, which limits the sorts of programs that can be written. In order to allow for a wider variety of examples, we extend this language with recursive types and processes. Because the extension to recursion is largely independent of what the semantics of the language look like, we present here a form of recursion that will also work for the shared-memory semantics in Section 5 with minimal changes.

To the base language, we add a new construct, which allows us to call *named processes*, and will allow for recursion. As is customary in the literature on session types, we use *equirecursive types*, gathered in a global signature Σ, which also contains definitions of (possibly recursive) named processes as well as their types. For each type definition $t = A$, the type A must be *contractive* so that we can treat types *equirecursively* with a straightforward coinductive definition and an efficient algorithm for type equality [14].

A named process p is *declared* as $B_{m_1}^1, \ldots, B_{m_n}^n \vdash p :: A_k$ which means it requires arguments of types $B_{m_i}^i$ (in that order) and provides a result of type A_k. For ease of readability, we may sometimes write in variable names for the arguments and result as well, but they are only needed for the corresponding *definitions* $x \leftarrow p\ y_1, \ldots, y_n = P$.

We can then formally define signatures as follows, allowing definitions of types, declarations of processes, and definitions of processes:

$$\text{Signatures} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, t = A \mid \Sigma, \overline{B_m} \vdash p :: A_k \mid \Sigma, x \leftarrow p\ \overline{y} = P$$

In order for a signature to be valid we require that each declaration $\overline{B_m} \vdash p :: A_k$ has a corresponding definition $x \leftarrow p\ \overline{y} = P$ with $\overline{y : B_m} \vdash P :: (x : A_k)$. This means that all type and process definitions can be mutually recursive.

In the remainder of this paper we assume that we have a fixed valid signature Σ, so we annotate neither the typing judgment nor the computation rules with an explicit signature, except in the rules dealing directly with named processes. We introduce three rules, one of which references Σ, along with a new judgment $\Gamma \vdash \overline{w : B_m}$ to describe when a sequence $\overline{w : B_m}$ of typed variables matches the context Γ:

$$\frac{\overline{B_m} \vdash p :: A_k \in \Sigma \quad \Gamma \vdash \overline{w : B_m}}{\Gamma \vdash z \leftarrow p\, \overline{w} :: (z : A_k)}\ \mathsf{call} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, x : B_m \vdash (\Delta, x : B_m)}\ \mathsf{call\_var} \qquad \frac{}{(\cdot) \vdash (\cdot)}\ \mathsf{call\_empty}$$

The rules call_empty and call_var define the judgment $\Gamma \vdash \overline{w : B_m}$. The call_empty rule allows the empty sequence to match the empty context, while the call_var rule lets sequences be extended. The call rule describes how an invocation $z \leftarrow p\, \overline{w}$ calling the named process p with arguments $\overline{w}$ and destination z may be typed. For such an invocation to be well-typed, the process p needs to have a type $\overline{B_m} \vdash p :: A_k$ specified in Σ, and the arguments $\overline{w : B_m}$ must match Γ.

This additional construct for named processes also yields a new computation rule, describing how a call to a named process evaluates. Operationally, a call $z \leftarrow p\, \overline{w}$ expands to its definition with a substitution $P[\overline{w}/\overline{y}, z/x]$. This is made precise by the following computation rule:

$$\mathsf{proc}(S, \Delta, z, z \leftarrow p\, \overline{w}) \quad \overset{\mathsf{call}}{\Longrightarrow} \quad \mathsf{proc}(S, \Delta, z, P[\overline{w}/\overline{y}, z/x]) \qquad (\text{if } x \leftarrow p\, \overline{y} = P \in \Sigma)$$

Equipped with recursion, we can examine a more programming-focused example, albeit a simple one — that of a pipeline of processes operating on a stream of bits.

Example 7. *We begin by defining a recursive type (technically a type family indexed by modes) $bits_m$ of potentially finite streams of bits:*

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

We can then define processes that operate on these streams of bits — for a simple example, a bit-flipping process that inverts the stream:

$$\begin{array}{l} (x : bits_m) \vdash flip :: (y : bits_m) \\ y \leftarrow flip\ x = \mathsf{case}\ y \ \ \begin{array}[t]{l} (\ \mathsf{b0}(y') \Rightarrow x' \leftarrow (\nu a)(a \leftarrow flip\ y'); x.\mathsf{b1}(x') \\ \mid \mathsf{b1}(y') \Rightarrow x' \leftarrow (\nu a)(a \leftarrow flip\ y'); x.\mathsf{b0}(x') \\ \mid \mathsf{e}(y') \Rightarrow x.\mathsf{e}(y')) \end{array} \end{array}$$

This process simply reads in a bit and then concurrently recurses on the remainder of the stream and passes on a message consisting of the flipped version of that bit and a reference to the flipped remainder of the stream. By composing this process with other processes of the same type (or with itself), we can form a pipeline of processes operating on the stream, reading in messages, processing them, and then sending new messages out to the next process on the pipeline, forming a modified stream.

4.2 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 12 (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 $\Psi \vDash \mathcal{C}' :: \Psi'$. $\square$

4.3 Deadlock-freedom

The progress theorem for a functional language states that an expression is either a value or it can take a step. Our values, however, are not quite the same as functional values. For instance, the closest analogue to the term $\lambda x.e$ that waits for an argument is a process $\mathsf{case}\ x\ (\langle y, z\rangle \Rightarrow P)$ that waits for an input, and so we need to broaden our definition of values slightly.

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

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 the analogue of values in the following progress theorem:

Theorem 13 (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.*[3]

Proof. This proof proceeds by an induction on the derivation of $(\cdot) \vDash \mathcal{C} :: \Psi$, making use of a lemma allowing us to reassociate the joins in a configuration 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 $\overline{\Psi'}$. $\square$

[3] 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.

4.4 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 related work (for example, in [8, 36]) 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.

Some of the prior work on affine session types [13, 28] addresses the problem of garbage collection by using equivalence rules which allow canceled channels to be removed from the context. These approaches, particularly that of Fowler et al. [13], 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).

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 (that is, whose type uses only the positive connectives $\oplus, \otimes, \mathbf{1}, \downarrow$[4]) 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*. More formally:

Definition 3. *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* $\mathcal{C} \rhd \Psi\ (d : A_m^\ell)$, *then* $\mathcal{C}\ \mathsf{proc}(\{c\}, \{d\}, x, x.\ell(d)) \rhd \Psi\ (c : \bigoplus_{i \in I} A_m^i)$.
3. *If* $\mathcal{C} \rhd \Psi\ (d : A_m)$, *then* $\mathcal{C}\ \mathsf{proc}(\{c\}, \{d\}, x, x.\mathsf{shift}(d)) \rhd \Psi\ (c : \downarrow_k^m A_m)$.
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:

[4] Note that this notion of positivity is different from that used in Section 2.3 where we deal with focusing. There, we took the shifts to reverse polarity, so $\downarrow$ could only be applied to a negative proposition, yielding a positive one. Here, when working with purely positive types, we allow $\downarrow$, suggesting a shift that preserves polarity, applying to a positive type and yielding a positive type. We have not investigated the focusing behavior of such polarity-preserving shifts, but it remains a possible item of future work.

Corollary 1. *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 $\cdot \vDash \mathcal{C} :: \Psi$, working right to left as in the proof of Theorem 13. 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.

This garbage collection result is limited to purely positive types, but given a suitable definition of observability at negative types, we believe that it can be extended to all configurations, and can better formalize our intuition that the explicit management of channels via weakening and contraction ensures that all processes left over at the end of computation are externally visible. Such an extension is part of the proposed work.

5 A Shared Memory Semantics

We now examine an example of a shared memory semantics for adjoint logic. This time, we will work with a presentation in which the structural rules are *implicit*. This works naturally with shared memory, as it leads to a semantics where reuse of stored data in memory is not explicit. However, we conjecture that applying the ideas of these semantics to a presentation of adjoint logic with explicit structural rules would give us more precise memory management, with tracking of aliases and explicit frees.

In this interpretation, the typing judgment $x_1 : A_1, \ldots, x_n : A_n \vdash P :: (y : B)$ expresses that a process P *may read* data of type A_i from addresses x_i, and *will write* data of type B to address y. While this shares some similarity to the message-passing semantics (e.g. P is responsible for producing $y : B$ in both cases), the key difference is that while in a message-passing setting, communication proceeds back and forth across channels, here, all communication is in one direction — P reads from the x_i and writes to y, and while P may read from some of the x_i multiple times if they are at modes that permit contraction, critically, it can only write once to y.

While we can reuse most of the syntax from Table 1, since we are working with implicit structural rules, there is no need for multicut, and instead, we use the simple cut $x \leftarrow P \; ; Q$ to spawn a new process P which will write to x, and continue as Q, which may read from x. We show the process terms and their new meanings in Table 2. Note that we subscript addresses as $\cdot_W$ and $\cdot_R$ to clarify which are written to and which read from in a process. This is not meant to be part of the syntax of the language, but rather just an aid for clarity, as the reads and writes here occur in different places than an intuition based on message-passing might suggest.

This interpretation of process terms has a few notable features. In the message-passing setting, the construct $c.V$ always sent a message V, but here, depending on the type of c (and

Process term P	Meaning
$c_W \leftarrow a_R$	Move (at affine and linear modes) or copy (at strict and unrestricted modes) the data stored at a to c
$x \leftarrow P \,;\, Q$	Spawn a new process P which *will write to* x and continue as Q, which *may read from* x.
$c_W.V$	Write the value V to address c. Note that this form requires c to have positive type $(\oplus, \otimes, \mathbf{1}, \downarrow)$.
$\mathsf{case}\, c_R\, K$	Read a value V from address c, and pass it to the continuation K. Again, this form requires c to have positive type.
$\mathsf{case}\, c_W\, K$	Write a continuation K to address c. Note that this form requires c to have negative type $(\&, \multimap, \uparrow)$.
$c_R.V$	Read a continuation K from address c, and pass the value V to it. Again, this form requires c to have negative type.

Table 2: The processes of our language.

therefore also V), $c.V$ may either write V into memory or read a stored continuation K from memory and continue as $V \circ K$. Likewise, $\mathsf{case}\, c\, K$ can either write K to memory or read a stored value V from memory and continue as $V \circ K$. This mismatch between sending (in the message-passing setting) and writing to memory (in the shared-memory setting) leads to most of the key differences in the semantics.

We now present the typing rules in Figure 6. The positive left rules $\oplus L$, $\otimes L$, $\mathbf{1}L$, and $\downarrow L$ are replaced with two separate forms, denoted $\oplus L_\alpha$, for instance, with $\alpha \in \{0, 1\}$. The case where $\alpha = 0$ consumes the principal formula of the rule, while the case where $\alpha = 1$ propagates it to the premise, implicitly duplicating it with contraction. The notation $(x : A_m)^\alpha$ encapsulates this, representing a copy of $(x : A_m)$ that is present only when $\alpha = 1$. Note that the contraction implicit in the $\alpha = 1$ rules means that they all additionally require as a side condition that $C \in \sigma(m)$.

These typing rules are largely similar to those for the message-passing language, with the primary differences arising from implicit contraction and weakening. The computation rules, however, are more significantly different, and it is here that our decision to examine a shared-memory semantics comes into play. We will work with three types of semantic objects here:

1. $\mathsf{thread}(c_m, P)$: a thread executing P with destination c_m
2. $\mathsf{cell}(c_m, \_)$: a cell c_m that has been allocated, but not yet written to
3. $!_m\mathsf{cell}(c_m, W)$: a cell c_m containing W

$$\frac{(\Gamma, \Delta \geq m \geq r) \quad C \in \sigma(\Gamma) \quad \Gamma, \Delta \vdash P :: (x : A_m) \quad \Gamma, \Delta', x : A_m \vdash Q :: (z : C_r)}{\Gamma, \Delta, \Delta' \vdash (x \leftarrow P \,;\, Q) :: (z : C_r)}\ \mathsf{cut}$$

$$\frac{W \in \sigma(\Gamma)}{\Gamma, y : A_m \vdash x \leftarrow y :: (x : A_m)}\ \mathsf{id}$$

$$\frac{(i \in L) \quad W \in \sigma(\Gamma)}{\Gamma, y : A_m^i \vdash x.i(y) :: (x : \oplus_m\{\ell : A_m^\ell\}_{\ell \in L})}\ \oplus R^0$$

$$\frac{\Gamma, (x : \oplus_m\{\ell : A_m^\ell\}_{\ell \in L})^\alpha, y : A_m^\ell \vdash Q_\ell :: (z : C_r) \quad (\text{for all } \ell \in L)}{\Gamma, x : \oplus_m\{\ell : A_m^\ell\}_{\ell \in L} \vdash \textbf{case}\, x\, (\ell(y) \Rightarrow Q_\ell)_{\ell \in L} :: (z : C_r)}\ \oplus L_\alpha$$

$$\frac{\Gamma \vdash P_\ell :: (y : A_m^\ell) \quad (\text{for all } \ell \in L)}{\Gamma \vdash \textbf{case}\, x\, (\ell(y) \Rightarrow P_\ell)_{\ell \in L} :: (x : \&_m\{\ell : A_m^\ell\}_{\ell \in L})}\ \& R \qquad \frac{(i \in L) \quad W \in \sigma(\Gamma)}{\Gamma, x : \&_m\{\ell : A_m^\ell\}_{\ell \in L} \vdash x.i(y) :: (y : A_m^i)}\ \& L^0$$

$$\frac{}{\cdot \vdash x.\langle\,\rangle :: (x : \mathbf{1}_m)}\ \mathbf{1}R^0 \qquad \frac{\Gamma, (x : \mathbf{1}_m)^\alpha \vdash P :: (z : C_r)}{\Gamma, x : \mathbf{1}_m \vdash \textbf{case}\, x\, (\langle\,\rangle \Rightarrow P) :: (z : C_r)}\ \mathbf{1}L_\alpha$$

$$\frac{\Gamma, w : A_m \vdash P :: (y : B_m)}{\Gamma \vdash \textbf{case}\, x\, (\langle w, y\rangle \Rightarrow P) :: (x : A_m \multimap_m B_m)}\ {\multimap}R \qquad \frac{W \in \sigma(\Gamma)}{\Gamma, w : A_m, x : A_m \multimap_m B_m \vdash x.\langle w, y\rangle :: (y : B_m)}\ {\multimap}L^0$$

$$\frac{W \in \sigma(\Gamma)}{\Gamma, w : A_m, y : B_m \vdash x.\langle w, y\rangle :: (x : A_m \otimes_m B_m)}\ \otimes R^0 \qquad \frac{\Gamma, (x : A_m \otimes_m B_m)^\alpha, w : A_m, y : B_m \vdash P :: (z : C_r)}{\Gamma, x : A_m \otimes_m B_m \vdash \textbf{case}\, x\, (\langle w, y\rangle \Rightarrow P) :: (z : C_r)}\ \otimes L_\alpha$$

$$\frac{W \in \sigma(\Gamma)}{\Gamma, y : A_m \vdash x_k.\textbf{shift}(y_m) :: (x : {\downarrow}_k^m A_m)}\ {\downarrow}R^0 \qquad \frac{\Gamma, (x : {\downarrow}_k^m A_m)^\alpha, y : A_m \vdash Q :: (z : C_r)}{\Gamma, x : {\downarrow}_k^m A_m \vdash \textbf{case}\, x_k\, (\textbf{shift}(y_m) \Rightarrow Q) :: (z :: C_r)}\ {\downarrow}L_\alpha$$

$$\frac{\Gamma \vdash P :: (y : A_k)}{\Gamma \vdash \textbf{case}\, x_m\, (\textbf{shift}(y_k) \Rightarrow P) :: (x : {\uparrow}_k^m A_k)}\ {\uparrow}R \qquad \frac{W \in \sigma(\Gamma)}{\Gamma, x : {\uparrow}_k^m A_k \vdash x_m.\textbf{shift}(y_k) :: (y : A_k)}\ {\uparrow}L^0$$

Figure 6: Typing rules ($\alpha \in \{0, 1\}$ with $\alpha = 1$ permitted only if $C \in \sigma(m)$)

Here, we prefix a semantic object with $!_m$ to indicate that it is persistent when $C \in \sigma(m)$, and ephemeral otherwise. Note that empty cells are always ephemeral, so that we can modify them by writing to them, while filled cells may be persistent, as each cell has exactly one writer, which will terminate on writing. We maintain the invariant that in a configuration either $\mathsf{thread}(c_m, P)$ appears together with $\mathsf{cell}(c_m, \_)$, or we have just $!_m\mathsf{cell}(c_m, W)$, as well as that if two semantic objects provide the same address c_m, then they are exactly a $\mathsf{thread}(c_m, P)$, $\mathsf{cell}(c_m, \_)$ pair. While this invariant can be made slightly cleaner by removing the $\mathsf{cell}(c_m, \_)$ objects, this leads to an interpretation where cells are allocated lazily just before they are written. This has some advantages, mostly in reducing the size of running configurations, but it is unclear how to inform the thread which will eventually *read from* a new cell where that cell can be found, and so, in the interest of having a realistically implementable semantics, we just allocate an empty cell when spawning a new thread, allowing both the new thread and its parent to see its location.

We can then define configurations with the following grammar (and the additional constraint of our invariant that no two cells share the same address):

$$\text{Configurations} \quad \mathcal{C} \quad ::= \quad \cdot \mid \mathsf{thread}(c_m, P), \mathsf{cell}(c_m, \_) \mid !_m\mathsf{cell}(c_m, W) \mid \mathcal{C}_1, \mathcal{C}_2$$

As with the message-passing semantics, we think of the join $\mathcal{C}_1, \mathcal{C}_2$ of two configurations as a commutative and associative operation so that this grammar defines a multiset rather than a list or tree, and is therefore amenable to a semantics given in terms of multiset rewriting rules.

Our evaluation rules are based on a few key ideas:

1. Variables represent addresses in shared memory.
2. Cut/spawn allocates new memory cells, and is the only way to do so.
3. Identity/forward moves or copies data between cells (depending on whether the cells are ephemeral or persistent).
4. A process $\mathsf{thread}(c, P)$ will write to the cell at address c and then terminate.
5. A process $\mathsf{thread}(d, Q)$ that is trying to read from $c \neq d$ will wait until the cell with address c is available (i.e. its contents are no longer $\_$), perform the read, and then continue.

These ideas are sufficient to describe the full semantics, given in Figure 7. We see that, as informally described above, cut allocates a new cell and spawns a new thread to fill that cell. Similarly, identity moves or copies cell contents (depending on whether the cell is ephemeral or persistent). Because we have separated out values V and continuations K, we can collapse the remaining rules into four cases — one pair for reading and writing values V, corresponding to the positive left and right rules, and one pair for reading and writing continuations K, corresponding to the negative left and right rules.

Recursion Revisited In order to add recursion to the shared-memory semantics, we can use much the same approach as for the message-passing semantics, as described in Section 4.1. We will retain the same fixed valid signature Σ, which will have the same form (except that the grammar of processes is slightly different in the shared-memory setting). We also retain the call rule, but because we are working with implicit structural rules, we need to modify the $\mathsf{call\_empty}$ and $\mathsf{call\_var}$ rules accordingly:

$$\frac{\Gamma, (x : B_m)^\alpha \vdash \Delta}{\Gamma, x : B_m \vdash (\Delta, x : B_m)}\ \mathsf{call\_var}_\alpha \qquad \frac{W \in \sigma(\Psi)}{\Psi \vdash (\cdot)}\ \mathsf{call\_empty}$$

$\mathsf{thread}(c, x \leftarrow P \; ; \; Q) \mapsto \mathsf{thread}(a, [a/x]P), \mathsf{cell}(a, \_), \mathsf{thread}(c, [a/x]Q)$ (a fresh)	*cut: allocate & spawn*
$!_m\mathsf{cell}(c_m, W), \mathsf{thread}(d_m, d_m \leftarrow c_m), \mathsf{cell}(d_m, \_) \mapsto !_m\mathsf{cell}(d_m, W)$	*id: move or copy*
$\mathsf{thread}(c_m, c_m.V), \mathsf{cell}(c_m, \_) \mapsto !_m\mathsf{cell}(c_m, V)$	$(\oplus R^0, \otimes R^0, \mathbf{1}R^0, \downarrow R^0)$
$!_m\mathsf{cell}(c_m, V), \mathsf{thread}(e_k, \mathbf{case}\, c_m\, K) \mapsto \mathsf{thread}(e_k, V \circ K)$	$(\oplus L, \otimes L, \mathbf{1}L, \downarrow L)$
$\mathsf{thread}(c_m, \mathbf{case}\, c_m\, K), \mathsf{cell}(c_m, \_) \mapsto !_m\mathsf{cell}(c_m, K)$	$(\multimap R, \& R, \uparrow R)$
$!_m\mathsf{cell}(c_m, K), \mathsf{thread}(d_k, c_m.V) \mapsto \mathsf{thread}(d_k, V \circ K)$	$(\multimap L^0, \& L^0, \uparrow L^0)$

Figure 7: Shared memory evaluation rules

The call_empty rule now allows any weakenable context Γ to match the empty sequence $(\cdot)$, while the call_var rule is indexed by α, which, like in the other typing rules, may only be 1 if $C \in \sigma(m)$. This allows a variable at a mode which admits contraction to be used for more than one of the arguments to a named process, in much the same way that the implicit contraction in other typing rules allows variables to be reused.

The computation rule for named processes also needs to change in order to reflect the different semantics, but this change is largely cosmetic, replacing the proc of Section 4.1 with thread.

$$\mathsf{thread}(c_m, c_m \leftarrow p\,\overline{w}) \quad \overset{\mathsf{call}}{\Longrightarrow} \quad \mathsf{proc}(c_m, P[\overline{w}/\overline{y}, c_m/x_m]) \quad (\text{if } x_m \leftarrow p\,\overline{y} = P \in \Sigma)$$

We can now also revisit the bit-flipping example from Section 4.

Example 8. *With the same type*

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

we can define a shared-memory version of the same bit-flipping process. Here, rather than a stream of messages, each representing a bit (or the end of the stream), we think of bits_m as representing a data structure in memory, consisting of a linked list where each cell contains either a stop indicator (e) *or a single bit* b0 *or* b1*. The bit flipping process then becomes one that reads over this linked list and produces a flipped copy of it (which can concurrently be read by the next process in a pipeline).*

$(x : \mathit{bits}_m) \vdash \mathit{flip} :: (y : \mathit{bits}_m)$
$y \leftarrow \mathit{flip}\; x = \mathsf{case}\; y \;\; (\; \mathsf{b0}(y') \Rightarrow x' \leftarrow (x' \leftarrow \mathit{flip}\; y'); x.\mathsf{b1}(x')$
$\qquad\qquad\qquad\qquad\quad \mid \mathsf{b1}(y') \Rightarrow x' \leftarrow (x' \leftarrow \mathit{flip}\; y'); x.\mathsf{b0}(x')$
$\qquad\qquad\qquad\qquad\quad \mid \mathsf{e}(y') \Rightarrow x.\mathsf{e}(y'))$

While the process has changed little syntactically, and still behaves much the same, making a recursive call and concurrently handling the first bit read, the operational interpretation is now *quite different. The recursive call handles the tail of the list, while a memory cell at the head of the newly constructed list is written to with the newly flipped bit and an address pointing to the under-construction tail of the list.*

For another example, if we assume that the bit stream is finite, we can interpret it as a binary number, in which we think of the first bit as least-significant, and the last bit (right before the e) as most-significant. We can then write programs that operate on these numbers:

Example 9. *A simple example is a process succ that reads the bits of a binary number n starting at address y and writes the bits for the binary number $n + 1$ starting at x. This process may block until the input cell (referenced as y) has been written to; the output cells are allocated one by one as needed.*

```
$(y : \mathit{bits}_m) \vdash \mathit{succ} :: (x : \mathit{bits}_m)$
$x \leftarrow \mathit{succ}\ y =$
    $\mathbf{case}\ y\ (\ \mathsf{b0}(y') \Rightarrow x' \leftarrow (x' \leftarrow y')\ ;$          $\%$ alloc $x'$ and copy $y'$ to $x'$
                       $x.\mathsf{b1}(x')$                              $\%$ write $\mathsf{b1}(x')$ to $x$
               $\mid \mathsf{b1}(y') \Rightarrow x' \leftarrow (x' \leftarrow \mathit{succ}\ y')\ ;$   $\%$ alloc $x'$ and spawn $\mathit{succ}\ y'$
                       $x.\mathsf{b0}(x')$                              $\%$ write $\mathsf{b0}(x')$ to $x$
               $\mid \mathsf{e}(y') \Rightarrow x'' \leftarrow (x'' \leftarrow y')\ ;$        $\%$ alloc $x''$ and copy $y'$ to $x''$
                      $x' \leftarrow x'.\mathsf{e}(x'')\ ;$                   $\%$ alloc $x'$ and write $\mathsf{e}(x'')$ to $x'$
                      $x.\mathsf{b1}(x')\ )$                            $\%$ write $\mathsf{b1}(x')$ to $x$
```

This process follows a simple carrying algorithm, replacing $\mathsf{b1}$ *with* $\mathsf{b0}$ *until a* $\mathsf{b0}$ *is found in the original number, which is then replaced with* $\mathsf{b1}$*. If no* $\mathsf{b0}$ *is found (i.e. the number in question is of the form $2^k - 1$), then a new* $\mathsf{b1}$ *is added just before the* e*, making the number one bit longer.*

The rules for typing configurations in this setting differ from those in the message-passing semantics of Section 4 in two key ways. First, since we are working with different semantic objects, the rules need to change slightly, typing a cell containing data W in the same way as the process that immediately writes W and terminates. This change arises from the difference between message-passing and shared-memory. The second change comes from the implicit contraction that cut allows in this setting. Because we are entitled to reuse contractible assumptions without explicitly duplicating them, these assumptions may be used to type a thread or cell without being consumed, and so Γ appears on the right-hand side of the thread and cell typing rules, rather than being split among the right-hand side and the premise.

$$\frac{C \in \sigma(\Gamma) \quad \Gamma, \Psi \vdash P :: (c : A_m)}{\Gamma, \Delta, \Psi \vdash \mathsf{thread}(c, P), \mathsf{cell}(c, \_) :: (\Gamma, \Delta, c : A_m)}$$

$$\frac{C \in \sigma(\Gamma) \quad \Gamma, \Psi \vdash c.V :: (c : A_m)}{\Gamma, \Delta, \Psi \vdash !_m\mathsf{cell}(c, V) :: (\Gamma, \Delta, c : A_m)} \qquad \frac{C \in \sigma(\Gamma) \quad \Gamma, \Psi \vdash \mathbf{case}\, c\, K :: (c : A_m)}{\Gamma, \Delta, \Psi \vdash !_m\mathsf{cell}(c, K) :: (\Gamma, \Delta, c : A_m)}$$

$$\frac{}{\Delta \vdash (\cdot) :: \Delta} \qquad \frac{\Gamma \vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \vdash \mathcal{C}_2 :: \Delta_2}{\Gamma \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_2}$$

Note that while we have no rule for typing an empty cell $\mathsf{cell}(c, \_)$ or a thread $\mathsf{thread}(c, P)$ on its own, we can nevertheless type every well-formed configuration, because of our invariant that empty cells and threads always go together.

As in Section 4, a typing derivation for a given configuration $\mathcal{C}$ induces an ordering on the configuration, although different derivations may induce different orderings, all of which are linear extensions of the partial order defined by dependency, where $\mathsf{thread}(c_1, P_1)$ comes before $\mathsf{thread}(c_2, P_2)$ if P_2 reads from c_1, and similarly for cells.

Using this definition of configuration typing, we are able to give standard variants of preservation and progress. The preservation theorem allows the collection Δ of typed addresses offered by a configuration $\mathcal{C}$ to grow after a step, as cut steps may introduce new cells at persistent modes (which will become truly persistent when they are written to). Note that we could be more precise, saying that either $\Delta' = \Delta$ or Δ' is obtained by adding one new typed address to Δ.

Theorem 14 (Type Preservation)**.** *If $\Gamma \vdash \mathcal{C} :: \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$ then $\Gamma \vdash \mathcal{C}' :: \Delta'$ for some $\Delta' \supseteq \Delta$.*

Progress is entirely standard, with configurations comprised entirely of filled cells taking the role that values play in a functional language. This makes sense because a configuration consisting entirely of filled cells is terminal and cannot step further, but is not stuck in that there are no threads representing active computations.

Theorem 15 (Progress)**.** *If $\cdot \vdash \mathcal{C} :: \Delta$ then either*

(i) $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$, or

(ii) for every channel $c_m : A_m \in \Delta$ there is an object $!_m\mathsf{cell}(c_m, W) \in \mathcal{C}$.

In addition to these type-safety properties, we also have a confluence result, for which we need to define a weak notion of equivalence on configurations to account for the fact that new addresses are created fresh by some steps, and their names are not prescribed by the semantics. We say $\mathcal{C}_1 \sim \mathcal{C}_2$ if there is a renaming ρ of addresses such that $\rho\mathcal{C}_1 = \mathcal{C}_2$. We can then establish the following version of the diamond property:

Theorem 16 (Diamond Property). *Assume $\Delta \vdash \mathcal{C} :: \Gamma$. If $\mathcal{C} \mapsto \mathcal{C}_1$ and $\mathcal{C} \mapsto \mathcal{C}_2$ such that $\mathcal{C}_1 \not\sim \mathcal{C}_2$. Then there exist $\mathcal{C}_1'$ and $\mathcal{C}_2'$ such that $\mathcal{C}_1 \mapsto \mathcal{C}_1'$ and $\mathcal{C}_2 \mapsto \mathcal{C}_2'$ with $\mathcal{C}_1' \sim \mathcal{C}_2'$.*

In this setting without recursion, this gives us a full confluence result via standard inductions, but even once we add recursion, we continue to have this form of local confluence.

5.1 Encoding Sequentiality

One notable feature of the shared-memory semantics is that each cell is write-once. This means that unlike in the message-passing setting, communication proceeds in one direction — once a process has written to memory, it terminates, and cannot read any further. This means that intuitively, at least, we should expect there to be a natural sequential semantics for this language, where processes evaluate in dependency order and all cells are written to before the processes that depend on them begin execution. We have explored such a semantics in [39, v1], where we give a fully sequential semantics for this language, with close relations to destination-passing style [45, 24, 9, 41]. However, while this approach works well for a purely sequential language, the need to work with completely different semantic objects and a completely different set of rules when evaluating sequentially is inconvenient, and so we will focus here on a different approach. Rather than providing sequential semantics completely separately from the concurrent semantics, we will instead look to *encode* sequentiality in the concurrent semantics.

To be precise, when we talk about sequentiality as a feature of semantics from here on, we mean that a language is *sequential* if there is at most one thread (the *active thread*) which can step, and all other threads are blocked waiting to read. This will inform our decisions on what it means for a portion of a language to be sequential. We first observe that the only process construct that can create a new thread is the cut $x \leftarrow P \,;\, Q$. One possible way, then, to enforce sequentiality is to replace this with a *sequential cut* construct that only allows one of P or Q to execute, leaving the other blocked. These two options give us forms of cut that are analogous to call-by-value and call-by-name, both of which we present below.

5.1.1 Call-by-value

To get a call-by-value sequential cut $x \leftarrow P \,;\, Q$, we want P to evaluate fully and write into x before Q begins executing. To implement such a sequential cut, we will take advantage of the fact that a shift from a mode m to itself does not affect provability (in that A_m and $\downarrow_m^m A_m$ are logically equivalent), but does force synchronization. If $x : A_m$, we would like to define the sequential cut

$$x \Leftarrow P \,;\, Q \triangleq x' \leftarrow P' \,;\, \mathbf{case}\ x'\ (\mathbf{shift}(x) \Rightarrow Q),$$

where $x' : \downarrow_m^m A_m$, and, intuitively, P' behaves like P, except that wherever P would write to x, P' writes simultaneously to x and x'. Setting aside for now a formal definition of P' and a discussion of the needed simultaneous writes, we see that Q cannot execute until a shift has been written to x', and so provided that P' writes to x' no later than it writes to x, we are guaranteed that x has been written to before Q can continue.

This condition, that P' writes to x no later than it writes to x, cannot be enforced in the language so far, and so we need to add some new features. A simple way to do this is to add new *atomic write* constructs that will write to x and x' at the same time. We define three new constructs corresponding to the three ways that x can be written to in P, shown here next to the non-atomic process that they behave as:

Atomic Write	Non-atomic equivalent
$x'.\mathbf{shift}(x.V)$	$x \leftarrow x.V \ ; x'.\mathbf{shift}(x)$
$x'.\mathbf{shift}(\mathbf{case}\ x\ K)$	$x \leftarrow \mathbf{case}\ x\ K \ ; x'.\mathbf{shift}(x)$
$x'.\mathbf{shift}(x \leftarrow y)$	$x \leftarrow (x \leftarrow y) \ ; x'.\mathbf{shift}(x)$

The typing rules for the atomic writes are the same as the derived rules to type their non-atomic equivalents:

$$\dfrac{\Delta \vdash x.V :: (x : A_m) \quad \dfrac{}{x : A_m \vdash x'.\mathbf{shift}(x) :: (x' : {\downarrow}^m_m A_m)}\,{\downarrow}R^0}{\Delta \vdash x'.\mathbf{shift}(x.V) :: (x' : {\downarrow}^m_m A_m)}\,\mathsf{cut}$$

$$\dfrac{\Delta \vdash \mathbf{case}\ x\ K :: (x : A_m) \quad \dfrac{}{x : A_m \vdash x'.\mathbf{shift}(x) :: (x' : {\downarrow}^m_m A_m)}\,{\downarrow}R^0}{\Delta \vdash x'.\mathbf{shift}(\mathbf{case}\ x\ K) :: (x' : {\downarrow}^m_m A_m)}\,\mathsf{cut}$$

$$\dfrac{\dfrac{}{\Gamma_W, y : A_m \vdash x \leftarrow y :: (x : A_m)}\,\mathsf{id} \quad \dfrac{}{x : A_m \vdash x'.\mathbf{shift}(x) :: (x' : {\downarrow}^m_m A_m)}\,{\downarrow}R^0}{\Gamma_W, y : A_m \vdash x'.\mathbf{shift}(x \leftarrow y) :: (x' : {\downarrow}^m_m A_m)}\,\mathsf{cut}$$

As with the typing, the behavior of each of these atomic writes is much like that of the non-atomic equivalent. The only difference is that the non-atomic forms take three steps — first for the cut, second to write to x, and third to write to x', all of which are collapsed into one step in the atomic form. This intuition is formalized in the following transition rules:

$$\begin{array}{ll}
\mathsf{thread}(x'_m, x'_m.\mathbf{shift}(x_k.V)) \mapsto !_k\mathsf{cell}(x_k, V), !_m\mathsf{cell}(x'_m, \mathbf{shift}(x_k)) & \textit{atom-val} \\
\mathsf{thread}(x'_m, x'_m.\mathbf{shift}(\mathbf{case}\ x_k\ K)) \mapsto !_k\mathsf{cell}(x_k, K), !_m\mathsf{cell}(x'_m, \mathbf{shift}(x_k)) & \textit{atom-cont} \\
\mathsf{thread}(x'_m, x'_m.\mathbf{shift}(x_k \leftarrow y_k)), !_k\mathsf{cell}(y_k, W) \mapsto !_k\mathsf{cell}(x_k, W), !_m\mathsf{cell}(x'_m, \mathbf{shift}(x_k)) & \textit{atom-id}
\end{array}$$

Note that the rule for the identity case is different from the other two — it requires the cell y_k to have been written to in order to continue. This is because the $x \leftarrow y$ construct reads from y and writes to x — if we wish to write to x and x' atomically, we must also perform the read from y.

Equipped with these atomic operations, we can define an operation $[x'.\textbf{shift}(x) /\!\!/ x]$ on processes that replaces writes to x with atomic writes to both x and x' as follows:

$$\begin{array}{rcl}(x.V)[x'.\textbf{shift}(x) /\!\!/ x] & = & x'.\textbf{shift}(x.V)\\ (\textbf{case}\ x\ K)[x'.\textbf{shift}(x) /\!\!/ x] & = & x'.\textbf{shift}(\textbf{case}\ x\ K)\\ (x \leftarrow y)[x'.\textbf{shift}(x) /\!\!/ x] & = & x'.\textbf{shift}(x \leftarrow y)\end{array}$$

Extending $[x'.\textbf{shift}(x) /\!\!/ x]$ compositionally over our other language constructs, we can define $P' = P[x'.\textbf{shift}(x) /\!\!/ x]$, giving us a full definition of the sequential cut:

$$x \Leftarrow P \ ; Q \triangleq x' \leftarrow P[x'.\textbf{shift}(x) /\!\!/ x] \ ; \textbf{case}\ x'\ (\textbf{shift}(x) \Rightarrow Q).$$

5.1.2 Call-by-name

In contrast to the call-by-value sequential cut, where P was allowed to execute while Q was blocked, to get a call-by-name sequential cut $x \Leftarrow^{\mathsf{N}} P \ ; Q$, we will block P until Q needs to use x. Then, when Q needs to use x, it is necessarily blocked trying to read from x, and so if we ensure that P completes execution before writing to x, this gives sequentiality.

To implement call-by-name, we make use of shifts in a similar way, this time using the up shift. When $x : A_m$, we define

$$x \Leftarrow^{\mathsf{N}} P \ ; Q \triangleq x' \leftarrow \textbf{case}\ x'\ (\textbf{shift}(x) \Rightarrow P) \ ; Q'$$

where $x' : \uparrow_m^m A_m$, and Q' behaves as Q, but wherever Q reads from x, Q' reads first from x', then from x.

To make this formal, we define another substitution operation $[x'.\textbf{shift}(x)\%x]$:

$$\begin{array}{rcl}(x.V)[x'.\textbf{shift}(x)\%x] & = & \textbf{case}\ x'\ (\textbf{shift}(x) \Rightarrow x.V)\\ (\textbf{case}\ x\ K)[x'.\textbf{shift}(x)\%x] & = & \textbf{case}\ x'\ (\textbf{shift}(x) \Rightarrow \textbf{case}\ x\ K)\\ (x \leftarrow y)[x'.\textbf{shift}(x)\%x] & = & \textbf{case}\ x'\ (\textbf{shift}(x) \Rightarrow x \leftarrow y)\end{array}$$

We can then (after extending this operation compositionally) define

$$x \Leftarrow^{\mathsf{N}} P \ ; Q \triangleq x' \leftarrow \textbf{case}\ x'\ (\textbf{shift}(x) \Rightarrow P) \ ; Q[x'.\textbf{shift}(x)\%x]$$

giving us a second form of sequential cut — when $Q[x'.\textbf{shift}(x)\%x]$ attempts to read from x', it will start executing P, but will block until x is written to. Provided that P is also a sequential process, this means that P will terminate, leaving no active threads, before Q can be woken again.

The two different types of sequential cuts have different benefits and drawbacks. Call-by-value requires the addition of atomic writes, while call-by-name does not. However, call-by-name will duplicate the effort of evaluating P if Q reads from x multiple times.

5.1.3 Futures

By restricting the language to disallow the general concurrent cut and only permit sequential cuts, we can get a fully sequential language, with only one active thread at a time (though if we make use of the call-by-name cut, threads can be paused and resumed, rather than running from start to finish all at once). If, instead, we allow both sequential and concurrent cuts, we get a language where the programmer can choose at each thread-spawn whether the new thread should run concurrently with the old one, sequentially before the old one, or sequentially after the old one. Concurrent cuts in this setting implement a form of *futures* [19] — in $x \leftarrow P \; ; Q$, P and Q are evaluated concurrently. When Q tries to read from x, it will block until P has computed a result W and written it to x. If we wish to add an explicit synchronization point, we can do so with minimal overhead by making use of identity to read from x. For instance, the process $z \Leftarrow (z \leftarrow x) \; ; Q$ will first copy or move the contents of cell x to cell z, and then run Q. As such, it delays the execution of Q until x has been written to, even if Q does not need to look at the value of x until later. This is analogous to the touch construct of some approaches to futures.

This approach is the opposite, in some ways, of typical presentations of futures, which add futures as a concurrency construct to a sequential base language. Here, concurrency via futures is the norm, although this does not become apparent until we introduce constructs for sequential computation. The addition of sequential cuts nevertheless gives the same expressive power as a typical sequential language with futures.

While we do not distinguish futures at the type level when using this approach, this is not unusual — Halstead's Multilisp does not have a separate type for futures either, and nor does it require an explicit touch construct. Of course, we can easily translate a use of futures with explicit touch into this language by making use of identities for explicit synchronization. It is also possible (see [39]) to encode a form of futures that do have a distinct type and that require explicit constructs to create and touch futures. However, we find the form of futures given by the concurrent cut more natural — in this formulation, the client of an address x does not need to care in what way the cell contents are computed — it simply uses them, blocking if necessary.

5.1.4 Linear Futures

One interesting aspect of the futures in this language is that they can be worked with at any mode. In particular, this allows us to work with *linear futures* in theoretically well-founded way. Linear futures have been used in analysis of some parallel algorithms, [6] and discussed somewhat in that context, but to our knowledge, had not been formalized prior to this work.

Similarly, we can give a rigorous foundation for affine futures (which must be used at most once) or strict futures (which are guaranteed to be used), although we are not aware of any prior work that makes use of these. More interestingly, however, since we are not limited to working with a single mode, we can in fact define a language which makes use of both linear and non-linear futures. Since an unrestricted mode may not depend on a linear mode, this would consist of a non-linear language (with futures), along with a monad containing a linear language (with linear futures). This limits the ways that linear futures can be used slightly, but since linear futures are *linear*, and must be used exactly once, it is unsurprising that they would be contained inside a linear host language.

5.2 Limited Concurrency

The concurrent cut is an extremely powerful tool for concurrency. Sometimes, however, it is convenient to be more restrictive with our concurrency model, either to make it easier to reason about programs, or to make it easier to implement. By placing restrictions on the language to forbid concurrent cuts in general, only allowing them in specific instances, we can capture concurrency models that are more limited than that of futures.

As a simple example, we can provide access to concurrency between independent computations — a form of fork/join parallelism — by only allowing concurrent cut in the following pattern, where $P_\star$ and $Q_\star$ are independent processes. We write $P_\star$ here to denote a process P that writes to address $\star$. If we think of writing to $\star$ as returning a value, this allows for a more functional-like notation:

$$x' \leftarrow P_\star[x'.\mathbf{shift}(x) /\!\!/ \star] \; ; \; y' \leftarrow Q_\star[y'.\mathbf{shift}(y) /\!\!/ \star] \; ; \; \mathbf{case}\; x' \; (\mathbf{shift}(x) \Rightarrow \mathbf{case}\; y' \; (\mathbf{shift}(y) \Rightarrow z.\langle x, y\rangle))$$

This will spawn two processes, $P_\star$ and $Q_\star$ which can run concurrently, and then block until both have written a result into memory. This is a more restrictive form of concurrency than futures in that we require $P_\star$ and $Q_\star$ to be independent.

More interestingly, we can take advantage of the mode system and allow concurrent cuts at some modes, but not at others. We have more research to do in this line, but as a first example, we can reconstruct a concurrency monad in the style of SILL [44, 43, 18]. SILL combines a session-typed linear concurrent language with a non-linear functional language by encapsulating the concurrent computations in a (contextual) monad. We take a somewhat simplified approach which is still quite expressive.

We will use two modes N and S with $\sigma(\mathsf{N}) = \{\}$, $\sigma(\mathsf{S}) = \{W, C\}$, and $\mathsf{N} < \mathsf{S}$. Intuitively, N contains the concurrent linear portion of the language, while S contains the sequential non-linear portion. We will enforce this by requiring that any cut whose principal formula is at mode S is a sequential cut, guaranteeing that the newly created cell at mode S is filled before it is read from. We will now present several constructs from SILL as temporary extensions to the base language, and then show how they can be treated as syntactic sugar for terms in the base language.

Borrowing notation from SILL, we will use the type $\{A_\mathsf{N}\}$ as shorthand for the monad, and types $A_\mathsf{S} \wedge B_\mathsf{N}$ and $A_\mathsf{S} \supset B_\mathsf{N}$ to send and receive non-linear values in the linear layer, respectively. The type $\{A_\mathsf{N}\}$ has as values process expressions $\{P_\star\}$ such that $P_\star :: (\star : A_\mathsf{N})$. These process expressions can be constructed and passed around in the non-linear layer. In order to actually execute these processes, however, we need to use a *bind* construct $\{c_\mathsf{N}\} \leftarrow Q_\star$ in the functional layer, which will evaluate $Q_\star$ into an encapsulated process expression $\{P_\star\}$ and then run $P_\star$, storing its result in c_N. We can add $\{\cdot\}$ to our language with the typing rules below. Here, Γ_S indicates that all assumptions in Γ are at mode S:

$$\frac{\Gamma_\mathsf{S} \vdash P_\star :: (\star_\mathsf{N} : A_\mathsf{N})}{\Gamma_\mathsf{S} \vdash y_\mathsf{S}.\{P_\star\} :: (y_\mathsf{S} :: \{A_\mathsf{N}\})}\ \{\cdot\}I \qquad \frac{\Gamma_\mathsf{S} \vdash Q_\star :: (\star_\mathsf{S} :: \{A_\mathsf{N}\})}{\Gamma_\mathsf{S} \vdash \{c_\mathsf{N}\} \leftarrow Q_\star :: (c_\mathsf{N} : A_\mathsf{N})}\ \{\cdot\}E$$

Since they live in the session-typed layer, the $\wedge$ and $\supset$ constructs fit more straightforwardly into our language. We will focus on the type $A_\mathsf{S} \wedge B_\mathsf{N}$, but $A_\mathsf{S} \supset B_\mathsf{N}$ can be handled similarly. A process of type $A_\mathsf{S} \wedge B_\mathsf{N}$ should write a pair of a non-linear sequential value with type A_S and a concurrent value with type B_N. These terms and their typing rules are shown below:

$$\frac{}{\Gamma_W, v_\mathsf{S} : A_\mathsf{S}, y_\mathsf{N} : B_\mathsf{N} \vdash x_\mathsf{N}.\langle v_\mathsf{S}, y_\mathsf{N}\rangle :: (x_\mathsf{N} :: A_\mathsf{S} \wedge B_\mathsf{N})}\ \wedge R^0$$

$$\frac{\Gamma, (v_\mathsf{S} : A_\mathsf{S}), (y_\mathsf{N} : A_\mathsf{N}) \vdash P_z :: (z_\mathsf{N} : C_\mathsf{N})}{\Gamma, x_\mathsf{N} : A_\mathsf{S} \wedge B_\mathsf{N} \vdash \mathbf{case}\ x_\mathsf{N}\ (\langle v_\mathsf{S}, y_\mathsf{N}\rangle \Rightarrow P_z) :: (z_\mathsf{N} : C_\mathsf{N})}\ \wedge L$$

To bring these new constructs into the base language by treating them as syntactic sugar, we define

$$\begin{array}{rcl}
\{A_\mathsf{N}\} & \triangleq & \uparrow_\mathsf{N}^\mathsf{S} A_\mathsf{N} \\
A_\mathsf{S} \wedge B_\mathsf{N} & \triangleq & (\downarrow_\mathsf{N}^\mathsf{S} A_\mathsf{S}) \otimes B_\mathsf{N} \\
d_\mathsf{S}.\{P_\star\} & \triangleq & \mathbf{case}\ d_\mathsf{S}\ (\mathbf{shift}(x_\mathsf{N}) \Rightarrow P_x) \\
\{c_\mathsf{N}\} \leftarrow Q_\star & \triangleq & y_\mathsf{S} \Leftarrow Q_y; y_\mathsf{S}.\mathbf{shift}(c_\mathsf{N}) \\
d_\mathsf{N}.\langle v_\mathsf{S}, y_\mathsf{N}\rangle & \triangleq & x_\mathsf{N} \leftarrow x_\mathsf{N}.\mathbf{shift}(v_\mathsf{S}); d_\mathsf{N}.\langle x_\mathsf{N}, y_\mathsf{N}\rangle \\
\mathbf{case}\ d_\mathsf{N}\ (\langle u_\mathsf{S}, w_\mathsf{N}\rangle \Rightarrow P_z) & \triangleq & \mathbf{case}\ d_\mathsf{N}\ (\langle x_\mathsf{N}, w_\mathsf{N}\rangle \Rightarrow \mathbf{case}\ x_\mathsf{N}(\mathbf{shift}(v_\mathsf{S}) \Rightarrow P_z))
\end{array}$$

These definitions give us the following type-correctness theorem:

Theorem 17. *If we expand all new constructs using $\triangleq$, then the typing rules for $\{\cdot\}$ and $\wedge$ are admissible.*

However, it is not enough to know that these definitions are well-typed — we would also like to verify that they have the behavior we expect for a concurrency monad. For both $\{\cdot\}$ and $\wedge$, this is relatively straightforward. Examining the term

$$d_\mathsf{S}.\{P_\star\} \quad \triangleq \quad \mathbf{case}\ d_\mathsf{S}\ (\mathbf{shift}(x_\mathsf{N}) \Rightarrow P_x),$$

we see that this writes a continuation into memory, containing the process P_x. A reference to this continuation can then be passed around freely, until it is executed using the bind construct:

$$\{c_\mathsf{N}\} \leftarrow P_\star \quad \triangleq \quad y_\mathsf{S} \Leftarrow P_y; y_\mathsf{S}.\mathbf{shift}(c_\mathsf{N})$$

This construct first evaluates P_y with destination y_{S}, to get a stored process, and then executes that stored process with destination c_{N}.

The $\wedge$ construct is even simpler. Writing a functional value using the term

$$d_{\mathsf{N}}.\langle v_{\mathsf{S}}, y_{\mathsf{N}}\rangle \quad \triangleq \quad x_{\mathsf{N}} \leftarrow x_{\mathsf{N}}.\mathbf{shift}(v_{\mathsf{S}}); d_{\mathsf{N}}.\langle x_{\mathsf{N}}, y_{\mathsf{N}}\rangle$$

sends both a shift (bringing the functional value into the concurrent layer) and the pair $\langle x_{\mathsf{N}}, y_{\mathsf{N}}\rangle$ of the continuation y_{N} and the shift-encapsulated value x_{N}. Reading such a value using the term

$$\mathbf{case}\ d_{\mathsf{N}}\ (\langle v_{\mathsf{S}}, y_{\mathsf{N}}\rangle \Rightarrow P_z) \quad \triangleq \quad \mathbf{case}\ d_{\mathsf{N}}\ (\langle x_{\mathsf{N}}, y_{\mathsf{N}}\rangle \Rightarrow \mathbf{case}\ x_{\mathsf{N}}(\mathbf{shift}(v_{\mathsf{S}}) \Rightarrow P_z))$$

just does the opposite — we read the pair out of memory, peel the shift off of the functional value v_{S} to return it to the sequential, functional layer, and continue with the process P_z, which may make use of both v_{S} and the continuation y_{N}.

These terms therefore capture the general behavior of a monad used to encapsulate concurrency inside a functional language. The details of the monad we present here are different from that of SILL's (contextual) monad, despite our use of similar notation, but the essential idea is the same. While this is a simple example, making use of only two modes, it demonstrates that we can separate concurrent and sequential computation at the mode level, an idea that merits further exploration. For instance, nothing forces a concurrent mode to be linear or a sequential mode to be non-linear, and nor need concurrent modes necessarily be less than sequential modes.

6 Proposed Work

The primary focus of the proposed work is an exploration of how modularity at the type level (as expressed by the declaration of independence) manifests itself at the level of processes and computation. For instance, just as a proof of A_k can only depend on assumptions B_m if $m \geq k$, a computation that yields a result of type A_k should only depend on subcomputations at mode m where $m \geq k$. In particular, if such a computation does some work at mode $\ell < k$, then that work should be *invisible* to an observer of the final result. This idea fits naturally in with ideas of parametricity and modularity — mode k should not need to know how computations at mode ℓ work. For instance, we might think of a system with runtime debugging checks at a lower mode than the overall computation — in a smoothly running system, these checks can be disabled without any effect on the results.

While the details are still unclear, the ideal end result of this line of work would be a general theorem describing how the semantics at different modes interact and what sorts of constraints the preorder places on what the semantics can look like at different modes. Just as the cut elimination results from Sections 2.1 to 2.3 or the focusing result from Section 2.3 give us cut elimination and focusing for a variety of logics that can be represented in the adjoint framework, such a theorem would give us a way to get information about isolation and modularity for any language that can be represented in the adjoint framework (though that framework may need some additions to account properly for different modes having different semantics).

The general idea of type-level isolation of portions of a language is not a new one, and has come up in several different ways. The use of monads to introduce effects to a pure functional language, for instance, is an example. The functional language, at the higher level, cannot see inside the monad, at the lower level. In a less programming-directed setting, Pfenning [34] develops a type theory with three layers embodying intensionality, extensionality, and proof-irrelevance. Just as with monads, we can think of the higher layers as being independent of the lower layers. Here, that manifests itself in the equations that hold at each layer — at the lowest, proof-irrelevant layer, any two terms of the same type are equal. However, these equalities are not inherited at the higher layers, with the middle, extensional layer having fewer equalities, and the top, intensional layer having fewer still. We believe that both of these are examples of a more general framework for isolation and modularity, which has some underlying basis in, if not exactly the adjoint logic presented in Section 2, then a similar logic.

In other work on concurrency, particularly in the space of concurrent separation logic, *auxiliary variables* [32] or *ghost state* [21] make use of isolation in a similar way — ghost state is relevant only to proofs, not to the actual execution of the program that is the subject of the proof. As such, it seems that we may be able to think of a combined program-proof as one object with parts in two different modes — an upper mode that is computational, and a lower mode that is only relevant for the proof.

In order to explore the idea of isolation (or modularity or parametricity) in this setting, it is likely that we will need to further investigate notions of equivalence for processes typed with our adjoint type system(s). For instance, we are not aware of a good notion of bisimulation or of observational equivalence for a shared-memory semantics like that of Seax (defined in Figure 7), nor even a good notion of what is observable. Some related work exists in a message-passing setting [23] which may be of use here, but does not immediately generalize. Likewise, there is some prior work on linear logical relations, and we have made use of that in proving termination for a simplified version of Seax [11], but have not yet investigated how this approach may lead to a notion of equivalence. While it seems likely that a modularity or isolation result will not depend on specific notions of equivalence, having them would provide us with more examples to work with. The most obvious example of a mixed-semantics language that would be useful for investigating isolation would be a two-mode language where one mode uses the message-passing semantics of Section 4, and the other uses the shared-memory semantics of Section 5. Investigating the nature of modularity in this specific case would then serve as guidance for what a more general result could look like.

7 Potential Extensions/Future Work

Several interesting questions remain that are related to the proposed work or to the space of adjoint logic-based programming in general, but do not appear to be on the critical path towards a modularity result, and so are left as future work or potential extensions to the thesis should they turn out to be more critical than they appear. We describe several of these below.

The sequential semantics in Section 5.1 seem to work well, but there are other choices we could have made when developing a sequential semantics. For instance, rather than using atomic writes in order to build a sequential cut, we could work instead with a general synchronizing read, perhaps of the form $\mathsf{case}\ x\ (y \Rightarrow Q)$, which simply blocks until x can be read from, and then continues as Q, which can reference the address x as y. This construct would also allow us to remove the added downshift in sequential cuts, which serves primarily to add an extra synchronization point. There may be other examples of simple primitives that are sufficient to let us model sequentiality, and it is unclear which ones are more natural or elegant.

The garbage collection theorem for the message-passing semantics in Section 5 works only for positive types because of a limited notion of observation. With a better notion of what is observable, perhaps that of Kavanagh [23], it should be possible to extend this theorem to negative types.

The languages that we examine in Sections 4 and 5 are, in some sense, very low-level. We need to be very precise about how communication takes place — what messages are sent, when memory is allocated and written to, and so on. While this is quite convenient for reasoning about programs and proving theorems about them, for actual programming, syntactic sugar to allow higher-level programming would be very useful. It is unclear what a good higher-level language based on these concepts would look like. Developing such a language, along with a compilation of that language into one of these more low-level systems, would be an interesting line of work. Similarly, on the other end, an actual implementation of Seax would also be interesting to develop.

While we focus quite heavily on operational semantics in the existing and proposed work, a categorical semantics for this form of adjoint logic would be interesting. In Benton's work on LNL [5], he presents some work on a categorical semantics for LNL, but this does not appear to generalize smoothly to adjoint logic. In particular, Benton's goal seems to have been to provide a semantics for LNL entirely within a single category. Licata et al. provide a very general categorical semantics for their form of adjoint logic [27], which is itself more general than our presentation here, but loses some properties, such as independence as a general principle, which we find useful. As such, there may be a simpler, if more restrictive categorical semantics of adjoint logic which is sufficient for our formulation.

Many programs that we can write in either of the languages in Sections 4 and 5 will work at more than one mode, but in order to work at multiple modes, we need to rewrite the program. It would be interesting and practically useful for programming to investigate mode polymorphism. Of course, polymorphism leads naturally to the idea of limited polymorphism, perhaps via some form of 'sub-moding', where we could write a process that works for all modes k above (or below) some fixed mode m, or even by making reference to $\sigma(k)$. Additionally, after polymorphism, it is natural to consider the dual idea of abstract or existential types, here abstracted over a mode rather than a type.

Language Extensions One broad category of future work deals with adding on to the language we have built up in a non-uniform way. In the existing and proposed work, we try to keep our languages as symmetric as possible, with all modes being treated the same except when the preorder or the structural properties at each mode disallow it. We see a small example of non-uniformity when working with sequentiality (Section 5.1), where we discuss restricting some modes to only allow sequential cuts. There are a variety of other types of constructs that we might be able to add to the language (in a restricted fashion, living only at some subset of the modes) and work with cleanly, particularly given a modularity result as proposed.

A simple example of this involves working with inductive and/or coinductive types, rather than general recursive types. While this may prove to be simply a restriction on certain modes, much like the restriction to only allow sequential cuts, it still serves as an example of non-uniformity, where different modes have different programs.

The work of Balzer et al. on sharing [2, 3] makes use of a similar mode and type system to our own, but different semantics, which, rather than being solely based on proof reductions, interleave proof reduction sequences with proof construction and proof deconstruction to model acquisition and release of shared resources. It would be interesting to use this semantics at some unrestricted modes, and our usual semantics at others, allowing us to distinguish between shared resources and replicated resources.

Work by Paykin and Zdancewic [33] extends the ideas of Benton's LNL [5] to classical logic. This raises the question of whether we could make some modes behave classically and take advantage of this for programming. In particular, classical linear logic has seen some use in session-typed programming (e.g. Wadler's CP calculus [46]), and it may be possible to use semantics based on this at a linear mode that allows classical reasoning. As with the prior examples, this requires a sufficient understanding of the isolation properties of adjoint logic to ensure that the classical reasoning does not "leak" to other modes.

References

[1] Andreoli, J.-M. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation 2*, 3 (1992), 197–347.

[2] Balzer, S., and Pfenning, F. Manifest sharing with session types. In *International Conference on Functional Programming (ICFP)* (Sept. 2017), ACM, pp. 37:1–37:29. Extended version available as Technical Report CMU-CS-17-106R, June 2017.

[3] Balzer, S., Toninho, B., and Pfenning, F. Manifest deadlock-freedom for shared session types. In *28th European Symposium on Programming (ESOP 2019)* (Prague, Czech Republic, Apr. 2019), L. Caires, Ed., Springer LNCS 11423, pp. 611–639.

[4] Barber, A. Dual intuitionistic linear logic. Tech. Rep. ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, Sept. 1996.

[5] Benton, N. A mixed linear and non-linear logic: Proofs, terms and models. In *Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94)* (Kazimierz, Poland, Sept. 1994), L. Pacholski and J. Tiuryn, Eds., Springer LNCS 933, pp. 121–135. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

[6] Blelloch, G. E., and Reid-Miller, M. Pipeling with futures. *Theory of Computing Systems 32* (1999), 213–239.

[7] Boudol, G. Asynchrony and the π-calculus. Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[8] Caires, L., and Pfenning, F. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)* (Paris, France, Aug. 2010), Springer LNCS 6269, pp. 222–236.

[9] Cervesato, I., Pfenning, F., Walker, D., and Watkins, K. A concurrent logical framework II: Examples and applications. Tech. Rep. CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[10] Cervesato, I., and Scedrov, A. Relating state-based and process-based concurrency through linear logic. *Information and Computation 207*, 10 (Oct. 2009), 1044–1077.

[11] DeYoung, H., Pfenning, F., and Pruiksma, K. Semi-axiomatic sequent calculus. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)* (2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[12] Fairtlough, M., and Mendler, M. Propositional lax logic. *Information and Computation 137*, 1 (Aug. 1997), 1–33.

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

[14] Gay, S. J., and Hole, M. Subtyping for session types in the π-calculus. *Acta Informatica 42*, 2–3 (2005), 191–225.

[15] Gentzen, G. 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.

[16] Girard, J.-Y. Linear logic. *Theoretical Computer Science 50* (1987), 1–102.

[17] Girard, J.-Y., and Lafont, Y. Linear logic and lazy computation. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development* (Pisa, Italy, Mar. 1987), H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, Eds., vol. 2, Springer-Verlag LNCS 250, pp. 52–66.

[18] Griffith, D. *Polarized Substructural Session Types.* PhD thesis, University of Illinois at Urbana-Champaign, Apr. 2016.

[19] Halstead, R. H. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 501–539.

[20] Honda, K., and Tokoro, M. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)* (Geneva, Switzerland, July 1991), P. America, Ed., Springer-Verlag LNCS 512, pp. 133–147.

[21] Jung, R., Krebbers, R., Birkedal, L., and Dreyer, D. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (2016), pp. 256–269.

[22] KANOVICH, M., KUZNETSOV, S., NIGAM, V., and SCEDROV, A. Subexponentials in non-commutative linear logic. *Mathematical Structures in Computer Science 29*, 8 (2019), 1217–1249.

[23] Kavanagh, R. Substructural observed communication semantics. In *27th International Workshop on Expressiveness in Concurrency (EXPRESS/SOS 2020)* (Aug. 2020), O. Dardha and J. Rot, Eds., EPTCS 322, pp. 69–87.

[24] Larus, J. R. *Restructuring symbolic programs for concurrent execution on multiprocessors.* PhD thesis, University of California at Berkeley, 1989.

[25] Liang, C., and Miller, D. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science 410*, 46 (Nov. 2009), 4747–4768.

[26] Licata, D. R., and Shulman, M. Adjoint logic with a 2-category of modes. In *International Symposium on Logical Foundations of Computer Science (LFCS)* (Jan. 2016), Springer LNCS 9537, pp. 219–235.

[27] Licata, D. R., Shulman, M., and Riley, M. A fibrational framework for substructural and modal logics. In *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17)* (Oxford, UK, Sept. 2017), D. Miller, Ed., LIPIcs, pp. 25:1–25:22.

[28] Mostrous, D., and Vasconcelos, V. Affine sessions. In *16th International Conference on Coordination Models and Languages* (Berlin, Germany, June 2014), E. Kühn and R. Pugliese, Eds., Springer LNCS 8459, pp. 115–130.

[29] Negri, S., and von Plato, J. *Structural Proof Theory.* Cambridge University Press, 2001.

[30] Nigam, V., and Miller, D. Algorithmic specifications in linear logic with subexponentials. In *Proceedings of the 11th International Conference on Principles and Practice of Declarative Programming (PPDP)* (Coimbra, Portugal, sep 2009), ACM, pp. 129–140.

[31] Nigam, V., Pimentel, E., and Reis, G. Specifying proof systems in linear logic with subexponentials. *Electronic Notes in Theoretical Computer Science 269* (2011), 109–123.

[32] Owicki, S., and Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM 19*, 5 (1976), 279–285.

[33] Paykin, J., and Zdancewic, S. A linear/producer/consumer model of classical linear logic. *Mathematical Structures in Computer Science 28*, 5 (2018), 710–735.

[34] Pfenning, F. Intensionality, extensionality, and proof irrelevance in modal type theory. Tech. Rep. CMU-CS-01-116, Department of Computer Science, Carnegie Mellon University, Apr. 2001.

[35] Pfenning, F., and Davies, R. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science 11* (2001), 511–540. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[36] Pfenning, F., and Griffith, D. Polarized substructural session types. In *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)* (London, England, Apr. 2015), A. Pitts, Ed., Springer LNCS 9034, pp. 3–22. Invited talk.

[37] Pruiksma, K., Chargin, W., Pfenning, F., and Reed, J. Adjoint logic. Unpublished manuscript, Apr. 2018.

[38] Pruiksma, K., and Pfenning, F. A message-passing interpretation of adjoint logic. In *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)* (Prague, Czech Republic, Apr. 2019), F. Martins and D. Orchard, Eds., EPTCS 291, pp. 60–79.

[39] Pruiksma, K., and Pfenning, F. Back to futures. *CoRR abs/2002.04607* (Feb. 2020).

[40] Reed, J. A judgmental deconstruction of modal logic. Unpublished manuscript, May 2009.

[41] Simmons, R. J. *Substructural Logical Specifications.* PhD thesis, Carnegie Mellon University, Nov. 2012. Available as Technical Report CMU-CS-12-142.

[42] Simmons, R. J. Structural focalization. *ACM Transactions on Computational Logic 15*, 3 (2014), 21:1–21:33.

[43] Toninho, B. *A Logical Foundation for Session-based Concurrent Computation.* PhD thesis, Carnegie Mellon University and Universidade Nova de Lisboa, May 2015. Available as Technical Report CMU-CS-15-109.

[44] Toninho, B., Caires, L., and Pfenning, F. Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the European Symposium on Programming (ESOP'13)* (Rome, Italy, Mar. 2013), M.Felleisen and P.Gardner, Eds., Springer LNCS 7792, pp. 350–369.

[45] Wadler, P. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), pp. 45–52.

[46] Wadler, P. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming* (Copenhagen, Denmark, Sept. 2012), ICFP 2012, ACM Press, pp. 273–286.