Abstract. We briefly introduce the semi-axiomatic sequent calculus for linear logic whose natural computational interpretation is session-typed asynchronous communication. This natural asynchrony allows us to endow it with a shared-memory semantics that is weakly bisimilar to its more standard message-passing semantics. We then show how to further refine the concurrent shared memory semantics into a sequential one. Exploiting the expressive framework of adjoint logic, we show how to combine instances of message-passing, shared-memory, and sequential languages into a coherent whole. We exemplify this by providing rational reconstructions for SILL and futures, two approaches for introducing concurrency into functional programming languages. As a byproduct we obtain a first complete definition of typed linear futures.

Keywords: Session types · Linear logic · Futures

1 Introduction

The computational interpretation of constructive proofs depends not only on the logic, but on the fine structure of its presentation. For example, intuitionistic logic may be presented in axiomatic form (and we obtain combinatory reduction [13]), or via natural deduction (which corresponds to the λ-calculus [30]), or a sequent calculus with a stoup (and we discern explicit substitutions [25]).

More recently, a correspondence between linear logic in its sequent formulation and the session-typed synchronous π-calculus has been discovered [3, 10, 7, 54] and analyzed. Cut reduction here corresponds to synchronous communication. We can also give an asynchronous message-passing semantics [19] to the same syntax (in the sense that input is blocking but output is not), but this is no longer directly related to cut reduction in the sequent calculus [16]. In this paper we briefly introduce a new style of inference system that combines features from Hilbert’s axiomatic form [26] and Gentzen’s sequent calculus [20] which we call semi-axiomatic sequent calculus [15] (Section 2). Cut reduction in the semi-axiomatic sequent calculus corresponds to asynchronous communication somewhat like the asynchronous π-calculus [6, 28] except that message order must be preserved.

In this paper we show that the apparently small change from the ordinary to the semi-axiomatic sequent calculus has profound and far-reaching consequences.
The first of these is that if we stick to intuitionistic (rather than classical) linear logic, we can easily give a natural (although restricted to write-once memory) shared-memory semantics which is weakly bisimilar with the distributed message-passing semantics (Section 4). In this semantics, channels are reinterpreted as locations in shared memory. Naively, we might expect that sending a message corresponds to writing to memory, while receiving a message reads it. Instead all right rules in the sequent calculus write to memory and all left rules read from memory. This presents the opportunity for a simple sequential semantics consistent with the more typical interpretations of intuitionistic logic, but here the role of memory is made explicit (Section 5). The sequential semantics is a refinement of the shared memory semantics that can be seen as a particularly simple stack-based scheduler. This observation, together with the conceptual tools from adjoint logic \[48, 35, 36, 47\] allow us to reconstruct SILL \[52, 51, 22\], fork/join parallelism \[11\], and futures \[24\], three approaches that introduce concurrency into (functional) programming languages (see Sections 7 and 8 and appendix A.4). As a side effect, we obtain a rigorous treatment of linear futures, already anticipated and used in parallel complexity analysis by Blelloch and Reid-Miller \[5\] without fully developing them.

For simplicity, we focus mainly on purely linear channels and processes. The ideas are robust enough to easily generalize to the case where weakening and contraction are permitted on the logical side, which leads to memory structures that may be referenced zero (weakening) or multiple (contraction) times (Section 9). Both concurrent and sequential semantics generalize, which means we have captured SILL and a typed version of futures in their usual generality: concurrency primitives layered over a fully expressive functional language.

The principal contributions of this paper are:

1. a shared-memory semantics for session-typed programs based on a computational interpretation of linear logic with a direct proof of bisimilarity to the usual message-passing semantics;
2. a sequential refinement of this semantics;
3. generalizations of these constructions from linear logic to adjoint logic, which supports conservative combinations of multiple modes of truth with varying structural properties among weakening and contraction;
4. a logical reconstruction of SILL which combines functional programming with session-typed message-passing concurrency;
5. a logical reconstruction of typed futures (including a rigorous definition of linear futures) within adjoint logic.

2 A Linear Sequent Calculus for Asynchronous Communication

The Curry-Howard interpretation of linear logic \[21, 3, 10, 7, 8, 54\] interprets linear propositions as session types \[27, 29\], proofs in the sequent calculus as processes, and cut reduction as communication. Communication is synchronous in
the sense that sender and receiver proceed to their continuations in one joint step. Asynchronous communication, in the sense that the sender may proceed before the message is received, can also be modeled operationally [19] (departing from the correspondence with linear logic) or logically [16]. The latter interprets a message as a process whose only job it is deliver a message. Nevertheless, to be faithful to the logical foundation, an implementation would have to support the default of synchronous communication.

One can ask if there is a different formulation of linear logic that forces asynchronous communication because cut reduction itself is asynchronous. An instantiation of this idea can be found in Solos [33, 23] which goes to an extreme in the sense that both send and receive of channels are asynchronous operations, although branching on labels received still has to block. Here we present a practical intermediate point where all send actions are asynchronous but receive actions block.

We call the underlying sequent calculus *semi-axiomatic*. It retains the invertible left or right rules for each connective and replaces the noninvertible ones with axioms, so half the rules will be axioms, while the other half will be the familiar sequent calculus rules. They are presented in Figure 1.

\[
\begin{align*}
\frac{\Gamma \vdash A}{\Delta, A \vdash C} & \quad \text{id} \\
\frac{\Delta, A \vdash C}{\Gamma, \Delta \vdash C} & \quad \text{cut} \\
\frac{A_1 \vdash A_1 \oplus A_2}{\oplus R^0} & \\
\frac{\Gamma, A_1 \vdash C \quad \Gamma, A_2 \vdash C}{\Gamma, A_1 \oplus A_2 \vdash C} & \quad \oplus L \\
\frac{A, B \vdash A \otimes B}{\otimes R^0} & \\
\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} & \quad \otimes L \\
\frac{\cdot \vdash 1}{1^R} & \\
\frac{\Gamma \vdash C}{\Gamma, 1 \vdash C} & \quad 1^L \\
\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\& R} & \\
\frac{\Gamma, A_1 \& A_2 \vdash C}{\Gamma, A_1 \& A_2 \vdash C} & \quad \& L^0_1 \\
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} & \quad \Rightarrow R \\
\frac{A, A \Rightarrow B \vdash B}{A, A \Rightarrow B \vdash B} & \quad \Rightarrow L^0 \\
\end{align*}
\]

Fig. 1. Semi-Axiomatic Sequent Calculus, linear fragment

In the presence of cut, it is easy to show that $\Gamma \vdash A$ in the ordinary sequent calculus if and only if $\Gamma \vdash A$ in the semi-axiomatic sequent calculus. In order to state and prove a result that is analogous to cut elimination, we syntactically
classify some proofs as **normal** and then show that for each proof of $\Gamma \vdash A$ there exists one that is normal. Normal forms allow some **analytic** cuts (which preserve the subformula property) and can also be translated compositionally to cut-free proofs in the ordinary sequent calculus. This is analogous to the different notions of normal form for natural deductions [45, 30] and combinatory terms [13], which is where the Curry-Howard isomorphism originated.

In this paper, our interest lies elsewhere, so we do not carry out this proof-theoretic analysis (see [15] for the results on the additive fragment), but we do extract the key steps of cut reduction from the proof of normalization. For this purpose, we label antecedents and succedent with distinct variables so we can better recognize the effects of cut reduction. For internal choice, the reduction selects a branch ($E_2$, in the example) and also substitutes for the antecedent.

For linear implication, we substitute both for antecedent and succedent.

We see that in both of these cases the cut disappears entirely rather than being reduced to smaller cuts as in the ordinary sequent calculus. In the operational interpretation presented in the next section, axioms correspond to messages, cut reduction corresponds to message receipt, and certain cuts correspond to the (asynchronous) sending of messages.

### 3 Process Expressions and a Message-Passing Semantics

We now assign process expressions to deductions in the linear semi-axiomatic sequent calculus and formalize their operational semantics. This represents a special case of the semantics of adjoint logic [47] so we omit some details. The typing judgment for processes has the form

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

where $P$ is a process expression providing along channel $x$ and using channels $x_i$. We have the generic rules of cut and identity.

$$\frac{\begin{array}{l} x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A) \\ \Delta, x : A \vdash Q :: (z : C) \end{array}} {\Delta, \Delta \vdash (x \leftarrow P ; Q) :: (z : C)} \quad \text{cut}$$

$$\frac{y : A \vdash (x \leftarrow y) :: (x : A)} {\text{id}}$$
We present the dynamic semantics in the form of multiset rewriting rules [9]. A configuration is a multiset of semantic objects \( \text{proc}(c, P) \) where \( c \) is a channel and \( P \) is a process providing \( c \). These semantic objects are ephemeral in the sense that if they appear on the left-hand side of a rewriting rule they are consumed and replaced by the right-hand side. Rules can be applied to any subconfiguration, leaving the remainder of the configuration unchanged. All the channels provided in a configuration must be pairwise distinct. By convention, we write the provider of a channel to the left of its client. We then have the following rule for cut, which corresponds to spawning a new process.

\[
\text{proc}(c, x \leftarrow P; Q) \rightarrow \text{proc}(a, [a/x]P), \text{proc}(c, [a/x]Q) \quad (a \text{ fresh})
\]

In the reduction rules we use \( c, d, e \) for channels, which are runtime artifacts, and we continue to use \( x, y, z \) for variables in programs that stand for channels that are created at runtime. In an object \( \text{proc}(c, P) \) the expression \( P \) will refer to channels, but will not have any free variables.

In some sense, the most natural semantics for identity \( d \leftarrow c \) is a global renaming of \( c \) to \( d \) or \( d \) to \( c \). This is difficult to implement, however, so we instead look for a local version of the semantics. As can be seen from the progress and preservation theorems, it is sufficient for \( \text{proc}(d, d \leftarrow c) \) to communicate with the provider of \( c \), and that only when the provider of \( \text{proc}(c, P) \) itself tries to communicate along \( c \). We express this as saying that \( P \) is poised on \( c \). This is also the solution adopted in Concurrent C0 [55].

\[
\text{proc}(c, P), \text{proc}(d, d \leftarrow c) \rightarrow \text{proc}(d, [d/c]P) \quad (P \text{ poised on } c)
\]

We introduce additional process expressions in the remainder of this section. A summary of all the rules can be found in Figures 4 to 6. In the purely linear case presented so far, the only mode is \( m = L \), so the mode subscripts can be ignored and we have no shifts. Also, \( \Gamma_W \) and \( \Gamma_C \) are always empty. For a summary restricted to the linear fragment, please refer to Appendix A.2.

**Internal and External Choice.** In order to support programming more naturally, we generalize internal choice and write \( \oplus \{ \ell : A_\ell \}_{\ell \in L} \) where \( L \) is a non-empty finite set of labels. The binary choice can be recovered as \( A_1 \oplus A_2 \equiv \oplus \{ \pi_1 : A_1, \pi_2 : A_2 \} \). We then obtain the following rules:

\[
\begin{align*}
(i \in L) & \quad y : A_i \vdash x.i(y) \vdash (x : \oplus \{ \ell : A_\ell \}_{\ell \in L}) \quad \oplus^0 R^0 \\
\Gamma, y : A_i \vdash Q_\ell :: (z : C) & \quad (\text{for all } \ell \in L) \\
\Gamma, x : \oplus \{ \ell : A_\ell \}_{\ell \in L} \vdash \text{case } x (\ell(y) \Rightarrow Q_\ell)_{\ell \in L} :: (z : C) & \quad \oplus^L
\end{align*}
\]

The right rule corresponds to a message carrying the label \( i \) and a continuation channel \( y \). The left rule provides a branch for every possible label \( \ell \in L \) and also a bound name \( y \) for the continuation channel. Interaction between these objects is captured in the following computation rule.

\[
\text{proc}(c, c.\pi(d)), \text{proc}(e, \text{case } c (\ell(y) \Rightarrow Q_\ell)_{\ell \in L}) \rightarrow \text{proc}(e, [d/y]Q_i)
\]
This rule represents receipt of the message $i(d)$ along channel $c$. There is no corresponding rule to send such a message, since sending is achieved asynchronously by spawning a process (using cut).

External choice $\& \{\ell : A\ell\}_{\ell \in L}$ behaves dually. The nullary left rule represents a message $x.i(y)$ with label $i$ and continuation channel $y$, while the right rule receives and branches on such a message. These rules can be found in Figure 5 or in Appendix A.2. Computationally, we have

$$\text{proc}(c, \text{case} c ((\ell(y) \Rightarrow Q\ell)_{\ell \in L}), \text{proc}(d, c.i(d)) \rightarrow \text{proc}(d, [d/y]Q_i))$$

The first process above continues, but now providing the continuation channel $d$ that is received with the message.

**Multiplicative Unit.** Because we use it in our examples, we briefly present the multiplicative unit $1$. The right rule corresponds to a message $\langle \rangle$ that also closes the channel after it is received, so there is no continuation. The left rule just waits for this message and then proceeds.

$$\frac{}{\cdot \vdash x.(\langle \rangle) :: (x : 1)} \quad 1R^0$$

$$\frac{\Gamma \vdash Q :: (z : C)}{\Gamma, x : 1 \vdash \text{case} x (\langle \rangle \Rightarrow Q) :: (z : C)} \quad 1L$$

Even though the usual $1R$ rule already has no premises, we call it here $1R^0$ to emphasize the principle that in the semi-axiomatic sequent calculus, all noninvertible rules become axioms. Operationally, the unit is straightforward.

$$\text{proc}(c, c.(\langle \rangle)), \text{proc}(e, \text{case} c (\langle \rangle \Rightarrow Q)) \rightarrow \text{proc}(e, Q)$$

**Linear Implication and Multiplicative Conjunction.** As is true generally for session types, the provider of a linear implication $c : A \rightarrow B$ receives a channel $e : A$. But because communication is asynchronous, it also must receive a continuation channel $d : B$, so the message has the form $c.(e, d)$. The provider has a corresponding $\text{case} c ((w, y) \Rightarrow P)$. The actions for the multiplicative conjunction $c : A \otimes B$ are exactly the same, just reversing the role of provider and client. The typing rules are in Figure 6 and the computation rules are included in the ones presented later in this section and in Appendix A.1.

**Recursive Types and Processes.** So far, we have strictly adhered to the logical foundation provided by the Curry-Howard interpretation of linear logic in its semi-axiomatic formulation. In a programming language, we also need recursive types and recursively defined processes. As is customary in session types, we use equirecursive types, collected in a signature $\Sigma$ in which we also collect recursive process definitions and their types. For each type definition $t = A$, the type $A$ must be contractive so we can treat types equirecursively with a straightforward coinductive definition and efficient algorithm for type equality [18].

A definition of the process $p$ has the form $x \leftarrow p \leftarrow y_1, \ldots, y_n = P$, where the type of $p$ is given as $y_1 : B_1, \ldots, y_n : B_n \vdash p :: (x : A)$.

Signature $\Sigma ::= \cdot | \Sigma, t = A | \Sigma, \{y_m : B_m \vdash p :: (x_k : A_k)\} | \Sigma, x \leftarrow p \leftarrow y_m = P$
For valid signatures we require that each declaration \( y_m : B_m \vdash p : (x_k : A_k) \) has a corresponding definition \( x_k \leftarrow p \leftarrow y_m = P \) with \( y_m : B_m \vdash P : (x_k : A_k) \). This means that all type and process definitions can be mutually recursive.

A call is then typed by

\[
\frac{y_1 : B_1, \ldots, y_n : B_n \vdash p : (x : A) \in \Sigma}{w_1 : B_1, \ldots, w_n : B_n \vdash z \leftarrow p \leftarrow w_1, \ldots, w_n : (z : A)} \quad \text{call}
\]

Operationally, we simply unfold the definition so this rule does not require communication.

\[
\text{proc}(c, c \leftarrow d) \mapsto \text{proc}(c, [c/x, d/y]P) \quad \text{for } x \leftarrow y = P \in \Sigma \quad \text{(call)}
\]

In the remainder of this paper we assume that we have a fixed valid signature \( \Sigma \), so we annotate neither the typing judgment nor the computation rules with an explicit signature.

**Example: Binary Numbers.** As a first simple example we consider binary numbers, defined as a type \( \text{bin} \).

\[
\text{bin} = \oplus \{ \text{b0} : \text{bin}, \text{b1} : \text{bin}, \text{e} : \text{1} \}
\]

For example, the number 6 = (110)\(_2\) would be represented by a sequence of labels \( \text{e, b1, b1, b0} \). The first bit to be received would be \( \text{b0} \). It arrives along a channel, say \( c_0 \), and carries with it the continuation channel \( c_1 \) for the remaining bits. Writing out the whole sequence as a configuration we have

\[
\begin{align*}
\text{proc}(c_4, c_4.), & \quad \text{proc}(c_3, c_3.e(c_4)), \quad \text{proc}(c_2, c_2.b1(c_3)), \\
& \quad \text{proc}(c_1, c_1.b1(c_2)), \quad \text{proc}(c_0, c_0.b0(c_1))
\end{align*}
\]

**Example: Computing with Binary Numbers.** We implement a process \( \text{succ} \) which receives the bits of a binary number \( n \) along a channel \( y \) and produces the bits for the binary number \( n + 1 \) along \( x \). For a spawn/cut in the examples, we consider a line break like a semicolon. Also, in the examples it is helpful to have a reverse cut, where the order of the premises is reversed.

\[
\frac{\Delta, x : A \vdash Q : (z : C) \quad \Gamma \vdash P : (x : A)}{\Gamma, \Delta \vdash (x \leftarrow Q ; P) : (z : C)} \quad \text{cut}^R
\]

Operationally, \( \text{cut} \) and \( \text{cut}^R \) behave identically. Using the channel names, it is easy to disambiguate which syntactic form of cut is used, so we will point out uses of reverse cut only in this first example. As a general convention in the example processes, we write the continuation of a channel \( x \) as \( x' \). The code for the successor process is in Figure 2.

To implement \( \text{plus2} \) we can just compose \( \text{succ} \) with itself. In the message-passing semantics, the two successor processes form a concurrently executing pipeline. This definition is also in Figure 2. We have taken a small syntactic liberty in that the second line of \( \text{plus2} \) should technically be \( y \leftarrow (y \leftarrow \text{succ} \leftarrow z) \).

We will continue to use the same abbreviation in the remaining examples.
\[\text{bin} = \oplus\{\text{b0 : bin}, \text{b1 : bin}, \text{e : 1}\}\]

\[\text{y : bin} \vdash \text{succ :: (x : bin)}\]

\[
x \leftarrow \text{succ} \leftarrow y = \begin{cases} \
\text{case y (b0(y') ⇒ x' ← x.b1(x'))} & \% \text{send b1 along x (cut\textsuperscript{B})} \\
\text{b1(y') ⇒ x' ← x.b0(x')} & \% \text{forward the remaining bits} \\
\text{e(y') ⇒ x' ← x.e(x'')} & \% \text{send b1 along x (cut\textsuperscript{A})} \\
\text{e(x'')} & \% \text{followed by e (cut\textsuperscript{B})}
\end{cases}
\]

\[\text{x'}' \leftarrow y'\]

\[
\begin{array}{l}
\text{cut R} \\
\% \text{carry, as a recursive call}
\end{array}
\]

\[
x'' \leftarrow x'.e(x'') \\
\% \text{terminate, by forward at type 1}
\]

\[\text{z : bin} \vdash \text{plus2 :: (x : bin)}\]

\[
z \leftarrow \text{plus2} \leftarrow x = \begin{cases} \
\text{y ← succ ← z} & \% \text{spawn first successor in pipeline} \\
\text{x ← succ ← y} & \% \text{continue as second successor}
\end{cases}
\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example2.png}
\caption{Successor and plus2 processes on binary numbers}
\end{figure}

**Example: A Binary Counter.** As a second example, now using external choice, we implement a binary counter for which the client has two choices: it can either increment the counter (message \text{inc}) or it can request the value in the form of a binary number from the preceding example (message \text{val}).

\[\text{ctr} = \&\{\text{inc : ctr, val : bin}\}\]

We implement the counter as a chain of processes, each carrying one bit of information, and a final process representing the end of the chain, see Figure 3. An increment message travels through the chain as far as necessary to account for all carries, and the value message travels all the way to the end, converting each process \text{bit0}, \text{bit1}, and \text{end} to its corresponding message \text{b0}, \text{b1}, and \text{e}.

**Summary.** The semantics presented in this section is a message-passing semantics and therefore suitable for distributed computation. It is easy to identify the messages and the threads of control that define the state of each process. A message has the form \(c.V\) where \(c\) is a channel and \(V\) is a value carried by the message. The values are all small and either of the form \(i(d)\) (where \(i\) is a label and \(d\) a continuation channel), \(⟨e,d⟩\) where \(e\) is a channel and \(d\) the continuation channel, and \(⟨⟩\) representing the final message on a channel with no continuation. They are uniformly received by a process of the form \(\text{case } c K\) where \(K\) is a continuation expression. This is either \(ℓ(y) ⇒ P_\ell\) (for internal or external choice), or \(⟨w,y⟩ ⇒ P\) (for linear implication or multiplicative conjunction), or \(⟨⟩ ⇒ P\) (for multiplicative unit).

A central operation (in both this message-passing semantics and the shared memory semantics of section 4) is to pass a value \(V\) to a continuation \(K\), written
\[ \text{ctr} = \&\{ \text{inc} : \text{ctr}, \text{val} : \text{bin} \} \]

\[ y : \text{ctr} \vdash \text{bit0} : \text{ctr} \]
\[ y : \text{ctr} \vdash \text{bit1} : \text{ctr} \]
\[ \vdash \text{end} : \text{ctr} \]

\[
\begin{align*}
x &\leftarrow \text{bit0} \leftarrow y = \\
\text{case } x \ (\text{inc} (x')) &\Rightarrow x' \leftarrow \text{bit1} \leftarrow y \quad \% \text{continue as bit1} \\
| \text{val} (x') &\Rightarrow x'' \leftarrow x'.\text{b0}(x'') \quad \% \text{respond with b0 along } x' \\
\quad y' &\leftarrow y.\text{val}(y') \quad \% \text{convert remaining bits to binary} \\
\quad x'' &\leftarrow y' \quad \% \text{and forward}
\end{align*}
\]

\[
\begin{align*}
x &\leftarrow \text{bit1} \leftarrow y = \\
\text{case } x \ (\text{inc} (x')) &\Rightarrow y' \leftarrow y.\text{inc}(y') \quad \% \text{carry bit as message inc along } y \\
\quad x' &\leftarrow \text{bit0} \leftarrow y \quad \% \text{continue as bit0} \\
| \text{val} (x') &\Rightarrow x'' \leftarrow x'.\text{b1}(x'') \quad \% \text{respond with b1 along } x' \\
\quad y' &\leftarrow y.\text{val}(y') \quad \% \text{convert remaining bits to binary} \\
\quad x' &\leftarrow y' \quad \% \text{and forward}
\end{align*}
\]

\[
\begin{align*}
x &\leftarrow \text{end} = \\
\text{case } x \ (\text{inc} (x')) &\Rightarrow y \leftarrow \text{end} \quad \% \text{spawn new end process} \\
\quad x' &\leftarrow \text{bit0} \leftarrow y \quad \% \text{continue as bit1} \\
| \text{val} (x') &\Rightarrow x'' \leftarrow x'.\text{e}(x'') \quad \% \text{respond with e as empty sequence} \\
\quad x'' &\leftarrow \omega (\langle \rangle) \quad \% \text{close channel } x'' \text{ by sending } \langle \rangle
\end{align*}
\]

**Fig. 3.** A binary counter

As \( V \circ K \) and defined by

\[
\begin{align*}
i (d) \circ (\ell (y) \Rightarrow P_i)_{i \in L} (i (d)) &\equiv[d/y]P_i \quad (\oplus, \&) \\
(e, c) \circ ((w, y) \Rightarrow P) &\equiv[e/w, c/y]P \quad (\ominus, \otimes) \\
(\emptyset) \circ (()) \Rightarrow P &\equiv P \quad (1)
\end{align*}
\]

With this notation we can summarize the transition rules as follows:

\[
\begin{align*}
\text{proc}(c, x &\leftarrow P ; Q) \mapsto \text{proc}(a, [a/x]P), \text{proc}(c, [a/x]Q) \quad (a \text{ fresh}) \quad \text{cut/spawn} \\
\text{proc}(c, P), \text{proc}(d, d &\leftarrow c) \mapsto \text{proc}(d, [d/c]P) \quad (P \text{ poised on } c) \quad \text{id/forward} \\
\text{proc}(c, c &\leftarrow p \leftarrow d) \mapsto \text{proc}(c, [c/x, d/y]P) \quad \text{for } x &\leftarrow p \leftarrow y = P \in \Sigma \text{ call}
\end{align*}
\]

\[
\begin{align*}
\text{proc}(c, c.V), \text{proc}(e, \text{case } c.K) &\mapsto \text{proc}(e, V \circ K) \quad (\oplus, \otimes, 1) \\
\text{proc}(c, \text{case } c.K), \text{proc}(d, V) &\mapsto \text{proc}(d, V \circ K) \quad (\&, \rightarrow)
\end{align*}
\]

**Process Configurations.** A process configuration \( \mathcal{C} \) is either a single process \( \text{proc}(c, P) \) providing along \( c \), or the join \( \mathcal{C}_1, \mathcal{C}_2 \) of two configurations. We think of the join as an associative and commutative operator and impose the condition that all channels \( c \) provided in a configuration are distinct. Configurations use some channels and provide others, so we type them with the judgment
$\Gamma \vdash C :: \Delta$ defined by the following rules.

\[
\begin{align*}
\Gamma \vdash P :: (c : A) & \quad \Delta, \Gamma \vdash \text{proc}(c, P) :: (\Delta, c : A) \\
\Delta \vdash (\cdot) :: \Delta & \quad \Gamma \vdash C_1 :: \Delta_1 \quad \Delta_1 \vdash C_2 :: \Delta_2
\end{align*}
\]

In the first rule, $\Delta$ contains channels not referenced by $P$ which are therefore still available to further clients to the right of $\text{proc}(c, P)$ and we can read the rule for the empty configuration similarly. We then have our first theorems: preservation and progress. The proofs follow standard patterns in the literature [47] and are therefore omitted here. Their origins lie in the correctness of the presented cut reductions in the semi-axiomatic sequent calculus.

**Theorem 1 (Type Preservation [47]).** If $\Gamma \vdash C :: \Delta$ and $C \xrightarrow{} C'$ then $\Gamma \vdash C' :: \Delta$.

A process is poised on channel $c$ if it is trying to communicate along $c$, that is, a process is poised on $c$ if it has the form $\text{proc}(\cdot, c.V)$ (send on $c$), $\text{proc}(\cdot, \text{case } c K)$ (receive on $c$), or $\text{proc}(d, d \leftarrow c)$ (forward between $c$ and $d$).

**Theorem 2 (Progress [47]).** If $\cdot \vdash C :: \Delta$ then either

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

(ii) every $\text{proc}(c, P) \in C$ is poised on $c$.

### 4 A Shared Memory Semantics

The semantics presented in the previous section is a message-passing semantics and therefore suitable for distributed computation. It is easy to identify the messages and the threads of control that define the state of each process. A message has the form $c.V$ and is encapsulated in processes of the form $\text{proc}(\cdot, c.V)$. Messages are small: no process expressions or complicated data structures need to be transmitted. As for threads of control, cut/spawn spawns a new process $[a/x]P$ and continues as $[a/x]Q$, id/forward terminates, and in the rules for logical connectives the process receiving the message continues while the message terminates. A process $\text{proc}(\cdot, \text{case } c K)$ represents a thread of control with continuation expression $K$, which proceeds as $V \circ K$ when a value $V$ is received along channel $c$.

We can now implement this message-passing semantics using a distributed model of computation, or we could implement it using shared memory, depending on our application or programming language. In fact, existing implementations of session types (see, for example, [31, 49, 55, 32]) leverage libraries that may support one or the other or even both. Because of the varying levels of abstractions and the complexity of the linear typing invariant, proving the correctness of such an implementation is difficult and we are not aware of any successful efforts in that direction.
### Types

\[
A_m, B_m ::= \oplus_m \{ \ell : A'_m \} \in L \quad \text{internal choice}
\]
\[
A_m \otimes_m B_m \quad \text{multiplicative conjunction}
\]
\[
1_m \quad \text{multiplicative unit}
\]
\[
\&_m \{ \ell : A'_m \} \in L \quad \text{external choice}
\]
\[
A_m \multimap_m B_m \quad \text{linear implication}
\]
\[
\downarrow^r_m A_r \quad \text{down shift, } r \geq m
\]
\[
\uparrow^k_m A_k \quad \text{up shift, } m \geq k
\]
\[
t \quad \text{type variables}
\]

### Processes

\[
P, Q ::= x_m \leftarrow P ; Q \quad \text{cut/spawn}
\]
\[
x_m \leftarrow y_m \quad \text{id/forward}
\]
\[
x_m, V \quad \text{send value } V
\]
\[
\text{case } x_m K \quad \text{receive and decompose a value}
\]
\[
x_k \leftarrow p \leftarrow \overline{y_m} \quad \text{call}
\]

### Values

\[
V ::= i(y_m) \quad \text{label } i \text{ with cont. } y_m (\oplus, \&)
\]
\[
\langle w_m, y_m \rangle \quad \text{channel } w_m \text{ with cont. } y_m (\otimes, \multimap)
\]
\[
\langle \rangle \quad \text{closing message } (1)
\]
\[
\text{shift}(y_k) \quad \text{shift with cont. } y_k (\downarrow, \uparrow)
\]

### Continuations

\[
K ::= (\ell (y_m) \Rightarrow P)_{\ell \in L} \quad \text{branches for labels } \ell \in L \text{ with conts. } y_m (\oplus, \&)
\]
\[
(\langle w_m, y_m \rangle \Rightarrow P) \quad \text{receiving channel } w_m \text{ and cont. } y_m (\otimes, \multimap)
\]
\[
(\langle \rangle \Rightarrow P) \quad \text{waiting on closing message } (1)
\]
\[
(\text{shift}(y_k) \Rightarrow P) \quad \text{receiving cont. channel } y_k (\downarrow, \uparrow)
\]

---

**Fig. 4.** ADJ0, types and process expressions. For the purely linear fragment, \( m = \mathbb{L} \) and there are no shifts.

Our approach to understanding the implementation of session types is to
give a high-level definition of a shared memory semantics. We use the style of
substructural operational semantics (SSOS) [42] and leverage the technique of
destination-passing style [44], which also has shown promise for a high-performance
implementation [50]. A significant advantage of SSOS is the modularity it affords,
which allows us to subsequently integrate the semantics with others (see
Section 6).

A key restriction behind the message-passing semantics is that messages
are small. In particular, they should carry no code in the form of closures—
constructing them and controlling their size is a well-known problem in the
practice of distributed computing in high-level languages (see, for example, [38]).
In the shared memory setting no such restriction is necessary since code can re-
side in shared memory without difficulty. On the other hand, we would like to
explicitly allocate shared memory and ensure that concurrent access is safe.

Here are the central ideas of the shared-memory semantics:

1. Channels are reinterpreted as addresses in shared memory.
2. Cut/spawn is the only way to allocate a new cell in shared memory.
3. Identity/forward will move data between cells.
\[
\Gamma_W, y : A_m \vdash x \leftarrow y :: (x : A_m)
\]

\[
\Gamma_C, \Delta \vdash P :: (x : A_m) \quad \Gamma_C, \Delta', x : A_m \vdash Q :: (z : C_r) \quad \text{cut}
\]

\[
\frac{(i \in L)}{\Gamma, y : A'_m \vdash x.i(y) :: (x : \oplus_{i \in L} \ell : A''_m)} \quad \oplus R^0
\]

\[
\frac{\Gamma, x : \oplus_{i \in L} \ell : A''_m \vdash \text{case } x.(\ell(y) \Rightarrow Q_{\ell})_{\ell \in L} :: (z : C_r)}{(\text{all } \ell \in L)} \quad \oplus L_\alpha
\]

\[
\frac{\Gamma \vdash P_{\ell} :: (y : A'_m) \quad (\text{for all } \ell \in L)}{\Gamma \vdash \text{case } x.(\ell(y) \Rightarrow P_{\ell})_{\ell \in L} :: (x : \&_{\ell \in L} \ell : A''_m) \quad \& R}
\]

\[
\frac{(i \in L)}{\Gamma, x : \&_{\ell \in L} \ell : A''_m \vdash x.i(y) :: (y : A'_m)} \quad \& L_0
\]

\[
\frac{\Gamma, (x : 1_m)^a \vdash P :: (z : C_r)}{\Gamma, x : 1_m \vdash \text{case } x.(\text{)} \Rightarrow P :: (z : C_r) \quad 1 L_\alpha}
\]

\[
\frac{y_m : B_m \vdash p :: (x_k : A_k) \in \Sigma}{\text{call}}
\]

**Fig. 5.** ADJ0 (\(\oplus, \&, 1\)). \(\alpha \in \{0, 1\}\) with \(\alpha = 1\) permitted only if \(C \in \sigma(m)\).

\[
\frac{\Gamma, w : A_m \vdash P :: (y : B_m)}{\Gamma \vdash \text{case } x.(w, y) \Rightarrow P :: (x : A_m \rightarrow_m B_m) \quad \rightarrow R}
\]

\[
\frac{\Gamma, w : A_m, x : A_m \rightarrow_m B_m \vdash x.(w, y) :: (y : B_m)}{\Gamma \vdash \text{case } x.(w, y) \Rightarrow P :: (x : A_m \rightarrow_m B_m) \quad \rightarrow L_0}
\]

\[
\frac{\Gamma_W, w : A_m, y : B_m \vdash x.(w, y) :: (x : A_m \&_m B_m)}{\Gamma \vdash \text{case } x.(w, y) \Rightarrow P :: (x : A_m \&_m B_m) \quad \oplus R^0}
\]

\[
\frac{\Gamma, (x : A_m \&_m B_m)^a, w : A_m, y : B_m \vdash P :: (z : C_r)}{\Gamma, x : A_m \&_m B_m \vdash \text{case } x.(w, y) \Rightarrow P :: (z : C_r) \quad \oplus L_\alpha}
\]

**Fig. 6.** ADJ0 (\(\rightarrow, \&\)). \(\alpha \in \{0, 1\}\) with \(\alpha = 1\) permitted only if \(C \in \sigma(m)\).
4. A process \texttt{proc}(c, P) that provides \texttt{c} will write to location \texttt{c} and then terminate.

5. A process \texttt{proc}(d, Q) that is poised on a channel \texttt{c} \neq \texttt{d} will read from location \texttt{c} (once it is available) and then continue.

In addition, memory cells that have been read can be deallocated due to linear typing. The counterintuitive part of this interpretation is that a process providing \texttt{c}: \texttt{A \& B} does not read a value from shared memory. Instead, it writes a continuation to memory and terminates. Conversely, a client of such a channel does not write a value to shared memory. Instead, it continues by jumping to the continuation. A memory cell may therefore contain either a value \texttt{V} or a continuation \texttt{K}, a class of expressions we denote by \texttt{W}.

**Cell contents**

\[ W ::= V | K \]

We formalize these principles using the following semantic objects:

1. \texttt{thread}(c, P): thread \texttt{P} with destination \texttt{c}
2. \texttt{cell}(c, _): cell \texttt{c} that has been allocated, but not yet written
3. \texttt{cell}(c, W): cell \texttt{c} containing \texttt{W}

We maintain the invariant that in a configuration either \texttt{thread}(c, P) appears together with \texttt{cell}(c, _), or we have just \texttt{cell}(c, W).

At a lower level of abstraction the continuation would presumably be a closure, pairing a code pointer with an environment assigning addresses to its free channel variables. Unlike messages, values do not carry their destination so we type cell contents with the straightforward judgment \(\Gamma \vdash W : A\) whose rules we elide. The typing rules for configurations can be found in Section 9, where the context \(\Gamma_C\) (for antecedents allowing contraction) are always empty for now. It is convenient to type a thread together with its pre-allocated destination. The rules for the operational semantics just formalize these intuitions.

\[
\begin{align*}
\text{thread}(c, x \leftarrow P ; Q) &\mapsto \text{thread}(a, [a/x]P), \text{cell}(a, _), \text{thread}(c, [a/x]Q) \text{ (a fresh) cut/spawn} \\
\text{cell}(c, _), \text{thread}(d, d \leftarrow c), \text{cell}(d, _) &\mapsto \text{cell}(d, W) \text{ id/move} \\
\text{thread}(c, c \leftarrow p \leftarrow \overline{d}) &\mapsto \text{thread}(c, [c/x, \overline{d/y}]P) \text{ for } x \leftarrow p \leftarrow \overline{y} = P \in \Sigma \text{ call} \\
\text{thread}(c, c.V), \text{cell}(c, _) &\mapsto \text{cell}(c, V) \text{ (} \oplus R^0, \odot R^0, \mathbf{1} R^0 \text{)} \\
\text{cell}(c, V), \text{thread}(c, \text{case } c K) &\mapsto \text{thread}(c, V \circ K) \text{ (} \oplus L, \odot L, \mathbf{1} L \text{)} \\
\text{thread}(c, \text{case } c K), \text{cell}(c, _) &\mapsto \text{cell}(c, K) \text{ (} \rightarrow R, &R \text{)} \\
\text{cell}(c, K), \text{thread}(d, c.V) &\mapsto \text{thread}(d, V \circ K) \text{ (} \odot L^0, \& L^0 \text{)} \\
\end{align*}
\]

Note in particular how the last rule reverses the earlier role of messages and processes. The thread computing with destination \texttt{d} continues execution by jumping to the stored continuation after substitution.

This semantics satisfies the expected variants of preservation and progress. Preservation does not change at all.

**Theorem 3 (Type Preservation).** If \(\Gamma \vdash C :: \Delta\) and \(C \mapsto C'\) then \(\Gamma \vdash C' :: \Delta\).
Progress changes in that a configuration that cannot take a step must have filled in all of its destination cells.

**Theorem 4 (Progress).** If \( \vdash C \vdash \Delta \) then either

(i) \( C \xrightarrow{} C' \) for some \( C' \), or

(ii) for every channel \( c : A \in \Delta \) there is an object \( \text{cell}(c, W) \in C \).

We can now establish a weak bisimulation with the earlier semantics. We define a relation \( R \) between configurations in the message-passing and shared-memory semantics.

\[
\begin{align*}
\text{proc}(c, x \leftarrow P ; Q) & \Rightarrow R \text{ thread}(c, x \leftarrow P ; Q), \text{cell}(c, \_)
\text{proc}(c, c \leftarrow d) & \Rightarrow R \text{ thread}(c, c \leftarrow d), \text{cell}(c, \_)
\text{proc}(c, c \leftarrow p \leftarrow d) & \Rightarrow R \text{ thread}(c, c \leftarrow p \leftarrow d), \text{cell}(c, \_)
\text{proc}(c, c; V) & \Rightarrow R \text{ thread}(c, c.V), \text{cell}(c, \_)
\text{proc}(e, \text{case } c K) & \Rightarrow R \text{ thread}(e, \text{case } c K), \text{cell}(e, \_)
\text{proc}(d, c; V) & \Rightarrow R \text{ thread}(d, c.V), \text{cell}(d, \_)
\text{proc}(c, \text{case } c K) & \Rightarrow R \text{ thread}(c, \text{case } c K), \text{cell}(c, \_)
\text{proc}(c, \text{case } c K) & \Rightarrow R \text{ cell}(c, K)
\end{align*}
\]

We extend this relation to whole configurations compositionally, recalling that the join operation for configurations is associative and commutative with unit (\( \cdot \)).

\[
\begin{align*}
(\cdot) \Rightarrow R (\cdot)
\end{align*}
\]

\[
\begin{array}{c}
\varepsilon \\vdash \varepsilon \vdash \Delta
\end{array}
\]
\[
\begin{array}{c}
C_1 \Rightarrow R \Delta_1
C_2 \Rightarrow R \Delta_2
\end{array}
\]

It is easy to see that \( R \) is a weak bisimulation. In the case of cut, we have to take advantage of the stipulation that new names may be chosen arbitrarily as long as they are fresh.

**Theorem 5 (Weak Bisimulation).** The relation \( R \) is a weak bisimulation between the message-passing and the shared-memory semantics.

The threads of control in the two semantic specifications are different, however. As we can see from the bisimulation itself, some messages \( \text{proc}(\_ , c.V) \) and \( \text{proc}(\_ , \text{case } c K) \) corresponds to threads and others to memory cells. We return to the examples to illustrate the differences.

**Example Revisited: Computing with Binary Numbers, Figure 2.** Recall that a \( \text{succ} \) process was a transducer from sequences of bits to sequences bits, all of those being messages. Under the shared memory semantics, \( \text{succ} \) is still the only process, and it reads the elements of the input sequence from memory and writes the elements of the output sequence to memory. In the \( \text{plus2} \) example, under both interpretations there are two \( \text{succ} \) processes in a pipeline operating concurrently, either on messages or on memory content. We see from this that the two interpretations are quite close for \( \text{positive} \) types \( (\oplus, \otimes, 1) \).
Example Revisited: A Binary Counter, Figure 3. In the message passing semantics for the binary counter we have a sequence of processes, one for each bit and one to mark the end. An increment message \texttt{inc} is passed through this sequence until there are no further carries to be processed. In the shared memory semantics each bit is a continuation expression stored in memory. An increment is actually a process that executes such a continuation, writes a new continuation to memory and then reads and executes the next continuation (if necessitated by a carry) or terminates. When receiving a value message, bits \texttt{b0} and \texttt{b1} are written to memory as each continuation \texttt{bit0} and \texttt{bit1} is reached and executed.

This alternative operational interpretation is greatly aided by our insistence on an intuitionistic sequent calculus. It is precisely the asymmetry between multiple antecedents and a single conclusion that supports the shared memory interpretation, together with the idea to take the semi-axiomatic sequent calculus as our formal basis.

5 A Sequential Semantics

Once we have introduced threads communicating through shared memory in the particular way we have, it is only a small step to a sequential semantics. The key idea is that for cut/spawn \texttt{x} ← \texttt{P} ; \texttt{Q} we execute \texttt{P} with destination \texttt{x} and wait until \texttt{P} has terminated (and therefore deposited a value in the location denoted by \texttt{x}). We then proceed with the execution of \texttt{Q}. This would not have been possible in the message-passing semantics because processes pause and resume as messages are exchanged between them. We choose here a call-by-value sequential semantics because futures (see Section 8) have first been described in this context. Of course, in the language so far we can code sequential computation by performing additional synchronization before proceeding with the continuation of a cut. As with the coding of asynchronous communication under the synchronous semantics, this would present at best a tortuous path towards an understanding how to combine sequential and concurrent modes of computation since a type by itself would remain agnostic to its intended interpretation.

The idea of the sequential semantics is to split the thread \texttt{(c, P)} objects into two: \texttt{eval(c, P)} which evaluates \texttt{P} with destination \texttt{c} and \texttt{cont(c, d, Q)} which waits for the evaluation of \texttt{c} to complete before continuing with the evaluation of \texttt{Q} with destination \texttt{d}. In addition, we have a special case of the \texttt{cell(c, W)} object, \texttt{retn(c, W)}, which means that \texttt{W} is returned to destination \texttt{c}, but has not yet been committed to storage. The configuration then always has one of the following two forms, adhering to our convention to show a provider to the left of its client.

\[
\text{cell(d1, W1), \ldots, eval(c0, P0), cont(c0, c1, P1), cont(c1, c2, P2), \ldots, cell(d1, W1), \ldots, retn(c0, W), cont(c0, c1, P1), cont(c1, c2, P2), \ldots}
\]

The continuation objects are threaded in the form of a stack, while the cells represent a global store as before. However, we no longer have uninitialized cells since a cell will always be written by its provider before it is accessed by the client. When typing configurations we have to express the ordered threading
of continuations. We accomplish this by generalizing the typing judgment to 
\[\Delta, \Gamma \vdash \cdot : \cdot \] where \(\Delta, \kappa\) are either empty (\(\cdot\)) or a single channel \(c\) denoting the top of the continuation stack.

\[
\begin{align*}
\Gamma \vdash P :: (c : A) & \quad \Delta, \Gamma \vdash \text{eval}(c, P) :: (\Delta, c : A) | c \\
\Gamma \vdash W : A & \quad \Delta, \Gamma \vdash \text{id}(a, c, P) :: (\Delta, c : A) | c \\
\Gamma \vdash W : A & \quad \Delta, \Gamma \vdash \text{ret}(c, W) :: (\Delta, c : A) | c \\
\end{align*}
\]

The dynamics consists of the following rules.

- eval\((c, x \leftarrow P : Q)\) \(\Rightarrow\) eval\((a, [a/x]P), \text{cont}(a, c, [a/x]Q)\) (a fresh) cut/spawn
- ret\((a, W), \text{cont}(a, c, Q) \Rightarrow \text{cell}(a, W), \text{eval}(c, Q)\) return/write
- cell\((c, W), \text{eval}(d, d \leftarrow c) \Rightarrow \text{ret}(d, W)\) id/move
- eval\((c, x \leftarrow P : Q)\) \(\Rightarrow\) eval\((e, [c/x][a/y]P)\) for \(x \leftarrow p \leftrightarrow y = P \in \Sigma\) call
- cell\((c, V), \text{eval}(e, \text{case } c K) \Rightarrow \text{eval}(e, V \circ K)\) \((\oplus R^0, \otimes R^0, \odot R^0)\)
- cell\((c, V), \text{eval}(c, V) \Rightarrow \text{ret}(c, V)\) \((\oplus L, \otimes L, 1L)\)
- cell\((c, K), \text{eval}(d, V) \Rightarrow \text{eval}(d, V \circ K)\) \((\ominus R, \& R)\)
- cell\((c, K), \text{eval}(d, V) \Rightarrow \text{eval}(d, V \circ K)\) \((\ominus L^0, \& L^0)\)

We have new but uninteresting versions of preservation and progress.

**Theorem 6 (Type Preservation).** If \(\Gamma \vdash C :: \Delta | \kappa\) and \(C \rightarrow C'\) then 
\[\Gamma | \cdot \vdash C' :: \Delta | \kappa\]

Progress changes in that a configuration that cannot take a step must have filled in all of its destinations either by writing values to memory or returning a value to the continuation on top of the stack.

**Theorem 7 (Progress).** If \(\cdot \vdash C :: \Delta | \kappa\) then either

(i) \(C \rightarrow C'\) for some \(C'\), or
(ii) we have
(a) for every channel \(c : A \in \Delta\) with \(c \neq \kappa\) there is a cell\((c, W)\) \(\in C\), and
(b) if \(\kappa = c_0\) then ret\((c_0, W_0)\) \(\in C\) for some \(W_0\).

The sequential semantics is no longer bisimilar to the concurrent one, but it represents a particular schedule. A simple way of stating this is that the concurrent semantics simulates the sequential one. More might be said by developing a notation of observational equivalence [41], but this is beyond the scope of this paper.

\[
\begin{align*}
\text{thread}(c, P), \text{cell}(c, \_), S \text{ eval}(c, P) \\
\text{thread}(c, P), \text{cell}(c, \_), S \text{ cont}(d, c, P) \\
\text{cell}(c, W), S \text{ cell}(c, W) \\
\text{cell}(c, W), S \text{ ret}(c, W)
\end{align*}
\]
Theorem 8 (Weak Simulation). The relation $S^{-1}$ is a weak simulation.

Example Revisited: Computing with Binary Numbers, Figure 2. Under the sequential semantics, a single succ process will consume the given input sequence until it encounters the first $b0$ (which requires no carry), building up continuation objects that then construct the output. When the two are composed in $\text{plus2}$, the first succ must finish its computation and write all results to memory before the second succ starts.

Example Revisited: A Binary Counter, Figure 3. In the binary counter, an increment represents the only thread of control. It invokes the sequence of continuations in memory in turn until it encounters a bit0 process, at which point it returns and writes fresh continuations. When multiple increments are executed, each has to finish completely before the next one starts. A value message instead traverses the entire sequence, invoking the stored continuations, and then writes corresponding bits to memory as it returns from each.

6 Adjoint Logic: Combining Languages

At this point we have developed several different versions of an operational semantics for the same source language and explored their relationship. They have different use cases: the message-passing semantics is suitable for distributed computation, the shared-memory semantics for multi-threaded computation, and the sequential semantics for functional programming (linear, for now). One obvious generalization of this is to allow the structural rules of weakening and contraction on the logical side and, correspondingly, persistent cells on the computational side. Another question is how we can combine these languages in a coherent manner, since an application may demand, say, mostly sequential programming with some concurrency.

In this section we review adjoint logic [48, 35, 36, 47], a general framework for combining logics with different structural properties. In the next sections we explore particular combinations to reconstruct the essence of SILL [52, 51, 22] and futures [24] as instances of the general schema.

In adjoint logic, propositions are stratified into distinct layers, each identified by a mode. For each mode $m$ there is a set $\sigma(m) \subseteq \{W, C\}$ of structural properties satisfied by antecedents of mode $m$ in a sequent. Here, $W$ stands for weakening and $C$ for contraction. For simplicity, we always assume exchange is possible. In addition, any instance of adjoint logic specifies a preorder $m \geq k$ between modes, expressing that the proof of a proposition $A_k$ of mode $k$ may depend on assumptions $B_m$. In order for cut elimination to hold, this ordering must be compatible with the structural properties: if $m \geq k$ then $\sigma(m) \supseteq \sigma(k)$. Sequents then have the form $\Gamma \vdash A_k$ where, critically, each antecedent $B_m$ in $\Gamma$ satisfies $m \geq k$. We express this concisely as $\Gamma \geq k$.

A prototypical and inspirational example is Benton’s LNL [4] combining linear and nonlinear intuitionistic logic. There are two modes, a linear one $L$ with
\( \sigma(L) = \{ \} \) and an unrestricted one \( U \) with \( \sigma(U) = \{ W, C \} \). Critically, \( U > L \), so the proof of a linear proposition can depend on unrestricted assumptions, while a proof of an unrestricted proposition can only depend on other unrestricted propositions but not linear ones.

We can go back and forth between the layers using shifts \( \uparrow^m_k \) (up from \( k \) to \( m \) requiring \( m \geq k \)) and \( \downarrow^m_k \) (down from \( r \) to \( m \) requiring \( r \geq m \)). A given pair \( \uparrow^m_k \) and \( \downarrow^m_k \) forms an adjunction, justifying the name adjoint logic.

Propositions \( A_m, B_m \) := \( \oplus_m \{ \ell : A_m^\ell \}_{\ell \in L} \mid A_m \to_m B_m \mid \ldots \mid \uparrow^m_k A_k \mid \downarrow^r_m A_r \)

We also sometimes restrict the available propositions of a given mode. For example, we can recover a formulation of intuitionistic linear logic [21] with the two modes \( L \) and \( U \) from LNL with \( U > L \), but stipulating that \( A_u := \uparrow^u_L A_l \). Then \( !A_k = \downarrow^u_r \uparrow^u_L A_k \).

The formulation of adjoint logic for a fixed set of modes denoting linear, affine, and unrestricted propositions has previously been proposed [43]. More recently, this has been generalized to a uniform message-passing semantics for all of adjoint logic [47] which gives rise to new communication patterns. One example is multicast, where a single message is sent to multiple recipients.

We skip a presentation of the pure sequent calculus [46] and just show its semi-axiomatic form in Figures 5, 6, and 7 for the new shift rules. For now, we require all modes to be linear, but remarkably few changes need to be made to handle the general case (see Section 9). When reading these rules it is important to keep in mind the general presupposition that for a sequent \( \Gamma \vdash A_k \) we have \( \Gamma \geq k \). All other sequents are meaningless and hence disallowed a priori. We therefore always assume the conclusion of the rule satisfies the presupposition and have to add sufficient conditions that it will hold for all premises. In this asynchronous formulation a condition is necessary only for cut: the cut proposition \( A_m \) must satisfy \( m \geq k \) so that the second premise is well-formed. Furthermore, since \( m \) may be strictly greater than \( k \), we must verify that \( \Gamma \geq m \), that is, the proof of \( A_m \) may depend on all of its antecedents.

\[
\begin{align*}
\Gamma, W, y : A_m \vdash x.\text{shift}(y) :: (x : \downarrow^m_k A_m) & \quad \downarrow^0 R \\
\Gamma, x : A_m \vdash Q :: (z : C_r) & \quad \downarrow^L \\
\Gamma, x : \downarrow^m_k A_m \vdash \text{case} \ x (\text{shift}(y) \Rightarrow Q) :: (z :: C_r) & \\
\Gamma \vdash P :: (y : A_k) & \quad \uparrow^R \\
\Gamma \vdash \text{case} \ x (\text{shift}(y) \Rightarrow P) :: (x : \uparrow^m_k A_k) & \\
\Gamma_W, x : \uparrow^m_k A_k \vdash x.\text{shift}(y) :: (y : A_k) & \quad \uparrow^L \uparrow^0
\end{align*}
\]

Fig. 7. ADJ0 (\( \uparrow^\downarrow \)).
6.1 A Shared Memory Semantics

We do not provide here a uniform message-passing semantics suitable for a distributed implementation, which has been provided in prior work [47]. Instead, we explore in the remainder of this paper how to combine different operational rules at different modes. We see that only the four shift rules and cut deal with different modes. We have to reexamine cut in each case, but the shift rules remain the same for a shared memory semantics because their actions are prescribed by their polarity. Specifically, $\downarrow A$ is positive and therefore sends, while $\uparrow A$ is negative and therefore receives. We annotate each channel with its mode because, unlike in the typing rules, this information is not readily discernible. Here, we imagine a uniform shared-memory semantics for all modes. The language of types with shifts we have already described; processes change only to the extent that the values and continuations in them change. We have a new value $V$ for shifts $\text{shift}(y_k)$ and a new continuation $K$ which receives a shift $(\text{shift}(y_k) \Rightarrow P)$ where $\text{shift}(d_k) \circ (\text{shift}(y_k) \Rightarrow P) = [d_k/y_k]P$. These shifts can go in both directions and in this way cover the four different rules.

6.2 Dynamic Interactions Between Layers

In the sequential semantics, we have a different set of semantic objects, but they behave in the same manner as above. Assume the mode $k$ has a sequential semantics and $m$ and $r$ are generic so we represent them as threads.

\[
\begin{align*}
\text{eval}(c_k, c_k, \text{shift}(d_m)) & \Rightarrow \text{retm}(c_k, \text{shift}(d_m)) \\
\text{cell}(c_k, \text{shift}(d_m)), \text{thread}(e_r, \text{case} c_k (\text{shift}(y_m) \Rightarrow Q)) & \Rightarrow \text{thread}(e_r, [d_m/y_m]Q) \quad (\downarrow L)
\end{align*}
\]

\[
\begin{align*}
\text{thread}(c_m, \text{case} c_m (\text{shift}(y_k) \Rightarrow P)), \text{cell}(c_m, _) & \Rightarrow \text{cell}(c_m, \text{shift}(y_k) \Rightarrow P) \\
\text{cell}(c_m, \text{shift}(y_k) \Rightarrow P), \text{eval}(d_k, c_m, \text{shift}(d_k)) & \Rightarrow \text{eval}(d_k, [d_k/y_k]P) \quad (\uparrow L^0)
\end{align*}
\]

The last line (for $\uparrow L^0$) is noteworthy because evaluation, usually entirely sequential, might have to synchronize on the cell $c_m$ if $m$ is a concurrent mode and therefore allocates $\text{cell}(c_m, _)$. In an uninitialized state.

Further interactions take place in a cut. If we assume $k$ is a sequential mode and $m$ is a concurrent mode, then we have the following two cross-mode cuts.

\[
\begin{align*}
\text{thread}(c_m, x_k \leftarrow P ; Q) & \Rightarrow \text{eval}(a_k, [a/x]P), \text{cont}(a_k, c_m, [a/x]Q) \quad (a_k \text{ fresh}) \\
\text{eval}(c_k, x_m \leftarrow P ; Q) & \Rightarrow \text{thread}(a_m, [a/x]P), \text{cell}(a_m, _), \text{eval}(c_k, [a/x]Q) \quad (a_m \text{ fresh})
\end{align*}
\]

The invariants we are maintaining are that a sequential cell is always already available before it is referenced (first rule) and that a thread always has an uninitialized destination (second rule). As we can see, the mode of the fresh channel or location determines the behavior of the cut. Note also that we must have $k \geq m$ for the first rule, and $m \geq k$ in the second rule, or the given configurations would not be well-typed.
7 Reconstructing the Essence of SILL

SILL [52, 51, 22] embeds the concurrency primitives of linear logic in a functional language. Briefly, and omitting the presence of shared channels, we have

Functional types $\tau, \sigma ::= \tau \to \sigma \mid \tau \times \sigma \mid 1 \mid \tau + \sigma \mid \cdots \mid \{S \leftarrow S_1, \ldots, S_n\}$

Session types $S, T ::= S \to T \mid S \& T \mid S \otimes T \mid 1 \mid S \oplus T \mid \tau \land S \mid \tau \lor S$

where the functional layer is pure and sequential. One key construct is the functional type $\{S \leftarrow S_1, \ldots, S_n\}$ whose values are process expressions $x \{P\} \leftarrow y_1, \ldots, y_n$ such that $y_1 : S_1, \ldots, y_n : S_n \vdash P : (x : S)$. We simplify our task somewhat by discharging the antecedents, so that $\{S \leftarrow S_1, \ldots, S_n\}$ becomes $\{S_1 \rightarrow \cdots \rightarrow S_n \rightarrow S\}$, which requires only a simple local transformation of programs. Note that values of this type are closed process expressions that can be constructed and passed around in the functional layer. At the top level we have a concurrent program that can call into the functional layer to construct process expressions that are then concurrently executed using a bind construct that corresponds to a cut.

The other two significant constructs are $\tau \land A$ and $\tau \lor A$ that send and receive a (functional) value of type $\tau$ and continue with $A$, respectively. Because no restrictions are placed on $\tau$, this general formulation of SILL is more directly suitable for a shared memory semantics than a distributed message-passing semantics, even if it is formally agnostic.

We model this language with two modes, $S^*$ and $C$ such that $S^* \geq C$ and $\sigma(S^*) = \{W, C\}$ and $\sigma(C) = \{\}$. For now, however, we use the mode $S$ with $\sigma(S) = \{\}$; see Section 9.1 for the general case. We assign the sequential semantics to $S$ and the concurrent (shared-memory) semantics to $C$. We define a translation $[\cdot]$ which is overloaded for types, functional terms, and process expressions. We show only some exemplars.

$[\tau \to \sigma] = [\tau] \rightarrow_c [\sigma]$

$[(S)] = \uparrow^c [S]$

$[S \rightarrow_0 T] = [S] \rightarrow_c [T]$

$[\tau \lor S] = (\downarrow^c [\tau]) \rightarrow_c [S]$

Functional terms $M$ in SILL are typed with $\Psi \vdash M : \tau$, where $\Psi$ is a context of purely functional variables. We translate $M$ with respect to a destination $x$ such that $[\Psi]_x \vdash [M]_x :: (x_0 : [\tau])$, where $[\Psi]_x$ translates every $y : \sigma$ to $y_x : [\sigma]$. Processes are already in close correspondence and are translated compositionally, where $\Psi; \Gamma \vdash P : (x : S)$ entails that $[\Psi]_x; [\Gamma]_c \vdash [P] : (x_c : [S])$. First, key cases of the translation of functional terms $M$.

$[w]_x = x \leftarrow w$

$[\lambda w . M]_x = \text{case } x ((w, y) \Rightarrow [M]_y)$

$[M \ N]_y = x \leftarrow [M]_x ; w \leftarrow [N]_w ; x.(w, y)$

Note the strong analogy with translations to continuation-passing style. For example, the last line could be read as

$[M \ N]_y = [M](\lambda x. [N](\lambda w. x w y))$
except that continuations are not themselves λ-terms but named by destinations. For process expressions, the translation is entirely straightforward except for the new constructs such as \( \tau \supset S \). Here, we let \( v \) stand for the address of a value \( v \) of type \( \tau \) whose translation has type \( \downarrow S \). Here, we let \( v \) stand for the address of a value \( v \) of type \( \tau \) whose translation has type \( \downarrow S \).

The most interesting part are the functional constructs embedding process expressions. We define:

\[
\begin{align*}
[[S]] &= \uparrow \downarrow S \\
[x \leftarrow (P)]_y &= \text{case } y_\ell (\text{shift}(x_c) \Rightarrow [P]) \\
[x \leftarrow M ; Q] &= x_c \leftarrow (y_\ell \leftarrow [M]_y ; y_\ell \cdot \text{shift}(x_c)) ; Q
\end{align*}
\]

We can see that the operational behavior is correct: the outer cut introducing \( x_c \) spawns a new process that first computes \([M]\) sequentially to a value (which represents a suspended process). It then provides access to it along \( x_c \) which is used in \( Q \). Together, this is exactly the semantics of \( \{ \} E \) in SILL.

The biggest difference to SILL is that the functional layer here is linear. A generalization to a nonlinear one is unproblematic and sketched in Section 9. The choice that \( S > C \) makes it straightforward to allow weakening and contraction in the sequential layer. There are a few other features such as polymorphism that are beyond the scope of the present paper.

8 Reconstructing Futures

Futures [24] present the opposite phenomenon from SILL: within functional computation we create a future that immediately returns a promise and spawns a concurrent computation. Touching a promise blocks until its value has been computed. Futures have been a popular mechanism for parallel execution in both statically and dynamically typed languages, and they are also used to encapsulate communication primitives. The simpler mechanism of fork/join parallelism [11] may be a helpful warmup exercise and can be found in Appendix A.4.

For the moment we are interested in linear futures, that is, futures that must be used exactly once. In order to preserve their invariants, the sequential language must also be linear. Uses for linear futures (without a full formalization) in the efficient expression of certain parallel algorithms have already been explored in prior work [5].

We present futures in the form of an extension to the purely sequential language and show how it can be elaborated into the adjoint framework in a compositional manner. The first change we need to make is to the preorder between modes: we now postulate \( S \geq C \) and also \( C \geq S \) because we have mutual dependency between concurrent and sequential computation. We then have the following new type and expressions in the otherwise purely sequential language:

\[
\begin{align*}
\text{Types} & \quad A ::= \ldots \mid \text{fut } A \\
\text{Expressions} & \quad P ::= \ldots \mid (z \leftarrow \text{future } P ; x.(z)) \mid \text{touch } x ((y) \Rightarrow P)
\end{align*}
\]
We type them as

\[
\begin{align*}
\Gamma \vdash P :: (z : A) \\
\Gamma, z \vdash Q :: (w : C)
\end{align*}
\]

\[
\frac{\text{fut } R}{\Gamma \vdash (z \leftarrow \text{future } P ; x.\langle z \rangle) :: (x : \text{fut } A)}
\]

\[
\frac{\Gamma, x : \text{fut } A \vdash \text{touch } x ((z) \Rightarrow Q) :: (w : C)}{\text{fut } L}
\]

The intuition is that \((z \leftarrow \text{future } P ; x.\langle z \rangle)\) will spawn a new thread to evaluate \(P\) with destination \(z\), and immediately send the promise of \(z\) (written as \(\langle z \rangle\)) along \(x\). It behaves almost exactly like a cut, except that with the sequential semantics of Section 5 the promise would not be sent until the evaluation of \(P\) had terminated. Here, we want it to be evaluated concurrently. The construct \(\text{touch } x ((z) \Rightarrow Q)\) waits not only for a promise along \(x\) (which would be a standard \(\text{case}\) construct), but also for this promise to have been fulfilled by a value. More formally:

\[
\begin{align*}
\text{eval}(c, z \leftarrow \text{future } P ; c.\langle z \rangle) &\Rightarrow \text{eval}(e, [e/z]P), \text{retn}(c, \langle e \rangle) \quad (e \text{ fresh}) \\
\text{eval}(w, \text{touch } c ((z) \Rightarrow Q)), \text{cell}(c, \langle e \rangle), \text{retn}(e, W) &\Rightarrow \text{eval}(w, [e/z]Q), \text{cell}(e, W)
\end{align*}
\]

The first clause violates our previous invariant that there is exactly one object \(\text{eval}\) or \(\text{retn}\) in a configuration. We should expect this since with futures we have multiple threads of sequential control, each one represented by an \(\text{eval}\) or \(\text{retn}\). In the second clause we see that we explicitly synchronize by waiting until the cell \(e\) is filled with a value. It would be type-correct to substitute \(e\) for \(z\) without testing this condition, but then we would not have specified futures.

These constructs can already be expressed compositionally within the framework of adjoint logic. We define:

\[
\begin{align*}
\text{fut } A &\leadsto \downarrow_{\text{fut}} A_s \\
z \leftarrow \text{future } P ; x.\langle z \rangle &\leadsto y_c \leftarrow (z \leftarrow P ; \text{case } y_c (\text{shift}(u_s) \Rightarrow u_s \leftarrow z_s)) ; x_s.\text{shift}(y_c) \\
\text{touch } x ((z) \Rightarrow Q) &\leadsto \text{case } x_s (\text{shift}(y_c) \Rightarrow (z \leftarrow y_c.\text{shift}(z_s) ; Q))
\end{align*}
\]

The typing of the expansions and the arguments for their operational correctness can be found in Appendices A.5 and A.6.

9 Weakening and Contraction

For simplicity of presentation, so far all of our languages and semantics have been entirely linear, even though adjoint logic permits the use of nonlinear modes. Surprisingly few changes are required to add weakening and contraction and thereby obtain a much more expressive language. For example, we will have a fully expressive functional language as a component of SILL as well as ordinary (nonlinear) futures embedded in a (nonlinear) functional language.

Because our primary interest in this paper is a shared-memory semantics, we use a formulation where the structural rules of weakening and contraction are
The operational semantics is just as before, the only change being additional contraction, see [47].

The only change is the careful management of antecedents. We write $\Gamma_W$ for antecedents all of whose modes admit weakening (used in the zero-premise rules), and $\Gamma_C$ for antecedents all admitting contraction (used in cut). In some left rules, if the mode of a channel $x_m : A_m$ admits contraction, we may choose to retain it in the premise. The notation $(x_m : A_m)^\alpha$ expresses this, where $\alpha = 1$ means the assumption is present, and $\alpha = 0$ means that it is not. The typing rules can be found in Figures 5 to 7, now used in their full generality.

We generalize our semantic objects, allowing cells to be persistent if their mode satisfies contraction. A persistent object $\phi$ on the left-hand side of a rewriting rule is not consumed, but remains in the configuration [9]. We do not explicitly account for weakening, because ephemeral or persistent objects may simply be left over at the end of the computation. We write $!_m \phi$ for a semantic object which is persistent ($!_m \phi$ if $C \in \sigma(m)$ and otherwise ephemeral (written as just $\phi$ as we have so far).

1. $\text{thread}(c_m, P)$: thread $P$ with destination $c$; always ephemeral
2. $\text{cell}(c_m, \_)$: cell $c$ that has been allocated, but not yet written; always ephemeral
3. $!_m \text{cell}(c_m, W)$: cell $c$ containing $W$, either ephemeral or persistent

\[
\begin{align*}
\Gamma_C, \Gamma \vdash P :: (c_m : A_m) & \quad \Gamma_C, \Delta, \Gamma \vdash \text{thread}(c_m, P), \text{cell}(c_m, \_): (\Gamma_C, \Delta, c_m : A_m) \quad \Delta \vdash () :: \Delta \\
\Gamma_C, \Gamma \vdash W : A & \quad \Gamma_C, \Gamma \vdash \text{cell}(c_m, W) :: (\Gamma_C, \Delta, c_m : A_m) \quad \Gamma \vdash C_1 :: \Delta_1 \quad \Delta_1 \vdash C_2 :: \Delta_2
\end{align*}
\]

The operational semantics is just as before, the only change being additional $!_m$ prefixes for some cells.

\[
\begin{align*}
\text{thread}(c_k, x_m \leftarrow P : Q) & \rightarrow \text{thread}(a_m, [a_m/x_m]P), \text{cell}(a_m, \_), \text{thread}(c_k, [a_m/x_m]Q) \quad (a_m \text{ fresh}) \text{ cut/spawn} \\
!_m \text{cell}(c_m, W), \text{thread}(d_m, d_m \leftarrow c_m), \text{cell}(d_m, \_) & \rightarrow !_m \text{cell}(d_m, W) \quad \text{id/move} \\
\text{thread}(c_k, c_k \leftarrow p \leftarrow d_m) & \rightarrow \text{thread}(c_k, [c/x, d/y]P) \quad \text{for } x \leftarrow y = P \in \Sigma \text{ call} \\
!_m \text{cell}(c_m, V), \text{cell}(c_m, \_) & \rightarrow !_m \text{cell}(c_m, V) \quad (\oplus R^0, \oplus R^0, 1 R^0) \\
!_m \text{cell}(c_m, V), \text{thread}(e_k, \text{case } c_m K) & \rightarrow \text{thread}(e_k, V \circ K) \quad (\oplus L, \oplus L, 1 L) \\
\text{thread}(c_m, \text{case } c_m K), \text{cell}(c_m, \_) & \rightarrow !_m \text{cell}(c_m, K) \quad (-\circ R, \& R) \\
!_m \text{cell}(c_m, K), \text{thread}(d_m, c.V) & \rightarrow \text{thread}(d_m, V \circ K) \quad (-\circ L^0)
\end{align*}
\]

In the revised statement of type preservation, we need to allow additional channels to be provided by a configuration. This could arise in two situations: channels that admit weakening no longer have any clients, or a (fresh) channel admitting contraction should be visible in the interface.

**Theorem 9 (Type Preservation).** If $\Gamma \vdash C :: \Delta$ and $C \rightarrow C'$ then $\Gamma \vdash C' :: \Delta, \Delta'$, where all channels in $\Delta'$ admit either weakening or contraction.
The progress theorem does not have to change, but the proof has to accommodate some additional cases.

**Theorem 10 (Progress).** If \( \vdash C : \Delta \) then either

(i) \( C \rightarrow C' \) for some \( C' \), or

(ii) for every channel \( c_m : A \in \Delta \) there is an object \( \text{cell}(c_m, W) \in C \).

The sequential semantics changes in a completely analogous way, using the semantic objects

1. \( \text{eval}(c_m, P) \): evaluate \( P \) with destination \( c_m \); always ephemeral
2. \( \text{cont}(c_m, d_k, P) \): cont. waiting on \( c_m \) with destination \( d_k \); always ephemeral
3. \( \text{return}(c_m, W) \): return \( W \) to \( c_m \), always ephemeral
4. \( !m \text{cell}(c_m, W) \): cell \( c_m \) holding value \( W \); may be ephemeral or persistent

The results stated in the previous sections continue to hold under these extensions with some small modifications to accommodate persistence.

### 9.1 SILL Revisited

As already indicated in Section 7, we obtain SILL with the following preorder of modes, where \( S^* \) is sequential and subject to weakening and contraction, and \( C \) is concurrent and linear, and \( C^* \) is concurrent with shared channels but limited to \( A_{C^*} ::= \uparrow^C_{S^*} A_C \). The \( C^* \) and \( C \) layers together constitute intuitionistic linear logic (with the possibility of embedded functional computation in the form of \( \downarrow^C_{S^*} A_{C^*} \)).

\[ S^* > C^* > C \quad \text{with} \quad \sigma(S^*) = \sigma(C^*) = \{W, C\} \quad \text{and} \quad \sigma(C) = \{\} \]

The distinguishing characteristic of SILL is that we have two layers populated with a full set of connectives: the functional \( (S^*) \) and the linear concurrent \( (C) \) ones. The functional layer, since it is subject to weakening and contraction as usual, cannot refer to linear channels but it can compose concurrent computations. The linear layer admits a message-passing as well as a shared memory semantics, although without any restrictions a distributed semantics would have to serialize closures.

### 9.2 Futures Revisited

In the preorder for futures, the top two modes represent (a statically typed version of) traditional futures, where both the underlying functional language with sequential semantics \( (S^*) \) and the futures \( (C^*) \) are subject to weakening and contraction. The lower two modes represent linear futures \( C \) embedded in a linear functional language \( S \), as presented in Section 8. Here, the two concurrent layers (whether subject to weakening and contraction or not) are occupied only by shifts \( A_C ::= \uparrow^C_S A_S \) and \( A_{C^*} ::= \uparrow^C_{S^*} A_{S^*} \).
The distinguishing characteristic of futures is that the concurrent layer is occupied only by a single construct, namely the “future”. In the distributed setting this may be somewhat problematic since serializing closures may be necessary even for relatively simple examples, but it fits perfectly into the shared memory semantics.

Linearity is then an orthogonal consideration, allowing the eager deallocation of futures once they have been touched. However, this is only safely possible if the surrounding functional language has a clearly delineated linear fragment.

10 Conclusion

We have presented a shared-memory semantics for a rich session-typed language based on a semi-axiomatic sequent calculus for linear logic. We then generalized this to the setting of adjoint logic (augmented with recursion), where weakening and contraction are also allowed, to reconstruct both SILL and futures from first principles. This opens up the possibility of further combinations of semantics, such as message-passing with shared memory, or futures together with SILL, and so on.

There are several interesting items of future work. One is the thorough analysis of the semi-axiomatic sequent calculus from the proof-theoretic perspective, which is ongoing work [15]. Another is a more modular and general understanding of how to combine different operational interpretations of layers in adjoint logic. The semantic stage for this has been set with work on polarization, effects and co-effects including linearity (see, for example, [12]) and call-by-push-value and similarly encompassing frameworks ([34, 39]).

In other ongoing research we are trying to recover the correspondence to logic by restricting recursive types to least and greatest fixed points (which can also be viewed as inductive and coinductive types) [14]. This has to go along with a suitable restriction on recursively defined processes. Some interesting work along these lines has already been carried out starting on the logical side [17, 1, 40]. As far as we are aware, only Toninho et al. [53] and Lindley and Morris [37] explicitly address the question in the context of session types.

Finally, the adjoint framework permits us to formulate a type system with shared resources [2] that can be acquired, interacted with, and then released. This represents a departure from a pure Curry-Howard isomorphism in that computation proceeds not only by proof reduction, but also by proof construction (acquire) and proof destruction (release). For example, it is straightforward to model mutable shared memory where memory access is protected by mutexes. We conjecture that this form of sharing dovetails nicely with the approach developed here because of the same logical roots.
Acknowledgments. We would like to thank Henry DeYoung for discussions regarding the subject of this paper and comments on an earlier draft. This material is based upon work supported by the National Science Foundation under Grant CCF-1718267.

References

2012, Leibniz International Proceedings in Informatics, Fontainebleau, France (Sep 2012)


23. Guenot, N.: Session types, solos, and the computational contents of the sequent calculus (Dec 2014), talk at the Types Meeting


A Appendix

A.1 Purely Linear Multiplicatives

We provide a short explanation and example for linear implication in its asynchronous form derived from the semi-axiomatic sequent calculus. Reasoning along similar lines as for internal choice, we assign process expressions to $\to R$ and $\to L^0$.

$$\Gamma, u : A \vdash P :: (y : B)$$

$$\Gamma \vdash \text{case } x ((u, y) \Rightarrow P) :: (x : A \to B) \quad \to R$$

$$w : A, x : A \to B \vdash x.(w, y) :: (y : B) \quad \to L^0$$

The process for the left rule corresponds to a message $x.(w, y)$ that sends channel $w : A$ along $x$, together with a continuation $y : B$. For the right rule we have used a case construct so that all message receipt rules can have a uniform syntax: $u$ stands for the received channel of type $A$ and $y : B$ for the continuation channel.

The operational rule is then rather straightforward:

$$\text{proc}(c, \text{case } c ((u, y) \Rightarrow P)), \text{proc}(d, c.(e, d)) \mapsto \text{proc}(d, [e/u, d/y]P)$$

As an example, consider a proof of $x : A \to B, y : B \to C \vdash z : A \to C$:

$$x : A \to B, y : B \to C \vdash \text{case } z ((u, z') \Rightarrow y' \leftarrow x.(u, y', z') :: (z : A \to C)$$

In this example, $u : A$, and the primed variables represent continuation channels $z' : C$, and $y' : B$.

Multiplicative conjunction is again dual, just reversing the role of sender and receiver. Rules can be found in Figure 6 (or Figure 8 for the purely linear form).

A.2 Purely Linear ADJ0

The process assignment and typing rules for the linear semi-axiomatic sequent calculus (plus recursive processes) can be found in Figure 8. Alternatively, we can think of this as adjoint logic with a single mode without structural rules and shifts, augmented with recursion.

A.3 Typing of the Translation of SILL

We verify the typing of the translations from linear adjoint logic to SILL, first for the introduction

$$\Psi; \cdot \vdash P :: (x : S)$$

$$\Psi \vdash x \leftarrow \{P\} : \{S\} \quad \{ \} I \quad [\Psi]_s \vdash [P] :: (x_c : [S])$$

$$[\Psi]_s \vdash \text{case } y_6 (\text{shift}(x_c) \Rightarrow [P]) :: (y_6 : \uparrow^c [S]) \quad \uparrow R$$
y : A ⊢ x ← y :: (x : A)  id  Δ ⊢ P :: (x : A)  Δ′, x : A ⊢ Q :: (z : C)  cut  

y : B ⊢ p :: (x : A) ∈ Σ  call  

w : B ⊢ z ← p ← w :: (z : A)  

(i ∈ L)  \( y : A^i ⊢ i(y) :: (x : ∪ \{ \ell : A^i \}_{\ell ∈ L}) \) ⊥R \( \Gamma, y : A^i ⊢ Q_\ell :: (z : C) \) (for all \( \ell \in L \)) ⊥L  

w : A, y : B ⊢ x.(w, y) :: (x : A ⊔ B) ⊥R \( \Gamma, w : A, y : B ⊢ P :: (z : C) \) ⊥L  

\( \Gamma ⊢ P_\ell :: (y : A^i) \) (for all \( \ell \in L \)) &R \( \Gamma, w : A ⊔ B ⊢ P :: (y : B) \) &L \( \Gamma ⊢ x_\ell :: (w, y) \Rightarrow P :: (x : A ⊔ B) \) &sL \( \Gamma ⊢ x :: (w, y) \Rightarrow P :: (x : A ⊔ B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x.(w, y) :: (y : B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x :: (y : B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x.(w, y) :: (x : A ⊔ B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x :: (y : B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x.(w, y) :: (x : A ⊔ B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x :: (y : B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x.(w, y) :: (x : A ⊔ B) \) &sL \( \Gamma, w : A ⊔ B ⊢ x :: (y : B) \) &sL 

and then for the elimination 

\[ Ψ ; M : \{ S \} \quad Ψ ; \Gamma, x : S ⊢ Q :: (z : T) \]  

\[ Ψ ; \Gamma ⊢ x ← M ; Q :: (z : T) \]  

\[ \{ \} E \]  

\[ \vdots \]  

\[ D \quad [Ψ]_Y, [Γ]_C, xc : [S] ⊢ Q :: (xc : [T]) \]  

\[ [Ψ]_Y, [Γ]_C ⊢ (xs ← [M]_Y ; ys.shift(xc)) :: [Q] :: (xc : [T]) \]  

\[ \text{cut} \]  

where 

\[ [Ψ]_Y ⊢ [M]_Y :: (ys : ↑_c[S]) \]  

\[ ys : ↑_c[S] ⊢ ys.shift(xc) :: (xc : [S]) \]  

\[ \uparrow L \]  

\[ D = [Ψ]_Y ⊢ (ys ← [M]_Y ; ys.shift(xc)) :: (xc : [S]) \]  

\[ \text{cut} \]  

A.4 Reconstructing Fork/Join Parallelism

We use fork/join parallelism as a simple, yet practically highly successful paradigm that provides preliminary insights into futures. In our reconstruction, we can fork
a new concurrent computation and use `join` to synchronize on its value. In the context of functional languages we can think of fork/join parallelism as “second-class futures” in that we cannot encapsulate the future as a functional value.

The mode structure for this is quite straightforward: in addition to mode $S$ for sequential computation we have $C$ for concurrent computation, where $S > C$. The sequential layer remains unchanged, and in the concurrent layer we only have $A_c := \frac{\iota_c}{\iota_c} A_s$.

Now assume we have a sequential computation $\Gamma \vdash P :: (x_5 : A_5)$ we would like to run concurrently, making its eventual value accessible in $Q$. We write this as

$$y_c \leftarrow (x_c \leftarrow P ; y_c.shift(x_5)) \ ; Q$$

which is typed with the derived rule

$$\Gamma \vdash P :: (x_5 : A_5) \quad \Delta, y_c : \frac{\iota_c}{\iota_c} A_s \vdash Q :: (u_5 : C_5)$$

$$\Gamma, \Delta \vdash (y_c \leftarrow (x_c \leftarrow P ; y_c.shift(x_5)) ; Q) :: (u_5 : C_5)$$

Because the outer cut is concurrent, this spawns a new thread executing $(x_c \leftarrow P ; d_c.shift(x_5))$ for a freshly chosen $d_c$, the eventual location of the result. At the same time it continues with the execution of $Q$. In the first thread, we have a sequential cut, so we execute $P$ with a new destination $e_5$, leaving the cell $cell(d_c, \_)$ empty for now and creating a continuation $cont(e_5, d_c, d_c.shift(e_5))$ waiting for the value of $P$ to be returned to $e_5$.

Meanwhile, we may synchronize in the evaluation of $Q$. We accomplish this with the phrase

```plaintext
\textbf{case} y_c (shift(x_5) \Rightarrow R)
```

which makes the value of the promise $y_c$ available as $x_5$ in $R$. It is typed with

$$\Delta, x_5 : A_5 \vdash R :: (u_5 :: D_5)$$

$$\Delta, y_c : \frac{\iota_c}{\iota_c} A_s \vdash \textbf{case} y_c (shift(x_5) \Rightarrow R) :: (u_5 :: D_5) \quad \downarrow L$$

In our operational narrative, $y_c$ will have been replaced by the promise $d_c$ when this expression is reached. It will now block until $cell(d_c, \_)$ has been filled.

When the executing $P$ returns a value to $e_5$ the continuation $d_c.shift(e_5)$ is invoked, which writes the value $\text{shift}(e_5)$ into cell $d_c$. This unlocks the case $d_c$ and continues with the evaluation of $[e_5/x_5]R$.

In our formal semantics:

\[
\begin{align*}
\text{eval} & (w_5, y_c \leftarrow (x_c \leftarrow P ; y_c.shift(x_5)) \ ; Q) \\
\Rightarrow & \text{thread}(d_c, x_c \leftarrow P ; d_c.shift(x_5)), cell(d_c, \_), \text{eval}(w_5, d_c/y_c)Q \\
\Rightarrow & \text{eval}(e_5, [e_5/x_5]P), cont(e_5, d_c, d_c.shift(e_5)), cell(d_c, \_), \text{eval}(w_5, d_c/y_c)Q
\end{align*}
\]

We see that continuation for $e_5$ is the bridge between the process $P$ and $Q$ which may decide to wait for its value. When the thread computing $Q$ joins the parallel thread $d_c$ we have the situation

\[
\begin{align*}
\text{eval}(e_5, [e_5/x_5]P), cont(e_5, d_c, d_c.shift(e_5)), cell(d_c, \_), \text{eval}(w_5, \text{case } d_c (shift(x_5) \Rightarrow R))
\end{align*}
\]
so evaluation of the case $d_c$ blocks until $d_c$ has been filled. This happens when the computation of $P$ returns a value to $e_5$.

\[
\text{eval}(e_5, W), \text{cont}(e_5, d_c, \text{shift}(e_5)), \text{cell}(d_c, \ldots), \text{eval}(v_5, \text{case } d_c (\text{shift}(x_5) \Rightarrow R)) \\
\Rightarrow \text{cell}(e_5, W), \text{thread}(d_c, d_c, \text{shift}(e_5)), \text{cell}(d_c, \ldots), \text{eval}(v_5, \text{case } d_c (\text{shift}(x_5) \Rightarrow R)) \\
\Rightarrow \text{cell}(e_5, W), \text{eval}(v_5, \text{shift}(e_5)), \text{eval}(v_5, \text{case } d_c (\text{shift}(x_5) \Rightarrow R)) \\
\Rightarrow \text{cell}(e_5, W), \text{eval}(v_5, [e_5/x_5 R])
\]

### A.5 Typing of the Translation of Futures

We show how to type the key components of the translation of futures into ADJ0.

\[
\begin{align*}
\Gamma \vdash P :: (z_5 : A_5) & \quad \text{id} \\
\Gamma \vdash z_5 :: A_5 & \quad \text{case } y_c (\text{shift}(u_5) \Rightarrow u_5 \leftarrow z_5) :: (y_c : \Gamma^z A_5) \\
\infer{\Gamma \vdash (y_c \leftarrow (z_5 \leftarrow P); \text{case } y_c (\text{shift}(u_5) \Rightarrow u_5 \leftarrow z_5); x_5, \text{shift}(y_c)) :: (x_5 : \Gamma^zl A_5)}{\Gamma \vdash z_5 \leftarrow P; \text{case } y_c (\text{shift}(u_5) \Rightarrow u_5 \leftarrow z_5) :: (y_c : \Gamma^z A_5) \quad \Gamma \vdash \text{case } y_c (\text{shift}(z_5) :: (z_5 : A_5) \quad \Gamma, z_5 : A_5 \vdash Q :: (w : C) \quad \text{cut} \\
\text{cut}}
\end{align*}
\]

### A.6 Simulation of the Dynamics of Futures

We illustrate how the dynamics of futures is simulated in ADJ0. First, the creation of the future.

\[
\begin{align*}
\text{eval}(c_5, y_c \leftarrow (z_5 \leftarrow P); \text{case } y_c (\text{shift}(u_5) \Rightarrow u_5 \leftarrow z_5)); C_5, \text{shift}(y_c)) \\
\Rightarrow \text{eval}(d_c, z_5 \leftarrow P; \text{case } d_c (\text{shift}(u_5) \Rightarrow u_5 \leftarrow z_5), \text{eval}(c_5, C_5, \text{shift}(d_c)) \\
\Rightarrow \text{eval}(e_5, P), \text{cont}(e_5, d_c, \text{case } d_c (\text{shift}(u_5) \Rightarrow u_5 \leftarrow e_5)), \text{retn}(c_5, \text{shift}(d_c))
\end{align*}
\]

At this point, $P$ executes sequentially with destination $e_5$ and the promise $d_c$ has been returned to $c_5$. Meanwhile, an expression touching the future $c_5$ executes as follows:

\[
\begin{align*}
\text{eval}(w, \text{case } c_5 (\text{shift}(y_c) \Rightarrow (z_5 \leftarrow y_c, \text{shift}(z_5) \Rightarrow Q))); \text{cell}(c_5, \text{shift}(d_c)) \\
\Rightarrow \text{eval}(w, z_5 \leftarrow d_c, \text{shift}(z_5) \Rightarrow Q) \\
\Rightarrow \text{eval}(f_s, d_c, \text{shift}(f_s)), \text{cont}(f_s, w, [f_s/z_5 Q])
\end{align*}
\]
Now \( \text{eval}(f_s, d_c.\text{shift}(f_s)) \) is blocked until the cell \( d_c \) is filled, which takes places when \( \text{eval}(e_s, P) \) finishes and invokes its continuation. Note that an \( \text{eval} \) object can only block if it attempts to read from a cell with concurrent mode, since otherwise the sequential semantics enforces that cells are always filled before they can be referenced. When the value \( V \) of the future has been computed, we continue with

\[
\begin{align*}
\text{eval}(f_s, d_c.\text{shift}(f_s)), \text{cell}(d_c, \text{shift}(u_s)) & \Rightarrow u_s \leftarrow e_s), \text{cont}(f_s, w, [f_s/z_s]Q), \text{cell}(e_s, V) \\
& \Rightarrow \text{eval}(f_s, f_s \leftarrow e_s), \text{cont}(f_s, w, [f_s/z_s]Q), \text{cell}(e_s, V) \\
& \Rightarrow \text{eval}(w, e_s/f_s, [f_s/z_s]Q), \text{cell}(e_s, V) \\
& = \text{eval}(w, e_s/z_s)Q), \text{cell}(e_s, V)
\end{align*}
\]

So we can see that this is the expected configuration given by the direct semantics.