Relating Message Passing and Shared Memory, Proof-Theoretically*

Frank Pfenning[1] and Klaas Pruiksma[2]

[1] Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
fp@cs.cmu.edu
[2] University of Stuttgart, Stuttgart, Germany
klaas.pruiksma@sec.uni-stuttgart.de

Abstract. We exhibit a strong bisimulation between asynchronous message passing concurrency with session types and shared memory concurrency with futures. A key observation is that both arise from closely related interpretations of the semi-axiomatic sequent calculus with recursive definitions, which provides a unifying framework. As a further result we show that the bisimulation applies to both linear and nonlinear versions of the two languages.

Keywords: Session types · futures · bisimulation.

1 Introduction

At first sight, message passing concurrency is quite different from shared memory concurrency. Then we remember the well-known encoding of shared memory cells in the π-calculus [25] and also implementations of message passing abstractions using shared memory [20]. Such mutual encodings are significant, but far from straightforward and difficult to reason about rigorously.

This paper is an attempt to reduce the relationship to its essence in the typed setting. On one side we have a language for asynchronous message passing using session types [14]. On the other side we have typed futures [13, 23]. The key conceptual tools in understanding their relationship are the *semi-axiomatic sequent calculus* [10] (SAX) and the *polarities* of the connectives [1, 18]. We introduce the relevant aspects of these tools one by one. At the end, we arrive at two *strong bisimulations* between the two sides, one each for linear and nonlinear versions of message passing and shared memory. This is the closest connection we could reasonably hope for.

2 Proof Reduction as Communication

The so-called *Curry-Howard correspondence* [8, 16] is often summarized as saying that propositions are types and proofs are programs. This neglects an even deeper

* Notes to an invited talk by the first author

aspect of the relationship between logic and computation: *proof reduction is computation*. In natural deduction, the fundamental engine of proof reduction is substitution; in Hilbert-style calculi it is combinatory reduction. What about sequent calculus? At the logical level, we write a sequent as

$$A_1, \ldots, A_n \vdash C$$

where propositions A_i are the *antecedents* and C is the *succedent*. In our investigation, the succedent will always be a singleton since we restrict ourselves to *intuitionistic logic*.

We examine the computational interpretation first in the context of a *purely linear calculus*, that is, we take exchange between antecedents for granted, but we allow neither weakening nor contraction. We write

$$\begin{array}{c} P \\ a_1 : A_1, \ldots, a_n : A_n \vdash c : C \end{array}$$

where the a_i and c stand for *means of communication* for the proof (= process) P. Under a message passing interpretation names a_i and c stand for *channels*; under a shared memory interpretation, they stand for *addresses*. We use Γ and Δ to stand for a collection of antecedents, always presupposing that all names a_i and c are distinct.

For the moment, we stick with the message passing interpretation. Then *cut* represents two processes P and Q that are connected via a private communication channel x.

$$\dfrac{\begin{array}{cc} P(x) & Q(x) \\ \Gamma_1 \vdash x : A & \Gamma_2, x : A \vdash c : C \end{array}}{\Gamma_1, \Gamma_2 \vdash c : C}\ \mathsf{cut}$$

It is a *private* communication channel because by our general presupposition x must be different from c and must not already occur in Γ_1 or Γ_2.

The *propositions* A are interpreted as *session types* that govern the particular kinds of messages that are exchanged along a private channel $x : A$. As an example we consider $A \oplus B$, which is the linear rendering of disjunction $A \vee B$. Here is a *principal cut reduction* for this proposition/type.

$$\dfrac{\dfrac{\begin{array}{c} P'(x') \\ \Gamma_1 \vdash x' : A \end{array}}{\Gamma_1 \vdash x : A \oplus B}\ \oplus R_1 \quad \dfrac{\begin{array}{cc} Q_1(x') & Q_2(x') \\ \Gamma_2, x' : A \vdash c : C & \Gamma_2, x' : B \vdash c : C \end{array}}{\Gamma_2, x : A \oplus B \vdash c : C}\ \oplus L}{\Gamma_1, \Gamma_2 \vdash c : C}\ \mathsf{cut}$$

$$\longrightarrow \qquad \dfrac{\begin{array}{cc} P'(x') & Q_1(x') \\ \Gamma_1 \vdash x' : A & \Gamma_2, x' : A \vdash c : C \end{array}}{\Gamma_1, \Gamma_2 \vdash c : C}\ \mathsf{cut}$$

We see that the first premise of the cut (rule $\oplus R_1$) has a single premise, while the second premise of the cut (rule $\oplus L$) has two branches (Q_1 and Q_2). In essence, the proof of the first premise (either $\oplus R_1$ or $\oplus R_2$) selects one of the two branches of the second premise. So the information flows along the channel x from left to right. The message itself therefore should be one bit to indicate whether the first or second branch was chosen.

In this example, the communication between processes P and Q (the two premises of the cut) is *synchronous* because P evolves to P' and Q evolves to Q_1. Other connectives of linear logic follow the same pattern and we conclude [4, 5]:

> *Principal cut reduction in the linear sequent calculus corresponds to synchronous message passing communication.*

3 Asynchronous Communication

In order to model *asynchronous* communication proof-theoretically we should have a proof that corresponds to a *message*. The salient aspects of a message are that (a) it carries relevant information, and (b) it does not have a continuation. We can achieve both if we represent *messages as axioms*.

Continuing the example from the previous section, we have two axioms for disjunction, where we write X for axiom instead of R for right rule.

$$\frac{}{a : A \vdash c : A \oplus B}\ \oplus X_1 \qquad \frac{}{b : B \vdash c : A \oplus B}\ \oplus X_2$$

The principal case of cut then becomes

$$\dfrac{\dfrac{}{a : A \vdash x : A \oplus B}\ \oplus X_1 \quad \dfrac{\begin{array}{cc} Q_1(x') & Q_2(x') \\ \Gamma_2, x' : A \vdash c : C & \Gamma_2, x' : B \vdash c : C \end{array}}{\Gamma_2, x : A \oplus B \vdash c : C}\ \oplus L}{\Gamma_2, a : A \vdash c : C}\ \mathsf{cut}$$

$$\longrightarrow \qquad \begin{array}{c} Q_1(a) \\ \Gamma_2, a : A \vdash c : C \end{array}$$

We see that the reduction corresponds to the message represented by the first premise being received by the second premise (process Q that ends in $\oplus L$). The message "disappears" and process Q continues as $[a/x']Q_1(x')$ which we write just as $Q_1(a)$. And while we have changed the two conventional $\oplus R$ rules into axioms, the $\oplus L$ rules remains the same.

We also observe that the message contains not only the bit to choose the first or second branch, it also contains a *continuation channel* a. This would be either of type A or B, depending on whether it is an instance of $\oplus X_1$ or $\oplus X_2$.

We can continue this pattern. For each connective of (purely) linear logic, either the right or left rule is invertible and the other one is noninvertible. Intuitively, the invertible rule carries no specific information (after all, the premise(s)

are derivable iff the conclusion is) while the noninvertible rule makes a choice. Therefore, we turn all noninvertible rules into axioms and keep the invertible rules as they are. The result is the linear semi-axiomatic sequent calculus [10, 9] (Semi-axiomatic since half the usual rules are now axioms.) We conclude:

> *Principal cut reduction in the semi-axiomatic sequent calculus corresponds to asynchronous message passing communication.*

The reader might wonder how we send a message, now that, for example, the $\oplus R_1$ and $\oplus R_2$ processes are no longer available. The solution is actually the same as in the asynchronous π-calculus: we use cut (= parallel composition) itself. For example,

$$\dfrac{\begin{array}{c}P(x)\\ \Gamma_1 \vdash x : A\end{array} \quad \dfrac{}{x : A \vdash c : A \oplus B}\,\oplus X_1}{\Gamma_1 \vdash c : A \oplus B}\,\mathsf{cut}$$

sends the message "*choose the first branch and continue with channel x*" along the channel c. We retain our connection to the receiving process via the new continuation channel x. This technique is made explicit by Kobayashi et al. [17] and can also be identified in other examples for the π-calculus [19, 25].

4 A Language for Asynchronous Communication

Following the motifs in the previous section, we now present a complete language for asynchronous message passing communication. We also specify the typing rules and how processes behave dynamically. For the purpose of more readable examples, we generalize binary sums to $\oplus_{\ell \in L}(\ell : A_\ell)$ for a finite set L of labels and, correspondingly, binary additive conjunction to $\&_{\ell \in L}(\ell : A_\ell)$.

$$\begin{array}{lll}
\text{Types} & A, B, C ::= A \otimes B \mid 1 \mid \oplus_{\ell \in L}(\ell : A_\ell) & \text{(positive)} \\
& \quad \mid\ A \multimap B \mid \&_{\ell \in L}(\ell : A_\ell) & \text{(negative)} \\
& \quad \mid\ t & \text{(type names)} \\
\text{Contexts } \Gamma & ::= \cdot \mid \Gamma, x : A \mid \Gamma, a : A &
\end{array}$$

Type names represent *equirecursive* types whose definitions are collected in a global signature Σ. Correspondingly, we allow mutually recursive process definitions, collected in the same signature. We use a, b, and c for *channels* (which are runtime objects) and x, y, and z for *variables* occurring in a process that stand for channels. Strictly speaking, we should introduce a category of *symbol* which may either be a variable or a channel, but since the rules do not need to distinguish between them we just follow the convention that we use x, y, and z for variables that are bound in a process expression and a, b, and c for channels or variables that are free. At runtime, a process will have only free channels and internally bound variables.

Processes are typed with

$$\underbrace{a_1 : A_1, \ldots, a_n : A_n}_{\text{use}} \vdash P :: \underbrace{(c : C)}_{\text{provide}}$$

where we say process P *provides* channel c and *uses* channels a_i. We also refer to P as a *client* of a_i and a *provider* of c.

Channels carry messages with *small values* V, which are either pairs of channels $\langle a, b\rangle$, a unit message $\langle\,\rangle$, or tagged channels $k(a)$ for labels k. The direction of the message depends on the *polarity* of the types. For a channel of *positive* type C the provider will send a message and the client will receive it. For a *negative* type the client will send a message and the provider will receive it.

4.1 The Dynamics of Process Configurations

We describe the state of the computation by a multiset of *semantic objects* we call a *configuration*. The possible state transitions are defined by *multiset rewriting rules* [7]. The left-hand side of a rule is matched against some objects in the configuration which are then replaced by the right-hand side. Later, we will refine this point of view slightly to allow *persistent objects* that always remain in a configuration.

As a first example, consider the construct $x \leftarrow P(x)\,;\,Q(x)$. A process of this form will allocate a new private channel a that is provided by $P(a)$ and used by $Q(a)$. Logically, it is a cut.

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, A \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\ \mathsf{cut} \qquad \frac{\Gamma_1 \vdash P(x) :: (x : A) \quad \Gamma_2, x : A \vdash Q(x) :: (c : C)}{\Gamma_1, \Gamma_2 \vdash (x \leftarrow P(x)\,;\,Q(x)) :: (c : C)}\ \mathsf{cut}$$

Our first semantic object is $\mathsf{proc}\ P$ that represents a running process P. In the dynamics, a cut process evolves into two.

$$\mathsf{proc}\ (x \leftarrow P(x)\,;\,Q(x)) \quad \mapsto \quad \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) \qquad (a \text{ a fresh channel})$$

We now dive into the meaning of each of the logical connectives, extracting their computational meaning.

4.2 Positive Connectives

Internal choice $\oplus_{\ell \in L}(\ell : A_\ell)$. As motivated in the preceding section, the right rules for sums are replaced by axioms. At the same time, we generalize from binary disjunction to finite sums, indexed by a label set L. This is a strict generalization under the definition $A \oplus B \triangleq (\mathtt{inl} : A) \oplus (\mathtt{inr} : B)$

$$\frac{}{A \vdash A \oplus B}\ \oplus X_1 \qquad \frac{}{B \vdash A \oplus B}\ \oplus X_2$$

$$\frac{(k \in L)}{b : A_k \vdash \mathbf{send}^+\ a\ k(b) :: (a : \oplus_{\ell \in L}(\ell : A_\ell))}\ \oplus X$$

The intent is for the process $\mathbf{send}^+\ a\ k(b)$ to send the tagged channel $k(b)$ along channel a. The polarity annotation of the **send** construct is not syntactically necessary, but the redundant information will be helpful later in formulating the connection between message passing and futures. In order to express the dynamics, we use a second kind of semantic object $\mathsf{msg}\ a\ V$, representing the value V as a message on channel a. Computationally, a sending process "becomes" a message.

$$\mathsf{proc}\ (\mathbf{send}^+\ a\ k(b)) \quad \mapsto \quad \mathsf{msg}^+\ a\ k(b)$$

The left rule for sums branches on the label received.

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\ \oplus L$$

$$\frac{\Gamma, x : A_\ell \vdash P_\ell(x) :: (d : C) \quad (\forall \ell \in L)}{\Gamma, c : \oplus_{\ell \in L}(\ell : A_\ell) \vdash \mathbf{recv}^+\ c\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} :: (d : C)}\ \oplus L$$

A receiving process *blocks* until a message arrives. In the dynamics we represent a process blocked on channel a as a continuation object $\mathsf{cont}\ a\ K$. Here K is the continuation to invoke once a message has arrived. In the case of sums, this is the branching construct.

$$\mathsf{proc}\ (\mathbf{recv}^+\ a\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}) \quad \mapsto \quad \mathsf{cont}^+\ a\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}$$

Messages always interact with continuations. Here, the message selects one of the branches and also carries the continuation channel for subsequent communication.

$$\mathsf{msg}^+\ a\ k(b), \mathsf{cont}^+\ a\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} \quad \mapsto \quad \mathsf{proc}\ P_k(b)$$

Pairs $A \otimes B$. Because $A \otimes B$ is a positive type, we turn the usual right rule of the sequent calculus into an axiom.

$$\frac{}{A, B \vdash A \otimes B}\ \otimes X \qquad \frac{}{a : A, b : B \vdash \mathbf{send}^+\ c\ \langle a, b\rangle :: (c : A \otimes B)}\ \otimes X$$

As for sums, a sending process simply becomes a message.

$$\mathsf{proc}\ (\mathbf{send}^+\ a\ \langle b, c\rangle) \quad \mapsto \quad \mathsf{msg}^+\ a\ \langle b, c\rangle$$

The left rule of the sequent calculus corresponds to the receipt of a message.

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes L \qquad \frac{\Gamma, x : A, y : B \vdash P(x, y) :: (d : C)}{\Gamma, c : A \otimes B \vdash \mathbf{recv}^+\ c\ (\langle x, y\rangle \Rightarrow P(x, y)) :; (d : C)}\ \otimes L$$

Again, just as for sums, a process receiving along a channel c will block until the message arrives. This is modeled by turning it into a continuation, which can then interact with a message.

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{recv}^+\ c\ (\langle x, y\rangle \Rightarrow P(x, y))) & \mapsto & \mathsf{cont}^+\ c\ (\langle x, y\rangle \Rightarrow P(x, y)) \\ \mathsf{msg}^+\ c\ \langle a, b\rangle, \mathsf{cont}^+\ c\ (\langle x, y\rangle \Rightarrow P(x, y)) & \mapsto & \mathsf{proc}\ P(a, b) \end{array}$$

Unit 1. The (multiplicative) unit type 1 is also positive. Instead of a pair of channels, messages of unit type are just $\langle\,\rangle$ and carry no information, except that there is a message. The rules are the nullary versions of the rules for $A \otimes B$.

$$\frac{}{\cdot \vdash 1}\ 1X \qquad \frac{}{\cdot \vdash \mathbf{send}^+\ c\ \langle\,\rangle :: (c : 1)}\ \otimes X$$

$$\frac{\Gamma \vdash C}{\Gamma, 1 \vdash C}\ 1L \qquad \frac{\Gamma \vdash P :: (d : C)}{\Gamma, c : 1 \vdash \mathbf{recv}^+\ c\ (\langle\,\rangle \Rightarrow P) :: (d : C)}\ 1L$$

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{send}^+\ c\ \langle\,\rangle) & \mapsto & \mathsf{msg}^+\ c\ \langle\,\rangle \\ \mathsf{proc}\ (\mathbf{recv}^+\ c\ (\langle\,\rangle \Rightarrow P)) & \mapsto & \mathsf{cont}^+\ c\ (\langle\,\rangle \Rightarrow P) \\ \mathsf{msg}^+\ c\ \langle\,\rangle, \mathsf{cont}^+\ c\ (\langle\,\rangle \Rightarrow P) & \mapsto & \mathsf{proc}\ P \end{array}$$

4.3 Refactoring the Rules of Computation

At this point we reflect on the dynamic rules and we see that we can refactor them, since both sending and receiving processes always turn into messages or continuations, respectively.

$$\begin{array}{lcll} \mathsf{proc}\ (x \leftarrow P(x)\ ;\ Q(x)) & \mapsto & \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a \text{ fresh}) \\ \mathsf{proc}\ (\mathbf{send}^+\ c\ V) & \mapsto & \mathsf{msg}^+\ c\ V & \\ \mathsf{proc}\ (\mathbf{recv}^+\ c\ K) & \mapsto & \mathsf{cont}^+\ c\ K & \\ \mathsf{msg}^+\ c\ V, \mathsf{cont}^+\ c\ K & \mapsto & \mathsf{proc}\ (V \rhd K) & \end{array}$$

Passing a value to a continuation is handled as a separate operation.

$$\begin{array}{lll} k(a)\ \rhd (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} = P_k(a) & (\oplus) \\ \langle a, b\rangle \rhd (\langle x, y\rangle \Rightarrow P(x, y)) = P(a, b) & (\otimes) \\ \langle\,\rangle \rhd (\langle\,\rangle \Rightarrow P) = P & (1) \end{array}$$

4.4 Process Definitions

Recall that all process definitions are collected in a global signature. At a call site we just check that the types of the channel used and provided match those described in the type declaration for a process.

$$\frac{(x_1 : A_1, \ldots, x_n : A_n \vdash f :: (z : C)) \in \Sigma}{a_1 : A_1, \ldots, a_n : A_n \vdash f\ c\ [a_1, \ldots, a_n] :: (c : C)}\ \mathsf{call}$$

In the first position after f is always the channel provided by the definition followed by a the list of channels used.

$$\begin{array}{l} \mathsf{proc}\ (\mathbf{call}\ f\ c\ [a_1, \ldots, a_n]) \quad \mapsto \quad \mathsf{proc}\ P(c, a_1, \ldots, a_n) \\ \text{for } (f\ z\ [x_1, \ldots, x_n] = P(z, x_1, \ldots, x_n)) \in \Sigma \end{array}$$

4.5 Some Examples

Even though our language is quite incomplete, we can already give some small examples. First, a process that flips a bit. We do not give an an explicit type declaration of the process *flip*, but show the type of the channel it provides (always first, here y) and the types of the channels it uses (here just x) in the left-hand side of the definition. We use sans serif for type names, `fixed width` for labels, **bold** for language keywords, and *italics* for process names.

$$
\begin{array}{l}
\mathsf{bit} = (\mathtt{b0} : 1) \oplus (\mathtt{b1} : 1) \\
\mathit{flip}\ (y : \mathsf{bit})\ [x : \mathsf{bit}] = \\
\quad \mathbf{recv}\ x\ (\ \mathtt{b0}(u) \Rightarrow \mathbf{send}\ y\ \mathtt{b1}(u) \\
\qquad\quad \mid \mathtt{b1}(u) \Rightarrow \mathbf{send}\ y\ \mathtt{b0}(u))
\end{array}
$$

Slightly more interesting is a recursive type that models an infinite stream of bits, and a process that flips them in turn.

$$
\begin{array}{l}
\mathsf{bits} = (\mathtt{b0} : \mathsf{bits}) \oplus (\mathtt{b1} : \mathsf{bits}) \\
\mathit{flips}\ (ys : \mathsf{bits})\ [xs : \mathsf{bits}] = \\
\quad \mathbf{recv}\ xs\ (\ \mathtt{b0}(xs') \Rightarrow ys' \leftarrow \mathbf{call}\ \mathit{flips}\ ys'\ [xs']\ ; \\
\qquad\qquad\qquad\qquad\ \ \mathbf{send}\ ys\ \mathtt{b1}(ys') \\
\qquad\quad \mid \mathtt{b1}(xs') \Rightarrow ys' \leftarrow \mathbf{call}\ \mathit{flips}\ ys'\ [xs']\ ; \\
\qquad\qquad\qquad\qquad\ \ \mathbf{send}\ ys\ \mathtt{b0}(ys')\ )
\end{array}
$$

Next, a simple pipeline of two bit-flipping processes which should be the identity, with some delay between incoming and outgoing messages.

$$
\begin{array}{l}
\mathsf{bits} = (\mathtt{b0} : \mathsf{bits}) \oplus (\mathtt{b1} : \mathsf{bits}) \\
\mathit{flip2}\ (zs : \mathsf{bits})\ [xs : \mathsf{bits}] = \\
\quad ys \leftarrow \mathbf{call}\ \mathit{flips}\ ys\ [xs]\ ; \\
\quad \mathbf{call}\ \mathit{flips}\ zs\ [ys]
\end{array}
$$

A very similar type is that of a binary number, where zero is represented by the label `e` followed by the unit. We start programming processes representing zero and computing the successor of a given stream (assuming the least significant bit arrives first).

$$
\mathsf{bin} = (\mathtt{b0} : \mathsf{bits}) \oplus (\mathtt{b1} : \mathsf{bits}) \oplus (\mathtt{e} : 1)
$$

$$
\begin{array}{l}
\mathit{zero}\ (y : \mathsf{bin})\ [\,] = \\
\quad u \leftarrow \mathbf{send}\ u\ \langle\,\rangle\ ; \\
\quad \mathbf{send}\ y\ \mathtt{e}(u)
\end{array}
$$

$$
\begin{array}{l}
\mathit{succ}\ (y : \mathsf{bin})\ [x : \mathsf{bin}] = \\
\quad \mathbf{recv}\ x\ (\ \mathtt{b0}(x') \Rightarrow \mathbf{send}\ y\ \mathtt{b1}(x') \\
\qquad\quad \mid \mathtt{b1}(x') \Rightarrow y' \leftarrow \mathbf{call}\ \mathit{succ}\ y'\ [x']\ ; \\
\qquad\qquad\qquad\qquad\ \ \mathbf{send}\ y\ \mathtt{b0}(y') \\
\qquad\quad \mid \mathtt{e}(u) \Rightarrow y' \leftarrow \mathbf{send}\ y'\ \mathtt{e}(u)\ ; \\
\qquad\qquad\qquad\ \ \mathbf{send}\ y\ \mathtt{b1}(y')\ )
\end{array}
$$

4.6 Negative Connectives

The negative connectives communicate in the opposite direction: the provider receives while the client sends. This is often the initial state of a provider/client system. In our language there are two such connectives: external choice $A \,\&\, B$ and linear implication $A \multimap B$. There could also be $\bot$ (dual to 1), but it would require an empty succedent, representing a process without a client. We choose to avoid this syntactic complication.

External choice $A \,\&\, B$. The right rule of *additive conjunction* or *external choice* of linear logic has two premises, and these remain the same in SAX since it is a negative connective. For programming convenience, we generalize from the binary to a finitary choice, where $A \,\&\, B \triangleq (\texttt{fst} : A) \,\&\, (\texttt{snd} : B)$.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\;\&R \qquad \frac{\Gamma \vdash P_\ell(x) :: (x : A_\ell) \quad (\forall \ell \in L)}{\Gamma \vdash \mathbf{recv}^-\; c\; (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} :: (c : \&_{\ell \in L}(\ell : A_\ell))}\;\&R$$

Symmetrically to the internal choice, the client now picks among the alternatives by sending a suitable message. In this way, a process providing an external choice represents an *object*, where each alternative is a *method*. This view of communication was already present in the original work on session types [15, 14].

$$\frac{}{A \,\&\, B \vdash A}\;\&X_1 \qquad \frac{}{A \,\&\, B \vdash B}\;\&X_2$$

$$\frac{}{c : \&_{\ell \in L}(\ell : A_\ell) \vdash \mathbf{send}^-\; c\; k(a) :: (a : A_k)}\;\&X$$

It turns out that *dynamically* there is nothing new: the receiving process suspends, and the sending process becomes a message. We repeat the relevant prior rules only to note the different polarities.

$$\begin{array}{lcl} \mathsf{proc}\;(\mathbf{recv}^-\; c\; K) & \mapsto & \mathsf{cont}^-\; c\; K \\ \mathsf{proc}\;(\mathbf{send}^-\; c\; V) & \mapsto & \mathsf{msg}^-\; c\; V \\ \mathsf{cont}^-\; c\; K, \mathsf{msg}^-\; c\; V & \mapsto & \mathsf{proc}\;(V \rhd K) \end{array}$$

$$k(a) \rhd (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} = P_k(a) \quad (\&)$$

The constructs exhibit a remarkable symmetry in SAX, usually associated with classical linear logic [11]. While it is possible to give a message passing interpretation for classical linear logic [28, 5], we stick with the intuitionistic version because of its conceptual and syntactic proximity to functional programming [12]. In particular, it helps to elucidate the connection to futures which have their origin in functional languages.

As an example of negative types, consider a binary counter that can receive a message to increment its value (`inc`) and to return its value (`val`). It maintains local state through a channel x that holds the current value as a binary number. In the case of a value request, we would like to "return" just that number. The

way we can accomplish that is a *forwarding* construct **fwd** c a that forwards messages from a to c. It turns out to be a process assignment for the identity rule of the sequent calculus, which we explain in Section 4.7.

$$\mathsf{ctr} = (\mathtt{inc} : \mathsf{ctr}) \,\&\, (\mathtt{val} : \mathsf{bin})$$

$$\begin{array}{l}
\mathit{counter}\ (c : \mathsf{ctr})\ [x : \mathsf{bin}] = \\
\quad \mathbf{recv}\ c\ (\mathtt{inc}(c') \Rightarrow y \leftarrow \mathbf{call}\ \mathit{succ}\ y\ [x]\ ; \\
\qquad\qquad\qquad\qquad \mathbf{call}\ \mathit{counter}\ c'\ [y] \\
\qquad\quad \mid \mathtt{val}(x') \Rightarrow \mathbf{fwd}\ x'\ x
\end{array}$$

$$\begin{array}{l}
\mathit{init}\ (c : \mathsf{ctr})\ [\,] = \\
\quad z \leftarrow \mathbf{call}\ \mathit{zero}\ z\ [\,]\ ; \\
\quad \mathbf{call}\ \mathit{counter}\ c\ [z]\,)
\end{array}$$

$$\begin{array}{l}
\mathit{two}\ (x : \mathsf{bin})\ [\,] = \\
\quad c_0 \leftarrow \mathbf{call}\ \mathit{init}\ c_0\ [\,]\ ; \\
\quad c_1 \leftarrow \mathbf{send}\ c_0\ \mathtt{inc}(c_1)\ ; \\
\quad c_2 \leftarrow \mathbf{send}\ c_1\ \mathtt{inc}(c_2)\ ; \\
\quad \mathbf{send}\ c_2\ \mathtt{val}(x)
\end{array}$$

Linear Implication $A \multimap B$. Linear implication $A \multimap B$ is the type of a process that receives a channel of type A together with a continuation channel of type B.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap R \qquad \frac{\Gamma, x : A \vdash P :: (y : B)}{\Gamma \vdash \mathbf{recv}^-\ c\ (\langle x, y\rangle \Rightarrow P(x,y)) :: (c : A \multimap B)}\ \multimap R$$

Sending, as for all other constructs, is asynchronous.

$$\frac{}{A, A \multimap B \vdash B}\ \multimap X \qquad \frac{}{a : A, c : A \multimap B \vdash \mathbf{send}^-\ c\ \langle a, b\rangle :: (b : B)}\ \multimap X$$

The dynamics once again does not change. We just recall

$$\langle a, b\rangle \rhd (\langle x, y\rangle \Rightarrow P(x,y)) = P(a,b)$$

As an example, consider a stack with push and pop methods. When the stack is empty, the response to `pop` will be `none` after which the stack process terminates. We don't treat first-class polymorphism here, so we think of stack_A as a family of types indexed by A.

$$\begin{array}{l}
\mathsf{stack}_A = (\mathtt{push} : A \multimap \mathsf{stack}_A) \\
\qquad\qquad \,\&\, (\mathtt{pop} : (\mathtt{some} : A \otimes \mathsf{stack}_A) \oplus (\mathtt{none} : 1))
\end{array}$$

$$\begin{array}{l}
\mathit{empty}\ (s : \mathsf{stack}_A)\ [\,] = \\
\quad \mathbf{recv}\ s\ (\mathtt{push}(s') \Rightarrow \mathbf{recv}\ s'\ (\langle x, s''\rangle \Rightarrow \\
\qquad\qquad\qquad\qquad t \leftarrow \mathbf{call}\ \mathit{empty}\ t\ [\,]\ ;
\end{array}$$

```
                    call elem s'' [x, t] )
  | pop(s') ⇒ u ← send u ⟨⟩ ;
                    send s' none(u) )

elem (s : stack_A) [x : A, t : stack_A] =
  recv s ( push(s') ⇒ recv s' ( ⟨y, s''⟩ ⇒
                    t' ← call elem t' [x, t] ;
                    call elem s'' [y, t'] )
         | pop(s') ⇒ p ← send p ⟨x, t⟩ ;
                    send s' some(p) )

stack10 (s10 : stack_bin) [] =
  n0 ← call zero [] ;
  s0 ← call empty [] ;
  s0' ← send s0 push(s0') ;
  s1 ← send s0' ⟨n0, s1⟩ ;
  n0 ← call zero [] ;        % necssary for linearity
  n1 ← call succ [n0] ;
  s1' ← send s1 push(s1') ;
  send s1' ⟨n1, s10⟩
```

4.7 Identity as Forwarding

The sequent calculus rule of identity essentially equates two channels. The way we define this in our dynamics is for the identity to become a form of continuation, waiting to forward a message on one channel to the other.

$$\frac{}{A \vdash A}\ \mathsf{id} \qquad \frac{}{a : A \vdash \mathbf{fwd}^{\pm}\ c\ a :: (c : A)}\ \mathsf{id}$$

The direction of the messages is prescribed by the polarity of the type, so we split the dynamics into two rules, forwarding message on one channel to another.

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{fwd}^{+}\ c\ a) & \mapsto & \mathsf{cont}^{+}\ a\ c \\ \mathsf{msg}^{+}\ a\ V, \mathsf{cont}^{+}\ a\ c & \mapsto & \mathsf{msg}^{+}\ c\ V \end{array}$$

$$\begin{array}{lcl} \mathsf{proc}\ (\mathbf{fwd}^{-}\ c\ a) & \mapsto & \mathsf{msg}^{-}\ a\ c \\ \mathsf{cont}^{-}\ a\ K, \mathsf{msg}^{-}\ a\ c & \mapsto & \mathsf{cont}^{-}\ c\ K \end{array}$$

This means a channel is another form of extended value or continuation. We write $\hat{V}$ and $\hat{K}$ when we need to include channels as values or continuations, respectively.

5 Preservation and Progress

The recursion-free fragment of SAX satisfies a variant of the cut elimination theorem that guarantees a subformula property [10]. In the presence of recursion, we are more interested in *preservation* and *progress*. These are properties

of *configurations*, so we need to provide typing rules for configurations. Even though configurations are unordered collection of semantic objects, the typing rules impose a partial order where the provider of a channel always precedes its client. We treat the join as an associative operation, with the empty configuration as its unit. Globally, in a configuration, each channel must be provided and used at most once.

It is convenient for the typing of messages and continuations objects to refer to a corresponding process for its typing to avoid a proliferation of typing rules.

$$\frac{}{\Delta \vdash (\cdot) :: \Delta}\ \mathsf{empty} \qquad \frac{\Delta_1 \vdash \mathcal{C}_1 :: \Delta_2 \quad \Delta_2 \vdash \mathcal{C}_2 :: \Delta_3}{\Delta_1 \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_3}\ \mathsf{join}$$

$$\frac{\Gamma \vdash P :: (a : A)}{\Delta, \Gamma \vdash \mathsf{proc}\ P :: (\Delta, a : A)}\ \mathsf{proc}$$

$$\frac{\Gamma \vdash \mathbf{send}^+\ a\ V :: (a : A)}{\Delta, \Gamma \vdash \mathsf{msg}^+\ a\ V :: (\Delta, a : A)}\ \mathsf{msg}^+ \qquad \frac{\Gamma \vdash \mathbf{recv}^+\ a\ K :: (c : C)}{\Delta, \Gamma \vdash \mathsf{cont}^+\ a\ K :: (\Delta, c : C)}\ \mathsf{cont}^+$$

$$\frac{\Gamma \vdash \mathbf{send}^-\ a\ V :: (c : C)}{\Delta, \Gamma \vdash \mathsf{msg}^-\ a\ V :: (\Delta, c : C)}\ \mathsf{msg}^- \qquad \frac{\Gamma \vdash \mathbf{recv}^-\ a\ K :: (a : A)}{\Delta, \Gamma \vdash \mathsf{cont}^-\ a\ K :: (\Delta, a : A)}\ \mathsf{cont}^-$$

$$\frac{}{\Delta, a : A \vdash \mathsf{cont}^+\ a\ c :: (\Delta, c : A)}\ \mathsf{fwd}^+ \qquad \frac{}{\Delta, a : A \vdash \mathsf{msg}^-\ a\ c :: (\Delta, c : A)}\ \mathsf{fwd}^-$$

With this bit of bureaucracy settled, we can now state the preservation theorem. Even though *internally* new channels might be created or closed, *externally* the interface to a configuration remains constant. For reference, the language and its operational semantics can be found in Fig. 1, the typing rules are collected in Fig. 2.

Theorem 1 (Preservation for Linear Message Passing). *If* $\Delta_1 \vdash \mathcal{C} :: \Delta_2$ *and* $\mathcal{C} \mapsto \mathcal{D}$ *then* $\Delta_1 \vdash \mathcal{D} :: \Delta_2$.

Proof. By induction on the typing of a configuration, using inversion on the typing of the semantic objects to observe that the endpoints of each channel perform complementary actions and that the continuation channels once again have matching types.

For the progress theorem, it is convenient to assume that we are executing a closed configuration, providing a finite collection Δ of channels. Such a configuration is *terminal* if all semantic objects are *positive messages* or *negative continuations*.

Theorem 2 (Progress for Linear Message Passing). *If* $\cdot \vdash \mathcal{C} :: \Delta$ *then either* $\mathcal{C} \mapsto \mathcal{D}$ *for some* $\mathcal{D}$*, or* $\mathcal{C}$ *is terminal.*

Proof. We proceed by right-to-left induction over the typing derivation of a configuration, analyzing the rightmost semantic object. We observe that $\mathcal{C} =$

$(\mathcal{C}_1, \phi)$ for a semantic object ϕ can make a transition if $\mathcal{C}_1$ can. So we may assume $\mathcal{C}_1$ is terminal. We distinguish cases based on the shape of ϕ.

(i) $\mathsf{proc}\ P$ can always make a transition.
(ii) $\mathsf{msg}^+\ a\ V$ is terminal, and therefore $\mathcal{C}$ is.
(iii) $\mathsf{cont}^-\ a\ K$ is terminal, and therefore $\mathcal{C}$ is.
(iv) For $\mathsf{msg}^-\ a\ \hat{V}$ there must be a continuation $\mathsf{cont}^-\ a\ K$ in $\mathcal{C}_1$. By inversion on typing, the two can interact.
(v) For $\mathsf{cont}^+\ a\ \hat{K}$ there must be a message $\mathsf{msg}^+\ a\ V$ in $\mathcal{C}_1$. By inversion on typing, the two can interact.

Summary. A summary of the asynchronous linear message passing language using session types can be found in Figs. 1 and 2. Here is a summary of the salient aspects of the language. We show the actions from the provider's perspective; the client will take the matching opposite reaction.

cut	Channel allocation and process spawn
id	Message forwarding
call	Invoking defined process
$\oplus_{\ell \in L}(\ell : A_\ell)$	sending a label with continuation channel
$A \otimes B$	sending a pair of channels
1	sending unit
$\&_{\ell \in L}(\ell : A_\ell)$	receiving and branching on a label with continuation channel
$A \multimap B$	receiving a pair of channels

6 Linear Futures

We stay with the SAX system of logical inference, giving a new interpretation to sequents and proofs. Instead of *channels*, variables now stand for *addresses* of memory cells. A sequent is read as follows:

$$\underbrace{a_1 : A_1, \ldots, a_n : A_n}_{\text{read from}} \vdash P :: \underbrace{(c : C)}_{\text{write to}}$$

Cut allocates a new memory cell a and spawns a process to write to a. As for *futures* [13], every cell has exactly one writer. Because futures are *linear* for now, every cell also has exactly one reader, a discipline sketched by Blelloch and Reid-Miller [3].

6.1 Statics and Dynamics of Futures

In our message passing interpretation, the type of a channel specifies a communication protocol. Here, the type of a cell specifies the shape of its contents. This

Language

$$
\begin{array}{lll}
\text{Types} & A, B, C ::= & A \otimes B \mid 1 \mid \oplus_{\ell \in L} (\ell : A_\ell) \quad \text{(positive)} \\
& \mid & A \multimap B \mid \&_{\ell \in L} (\ell : A_\ell) \quad \text{(negative)} \\
& \mid & t \quad \text{(type names)} \\
\\
\text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \mid \Gamma, a : A \\
\\
\text{Processes} & P, Q & ::= x \leftarrow P(x) \,;\, Q(x) \quad \text{(spawn } P(a)\text{, continue as } Q(a)\text{, } a \text{ fresh)} \\
& \mid & \mathbf{fwd}^{\pm}\ a\ b \quad \text{(forward between } a \text{ and } b\text{)} \\
& \mid & \mathbf{send}^{\pm}\ c\ V \quad \text{(send value } V \text{ on } c\text{)} \\
& \mid & \mathbf{recv}^{\pm}\ c\ K \quad \text{(receive a value on } c \text{ and pass it to } K\text{)} \\
& \mid & \mathbf{call}\ f\ c\ [a_1, \ldots, a_n] \quad \text{(call } f \text{ to provide } c\text{, using } a_1, \ldots, a_n\text{)} \\
\\
\text{Values} & V & ::= \langle a, b \rangle \quad (\otimes, \multimap) \\
& \mid & \langle\,\rangle \quad (1) \\
& \mid & k(a) \quad (\oplus, \&) \\
\\
\text{Continuations} & K & ::= \langle x, y \rangle \Rightarrow P(x, y) \quad (\otimes, \multimap) \\
& \mid & \langle\,\rangle \Rightarrow P \quad (1) \\
& \mid & (\ell(x) \Rightarrow P(x))_{\ell \in L} \quad (\oplus, \&) \\
\\
\text{Signature} & \Sigma & ::= \cdot \\
& \mid & \Sigma, t = A \quad \text{(type definition)} \\
& \mid & \Sigma, (\Gamma \vdash f :: (z : C)) \quad \text{(process declaration)} \\
& \mid & \Sigma, f\ z\ [x_1, \ldots, x_n] = P \quad \text{(process definition)}
\end{array}
$$

Dynamics

$$
\begin{array}{lcll}
\mathsf{proc}\ (x \leftarrow P(x) \,;\, Q(x)) & \mapsto & \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a \text{ fresh}) \\
\mathsf{proc}\ (\mathbf{send}^{\pm}\ c\ V) & \mapsto & \mathsf{msg}^{\pm}\ c\ V & \\
\mathsf{proc}\ (\mathbf{recv}^{\pm}\ c\ K) & \mapsto & \mathsf{cont}^{\pm}\ c\ K & \\
\mathsf{msg}^{\pm}\ c\ V, \mathsf{cont}^{\pm}\ c\ K & \mapsto & \mathsf{proc}\ (V \rhd K) & \\
\mathsf{proc}\ (\mathbf{call}\ f\ c\ [a_1, \ldots, a_n]) & \mapsto & \mathsf{proc}\ (P(c, a_1, \ldots, a_n)) & \\
\multicolumn{4}{r}{\text{for } (f\ z\ [x_1, \ldots, x_n] = P(z, x_1, \ldots, x_n)) \in \Sigma} \\
\\
\mathsf{proc}\ (\mathbf{fwd}^{+}\ c\ a) & \mapsto & \mathsf{cont}^{+}\ a\ c & \\
\mathsf{msg}^{+}\ a\ V, \mathsf{cont}^{+}\ a\ c & \mapsto & \mathsf{msg}^{+}\ c\ V & \\
\\
\mathsf{proc}\ (\mathbf{fwd}^{-}\ c\ a) & \mapsto & \mathsf{msg}^{-}\ a\ c & \\
\mathsf{cont}^{-}\ a\ K, \mathsf{msg}^{-}\ a\ c & \mapsto & \mathsf{cont}^{-}\ c\ K &
\end{array}
$$

Passing a value to a continuation

$$
\begin{array}{lll}
k(a) \rhd (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} & = P_k(a) & (\oplus, \&) \\
\langle a, b \rangle \rhd (\langle x, y \rangle \Rightarrow P(x, y)) & = P(a, b) & (\otimes, \multimap) \\
\langle\,\rangle \rhd (\langle\,\rangle \Rightarrow P) & = P & (1)
\end{array}
$$

Fig. 1. Language for asynchronous message passing

approach leads to the following correspondences. We refer to antecedents in a sequent as "left" and succedents as "right".

	Logic	**Message Passing**	**Shared Memory**
Positive/Right	Axiom	send value V	write value V
Positive/Left	Rule	receive value V	read value V
Negative/Right	Rule	receive value V	write continuation K
Negative/Left	Axiom	send value V	read continuation K

The language of types and values does not change, and continuations only change to the extent that the embedded processes now have a different syntax.

$$\text{Storable}\quad S \quad ::= V \mid K$$

$$\begin{array}{lll}
\text{Processes } P, Q ::= x \leftarrow P(x)\,;\,Q(x) & & \text{(spawn } P(a)\text{, continue as } Q(a)\text{, } a \text{ fresh)} \\
\quad\mid\ \mathbf{move}^{\pm}\ c\ a & & \text{(move storable from } a \text{ to } c\text{)} \\
\quad\mid\ \mathbf{write}^{\pm}\ c\ S & & \text{(write storable } S \text{ to } c\text{)} \\
\quad\mid\ \mathbf{read}^{\pm}\ c\ S & & \text{(read storable from } c \text{ and pass to } S\text{)} \\
\quad\mid\ \mathbf{call}\ f\ c\ [a_1, \ldots, a_n] & & \text{(call } f \text{ with dest. } c\text{, reading } a_1, \ldots, a_n\text{)}
\end{array}$$

Note that defined processes f are always called with a *destination* [27]. Remarkably, we do not need any new typing rules! Instead we define

$$\begin{array}{lcl}
\mathbf{move}^{\pm}\ c\ a & \triangleq & \mathbf{fwd}^{\pm}\ c\ a \\
\mathbf{write}^{+}\ c\ V & \triangleq & \mathbf{send}^{+}\ c\ V \\
\mathbf{read}^{+}\ c\ K & \triangleq & \mathbf{recv}^{+}\ c\ K \\
\mathbf{write}^{-}\ c\ K & \triangleq & \mathbf{recv}^{-}\ c\ K \\
\mathbf{read}^{-}\ c\ V & \triangleq & \mathbf{send}^{-}\ c\ V
\end{array}$$

and the previous set of rules apply!

The dynamics can be similarly derived. Instead of messages and continuations we have *memory cells* $\mathsf{cell}^{\pm}\ c\ S$ and *suspensions* $\mathsf{susp}^{\pm}\ c\ S$. A suspension may block because the corresponding cell may not have been written yet. These can be defined from the message passing dynamics.

$$\begin{array}{lcl}
\mathsf{cell}^{+}\ c\ V & \triangleq & \mathsf{msg}^{+}\ c\ V \\
\mathsf{susp}^{+}\ c\ \hat{K} & \triangleq & \mathsf{cont}^{+}\ c\ \hat{K} \\
\mathsf{cell}^{-}\ c\ K & \triangleq & \mathsf{cont}^{-}\ c\ K \\
\mathsf{susp}^{-}\ c\ \hat{V} & \triangleq & \mathsf{msg}^{-}\ c\ \hat{V}
\end{array}$$

Under the shared memory semantics, forwarding becomes a move from one cell to another—simpler than in the message passing semantics. The correspondences continue to hold if we generalize suspensions to allow the form $\mathsf{susp}^{\pm}\ a\ c$ where a is a channel to read a storable S from, and c is the destination write S to. The

table below visualizes the correspondences.

Shared Memory		Message Passing	
$\mathsf{proc}\ (\mathbf{move}^+\ c\ a)$	$\mapsto\ \mathsf{susp}^+\ a\ c$	$\mathsf{proc}\ (\mathbf{fwd}^+\ c\ a)$	$\mapsto\ \mathsf{cont}^+\ a\ c$
$\mathsf{proc}\ (\mathbf{move}^-\ c\ a)$	$\mapsto\ \mathsf{susp}^-\ a\ c$	$\mathsf{proc}\ (\mathbf{fwd}^-\ c\ a)$	$\mapsto\ \mathsf{msg}^-\ a\ c$
$\mathsf{cell}^+\ a\ V, \mathsf{susp}^+\ a\ c$	$\mapsto\ \mathsf{cell}^+\ c\ V$	$\mathsf{msg}^+\ a\ V, \mathsf{cont}^+\ a\ c$	$\mapsto\ \mathsf{msg}^+\ c\ V$
$\mathsf{cell}^-\ a\ K, \mathsf{susp}^-\ a\ c$	$\mapsto\ \mathsf{cell}^-\ c\ K$	$\mathsf{cont}^-\ a\ K, \mathsf{msg}^-\ a\ c$	$\mapsto\ \mathsf{cont}^-\ c\ K$

Theorem 3 (Bisimulation). *There is a strong bisimulation between the shared memory and the message passing semantics on well-typed processes.*

Proof. Under the correspondences shown above, the steps of the two operational semantics rules correspond exactly, by definition.

Corollaries of this bisimulation are analogues of preservation, terminal configurations, and progress. We say a configuration is *final* if it consists only of objects $\mathsf{cell}^{\pm}\ a\ S$.

Corollary 1 (Preservation and Progress for Linear Futures).

1. *If* $\Delta_1 \vdash \mathcal{C} :: \Delta_2$ *and* $\mathcal{C} \mapsto \mathcal{D}$ *then* $\Delta_1 \vdash \mathcal{D} :: \Delta_2$.
2. *If* $\cdot \vdash \mathcal{C} :: \Delta$ *then either* $\mathcal{C} \mapsto \mathcal{D}$ *for some* $\mathcal{D}$, *or* $\mathcal{C}$ *is final.*

Proof. By the correspondence with the message passing semantics and Theorems 1 and 2.

6.2 Shared Memory Examples

We can transliterate the earlier examples. Here is just one.

$$\mathsf{bin} = (\mathtt{b0} : \mathsf{bin}) \oplus (\mathtt{b1} : \mathsf{bin}) \oplus (\mathtt{e} : 1)$$

$$\begin{array}{l}
\mathit{zero}\ (y : \mathsf{bin})\ [\,] = \\
\quad u \leftarrow \mathbf{write}\ u\ \langle\,\rangle\ ; \\
\quad \mathbf{write}\ y\ \mathtt{e}(u)
\end{array}$$

$$\begin{array}{l}
\mathit{succ}\ (y : \mathsf{bin})\ [x : \mathsf{bin}] = \\
\quad \mathbf{read}\ x\ (\ \mathtt{b0}(x') \Rightarrow \mathbf{write}\ y\ \mathtt{b1}(x') \\
\qquad\quad \mid \mathtt{b1}(x') \Rightarrow y' \leftarrow \mathbf{call}\ \mathit{succ}\ y'\ [x']\ ; \\
\qquad\qquad\qquad\quad \mathbf{write}\ y\ \mathtt{b0}(y') \\
\qquad\quad \mid \mathtt{e}(u) \Rightarrow y' \leftarrow \mathbf{write}\ y'\ \mathtt{e}(u)\ ; \\
\qquad\qquad\qquad\quad \mathbf{write}\ y\ \mathtt{b1}(y')\ )
\end{array}$$

As an example that uses two negative types (external choice and linear implication), we revisit the stack data structure. The *empty* and *elem* processes, for example, *write* a continuation to memory and thereby terminate immediately.

A client *reads* this continuation and passes it either a push or pop label together with a *destination* for the results. In general, all functions and objects are written in destination-passing style [27]. Processes never return a value; instead they are given a destination where to write the result.

$$\begin{aligned}
&\mathsf{stack}_A = (\texttt{push} : A \multimap \mathsf{stack}_A) \\
&\qquad \& \, (\texttt{pop} : (\texttt{some} : A \otimes \mathsf{stack}_A) \oplus (\texttt{none} : 1))
\end{aligned}$$

$$\begin{aligned}
&\mathit{empty}\ (s : \mathsf{stack}_A)\ [\,] = \\
&\quad \textbf{write}\ s\ (\ \texttt{push}(s') \Rightarrow \textbf{write}\ s'\ (\ \langle x, s'' \rangle \Rightarrow \\
&\qquad\qquad\qquad\qquad t \leftarrow \textbf{call}\ \mathit{empty}\ t\ [\,]\ ; \\
&\qquad\qquad\qquad\qquad \textbf{call}\ \mathit{elem}\ s''\ [x, t]\ ) \\
&\qquad\qquad | \ \texttt{pop}(s') \Rightarrow u \leftarrow \textbf{write}\ u\ \langle\,\rangle\ ; \\
&\qquad\qquad\qquad\qquad \textbf{write}\ s'\ \texttt{none}(u)\ )
\end{aligned}$$

$$\begin{aligned}
&\mathit{elem}\ (s : \mathsf{stack}_A)\ [x : A, t : \mathsf{stack}_A] = \\
&\quad \textbf{write}\ s\ (\ \texttt{push}(s') \Rightarrow \textbf{write}\ s'\ (\ \langle y, s'' \rangle \Rightarrow \\
&\qquad\qquad\qquad\qquad t' \leftarrow \textbf{call}\ \mathit{elem}\ t'\ [x, t]\ ; \\
&\qquad\qquad\qquad\qquad \textbf{call}\ \mathit{elem}\ s''\ [y, t']\ ) \\
&\qquad\qquad | \ \texttt{pop}(s') \Rightarrow p \leftarrow \textbf{write}\ p\ \langle x, t \rangle\ ; \\
&\qquad\qquad\qquad\qquad \textbf{write}\ s'\ \texttt{some}(p)\ )
\end{aligned}$$

$$\begin{aligned}
&\mathit{stack10}\ (s_{10} : \mathsf{stack}_{\mathsf{bin}})\ [\,] = \\
&\quad n_0 \leftarrow \textbf{call}\ \mathit{zero}\ [\,]\ ; \\
&\quad s_0 \leftarrow \textbf{call}\ \mathit{empty}\ [\,]\ ; \\
&\quad s_0' \leftarrow \textbf{read}\ s_0\ \texttt{push}(s_0')\ ; \\
&\quad s_1 \leftarrow \textbf{read}\ s_0'\ \langle n_0, s_1 \rangle\ ; \\
&\quad n_0 \leftarrow \textbf{call}\ \mathit{zero}\ [\,]\ ; \\
&\quad n_1 \leftarrow \textbf{call}\ \mathit{succ}\ [n_0]\ ; \\
&\quad s_1' \leftarrow \textbf{read}\ s_1\ \texttt{push}(s_1')\ ; \\
&\quad \textbf{read}\ s_1'\ \langle n_1, s_{10} \rangle
\end{aligned}$$

This program highlights that there is a rather immediate *sequential* interpretation of parallel composition $x \leftarrow P(x)\ ;\ Q(x)$. As usual, we allocate a fresh memory cell a for x, but rather than executing $P(a)$ and $Q(a)$ in parallel, we first complete the execution of $P(a)$ (which will write to cell a), and then proceed with $Q(a)$. This corresponds to an eager (by-value) strategy. We can also pursue a lazy (by-need) strategy: postpone computation of $P(a)$ and start with $Q(a)$. When $Q(a)$ attempts to read from a, $P(a)$ is awakened and will run to completion (writing to a), after which $Q(a)$ continues by reading from a. This embodies call-by-need and not call-by-name because other readers of a now directly access the value stored in the cell.

Such simple sequential interpretations of computations are not immediately available in the message passing setting, but are quite clear here. In particular, in our language all memory allocation is for futures. In a more realistic language we would have both parallel composition, and maybe two forms of sequential composition: one eager and one lazy.

7 From Linear to Nonlinear Futures

So far all constructs, whether message passing or shared memory, have been strictly linear. It is easy to imagine how we can take the shared memoryx interpretation and make it nonlinear. We add two rules, one for weakening and one for contraction.

$$\frac{\Gamma \vdash P :: (c : C)}{\Gamma, a : A \vdash P :: (c : C)}\ \mathsf{weaken} \qquad \frac{\Gamma, a : A, a : A \vdash P :: (c : C)}{\Gamma, a : A \vdash P :: (c : C)}\ \mathsf{contract}$$

At this point an object $\mathsf{cell}\ a\ S$ may have multiple readers. This means when it is read, it cannot be immediately deallocated but has to be left for eventual garbage collection. Therefore memory cells in the operational semantics are now *persistent*. In multiset rewriting we indicate this by prefixing a semantic object with an exclamation mark "!" (the exponential of linear logic). Such objects, when matched on the left of a rule are carried over implicitly and remain in the configuration. We call the others *ephemeral*.

In our semantics, now formulated using the syntax of shared memory, cells are persistent and processes as well as suspensions remain linear. That must be the case so that they can change state. A "persistent process" $!\mathsf{proc}\ P$ could transition over and over again and, for example, allocate an unbounded amount of memory without ever making progress. Parallel composition (cut) and **call** (definitions) remain unchanged.

$$\begin{array}{lcl}
\mathsf{proc}\ (\mathbf{write}^{\pm}\ c\ S) & \mapsto & !\mathsf{cell}^{\pm}\ c\ S \\
\mathsf{proc}\ (\mathbf{read}^{\pm}\ c\ S) & \mapsto & \mathsf{susp}^{\pm}\ c\ S \\
!\mathsf{cell}^{+}\ a\ V, \mathsf{susp}^{+}\ a\ K & \mapsto & \mathsf{proc}\ (V \rhd K) \\
!\mathsf{cell}^{-}\ a\ K, \mathsf{susp}^{-}\ a\ V & \mapsto & \mathsf{proc}\ (V \rhd K) \\
\mathsf{proc}\ (\mathbf{move}^{\pm}\ c\ a) & \mapsto & \mathsf{susp}^{\pm}\ a\ c \\
!\mathsf{cell}^{\pm}\ a\ S, \mathsf{susp}^{\pm}\ a\ c & \mapsto & !\mathsf{cell}^{\pm}\ c\ S
\end{array}$$

The **move** process now *copies* from one cell to another. We postpone the metatheory of the nonlinear version of future to Cor. 2.

Now we consider a binary trie as a data structure for maintaining sets of binary numbers (and other data that can be interpreted in this form). We take the liberty of writing an underscore (_) for an anonymous variable and combining consecutive pattern matches and consecutive writes. The interface to this data structure would construct empty and singleton tries, as well as union, intersection and difference. We show only empty, singleton, and difference.

First, the straightforward setup of the booleans with the operation of $b \wedge \neg c$. If we were to show the definitions of union and intersection we would also need conjunction and disjunction.

$\mathsf{bool} = (\mathtt{true} : 1) \oplus (\mathtt{false} : 1)$

true $(b : \mathbf{bool})\ [\,] = u \leftarrow \mathbf{write}\ u\ \langle\,\rangle\ ;\ \mathbf{write}\ b\ \mathtt{true}(u)$
false $(b : \mathbf{bool})\ [\,] = u \leftarrow \mathbf{write}\ u\ \langle\,\rangle\ ;\ \mathbf{write}\ b\ \mathtt{false}(u)$

```
andnot (d : bool) [b : bool, c : bool] =
  read b ( true(_) ⇒ read c ( true(_) ⇒ call false d [ ]
                            | false(_) ⇒ call true d [ ] )
         | false(_) ⇒ call false d [ ] )
```

We reuse the binary numbers and define tries as being either a leaf or a node containing three addresses: the left subtrie selected for the bit 0, the boolean b which is true if the sequence of bits which led to this node is in the trie, and the right subtrie selected for the bit 1.

The process *empty* constructs a leaf (the empty trie), while *singleton* traverses a binary number, constructing a trie with exactly one node marked true.

```
trie = (leaf : 1) ⊕ (node : trie ⊗ bool ⊗ trie)

empty (r : trie) [ ] =
  u ← write u ⟨ ⟩ ; write r leaf(u)

singleton (r : trie) [x : bin] =
  read x ( b0(x′) ⇒ r0 ← call singleton r0 [x′] ;
                    b ← call false [ ] ;
                    r1 ← call empty [ ] ;
                    write r node⟨r0, b, r1⟩
         | b1(x′) ⇒ r0 ← call empty [ ] ;
                    b ← call false [ ] ;
                    r1 ← call singleton r1 [x′] ;
                    write r node⟨r0, b, r1⟩
         | e(_) ⇒ r′ ← call empty r′ [ ] ;
                  b ← call true b [ ] ;
                  write r node⟨r′, b, r′⟩ )
```

Finally, the *diff* process traverses the two tries in parallel, short-circuiting if one is a leaf. If not, it applies the *andnot* operation to decide if the resulting node should be true. While *singleton* can easily be made linear, this would take significant effort here. For example, when s is empty, t is ignored entirely. The *remove* process just computes the difference with a singleton.

```
diff (r : trie) [s : trie, t : trie] =
  read s ( leaf⟨ ⟩ ⇒ call empty r [ ]
         | node⟨s0, b, s1⟩ ⇒
  read t ( leaf⟨ ⟩ ⇒ move r s
         | node⟨t0, c, t1⟩ ⇒
         r0 ← call diff r0 [s0, t0] ;
         d ← call andnot d [b, c] ;
         r1 ← call diff r1 [s1, t1] ;
         write r node⟨r0, d, r1⟩ ))

remove (r : trie) [s : trie, x : bin] =
  t ← call singleton t [x] ;
  call diff r [s, t]
```

8 Backporting Persistence to Message Passing

Persistent cells are quite easy to understand from the shared memory perspective. Now we can use our correspondences in the opposite direction to obtain a bisimilar version of message passing in which certain messages and suspensions are persistent! The language of programs itself does not change, but as defined above the client can use weakening and contraction on channels it uses.

A positive message, flowing from the provider to the client, may then have multiple recipients. We therefore make such messages *persistent* in the dynamic rules. The recipient of such a message only reacts once, so it will not be persistent. Conversely, a negative suspension may be waiting for messages from multiple clients and therefore should be *persistent*, but each such message should be processed only once.

$$
\begin{array}{lcl}
!\mathsf{cell}^+\ c\ V & \triangleq & !\mathsf{msg}^+\ c\ V \\
\mathsf{susp}^+\ c\ K & \triangleq & \mathsf{cont}^+\ c\ K \\
!\mathsf{cell}^-\ c\ K & \triangleq & !\mathsf{cont}^-\ c\ K \\
\mathsf{susp}^-\ c\ V & \triangleq & \mathsf{msg}^-\ c\ V \\
\\
\mathsf{susp}^+\ a\ c & \triangleq & \mathsf{cont}^+\ c\ a \\
\mathsf{susp}^-\ a\ c & \triangleq & \mathsf{msg}^-\ c\ a
\end{array}
$$

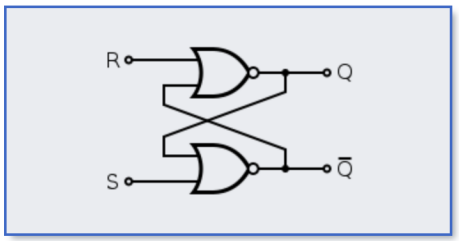
8.1 Examples

As example we start with *nor* which takes two bits x and y and produces the negation of the disjunction of x and y on the output channel z.

$$\mathsf{bit} = (\mathtt{b0} : 1) \oplus (\mathtt{b1} : 1)$$

$$
\begin{array}{l}
\mathit{nor}\ (z : \mathsf{bit})\ [x : \mathsf{bit}, y : \mathsf{bit}] = \\
\quad \mathbf{recv}\ x\ (\ \mathtt{b0}(\_) \Rightarrow \mathbf{recv}\ y\ (\ \mathtt{b0}(u) \Rightarrow \mathbf{send}\ z\ \mathtt{b1}(u) \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{b1}(u) \Rightarrow \mathbf{send}\ z\ \mathtt{b0}(u)\ ) \\
\qquad\quad \mid \mathtt{b1}(\_) \Rightarrow \mathbf{recv}\ y\ (\ \mathtt{b0}(u) \Rightarrow \mathbf{send}\ z\ \mathtt{b0}(u) \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{b1}(u) \Rightarrow \mathbf{send}\ z\ \mathtt{b0}(u)\ ))
\end{array}
$$

We now use this in the construction of a latch which uses a feedback loop and recursion.



In the code below the stream of pairs of signals R and S is represented by channel in : bits2 and the pair of signals Q and $\overline{Q}$ is represented by out : bits2. The initial (and in later calls, previous) value of Q and $\overline{Q}$ is provided on the channels q and $\overline{q}$. We have combined two consecutive receives and sends for readability.

$$\mathsf{bits2} = (\mathsf{bit} \otimes \mathsf{bit}) \otimes \mathsf{bits2}$$

$$\begin{array}{l} \mathit{latch}\ (\mathit{out} : \mathsf{bits2})\ [q : \mathsf{bit}, \overline{q} : \mathsf{bit}, \mathit{in} : \mathsf{bits2}] = \\ \quad \mathbf{recv}\ \mathit{in}\ (\langle\langle r, s\rangle, \mathit{in}'\rangle \Rightarrow \\ \qquad q' \leftarrow \mathbf{call}\ \mathit{nor}\ q'\ [r, \overline{q}]\ ; \\ \qquad \overline{q}' \leftarrow \mathbf{call}\ \mathit{nor}\ \overline{q}'\ [s, q]\ ; \\ \qquad \mathit{out}' \leftarrow \mathbf{call}\ \mathit{latch}\ \mathit{out}'\ [q', \overline{q}', \mathit{in}']\ ; \\ \qquad \mathbf{send}\ \mathit{out}\ \langle\langle q', \overline{q}'\rangle, \mathit{out}'\rangle\ ) \end{array}$$

8.2 Metatheory

The metatheory for nonlinear message passing changes systematically from the linear case, reflecting persistence of positive messages and negative suspensions. Instead of splitting the context to check the processes, messages, and continuations embedded in them, we pass all channels in all configuration typing rules.

$$\frac{}{\Delta \vdash (\cdot) :: \Delta}\ \mathsf{empty} \qquad \frac{\Delta_1 \vdash \mathcal{C}_1 :: \Delta_2 \quad \Delta_2 \vdash \mathcal{C}_2 :: \Delta_3}{\Delta_1 \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_3}\ \mathsf{join}$$

$$\frac{\Delta \vdash P :: (a : A)}{\Delta \vdash \mathsf{proc}\ P :: (\Delta, a : A)}\ \mathsf{proc}$$

$$\frac{\Delta \vdash \mathbf{send}^+\ a\ V :: (a : A)}{\Delta \vdash \mathsf{msg}^+\ a\ V :: (\Delta, a : A)}\ \mathsf{msg}^+ \qquad \frac{\Delta \vdash \mathbf{recv}^+\ a\ K :: (c : C)}{\Delta \vdash \mathsf{cont}^+\ a\ K :: (\Delta, c : C)}\ \mathsf{cont}^+$$

$$\frac{\Delta \vdash \mathbf{send}^-\ a\ V :: (c : C)}{\Delta \vdash \mathsf{msg}^-\ a\ V :: (\Delta, c : C)}\ \mathsf{msg}^- \qquad \frac{\Delta \vdash \mathbf{recv}^-\ a\ K :: (a : A)}{\Delta \vdash \mathsf{cont}^-\ a\ K :: (\Delta, a : A)}\ \mathsf{cont}^-$$

$$\frac{a : A \in \Delta}{\Delta \vdash \mathsf{cont}^+\ a\ c :: (\Delta, c : A)}\ \mathsf{fwd}^+ \qquad \frac{a : A \in \Delta}{\Delta \vdash \mathsf{msg}^-\ a\ c :: (\Delta, c : A)}\ \mathsf{fwd}^-$$

In the statement of preservation we now have to account for a freshly allocated channel to become visible at the external interface to the configuration.

Theorem 4 (Preservation for Nonlinear Message Passing). *If $\Delta_1 \vdash \mathcal{C} :: \Delta_2$ and $\mathcal{C} \mapsto \mathcal{D}$ then $\Delta_1 \vdash \mathcal{D} :: \Delta_2'$ for some $\Delta_2' \supseteq \Delta_2$.*

Proof. By induction on the typing of a configuration as before. In the case the step is a spawn which allocates a fresh channel $a : A$, we have $\Delta_2' = (\Delta_2, a : A)$.

Recall that a configuration was defined to be *terminal* if all semantics objects are *positive messages* or *negative continuations*. These objects are precisely those that become persistent, so terminal configurations now consist entirely of persistent objects.

Theorem 5 (Progress for Nonlinear Message Passing). *If $\cdot \vdash \mathcal{C} :: \Delta$ then either $\mathcal{C} \mapsto \mathcal{D}$ for some $\mathcal{D}$, or $\mathcal{C}$ is terminal.*

Proof. As before, by right-to-left induction over the typing derivation of the given configuration.

Now we can transport this result to nonlinear futures as before.

Corollary 2 (Preservation and Progress for Nonlinear Futures).

1. *If* $\Delta_1 \vdash \mathcal{C} :: \Delta_2$ *and* $\mathcal{C} \mapsto \mathcal{D}$ *then* $\Delta_1 \vdash \mathcal{D} :: \Delta_2'$ *for some* $\Delta_2' \supseteq \Delta_2$.
2. *If* $\cdot \vdash \mathcal{C} :: \Delta$ *then either* $\mathcal{C} \mapsto \mathcal{D}$ *for some* $\mathcal{D}$, *or* $\mathcal{C}$ *is final.*

Proof. By the correspondence with the message passing semantics and Theorems 4 and 5.

9 Conclusion

We have taken the journey from linear asynchronous message passing through linear futures and nonlinear futures back to nonlinear asynchronous message passing. In each layer, the operational semantics of message passing and futures are (strongly) bisimilar. This tight relationship is possible because all formalisms are based on the semi-axiomatic sequent calculus. The two kinds of interpretations have different characteristics: message passing exchanges only small messages ($\langle\,\rangle$, $\langle a, b\rangle$, and $k(a)$ for channels a, b, and labels k), while futures allow two natural sequential interpretations (eager and lazy) in addition to the parallel one.

We have not discussed type checking for the languages here, but standard techniques, including input/output contexts [6] apply. We can also use standard translations from natural deduction to sequent calculi to map a more familiar functional syntax to either message passing or futures (see, for example, [26]).

An alternative operational semantics for the language with weakening and contraction tracks multiple clients precisely, which can then be deallocated eagerly, avoiding the need for a general garbage collector [22]. This dynamics is significantly more complex than the model we have presented here, so we have not yet attempted to relate message passing and futures when both use explicit deallocation.

We can easily extend our bisimulation further by following the blueprint of mixed linear/nonlinear logic [2] and its generalization in adjoint logic [24, 21]. In brief, we can extend the type systems of this paper by introducing multiple *modes* of types, potentially with different structural properties (e.g, linear/nonlinear, or message passing/futures), and then combine them using adjoint pairs of modalities. We have already investigated adjoint types separately for message passing [22] and futures [23]. These prior formulations are incompatible with each other, and the present paper recasts them into a single unifying framework of SAX.

Acknowledgments. We would like to thank Henry DeYoung, Luiz de Sa, and Siva Somayyajula for helpful discussions regarding the subject of this paper and comments on an earlier draft.

Judgmental Rules

$$\frac{\Gamma_1 \vdash P(x) :: (x : A) \quad \Gamma_2, x : A \vdash Q(x) :: (c : C)}{\Gamma_1, \Gamma_2 \vdash (x \leftarrow P(x)\ ;\ Q(x)) :: (c : C)}\ \mathsf{cut} \qquad \frac{}{a : A \vdash \mathbf{fwd}\ \mathbf{c}\ \mathbf{a} :: (\mathbf{c} : \mathbf{A})}\ \mathsf{id}$$

Positives

$$\frac{(k \in L)}{b : A_k \vdash \mathbf{send}\ a\ k(b) :: (a : \oplus_{\ell \in L}(\ell : A_\ell))}\ \oplus X$$

$$\frac{\Gamma, x : A_\ell \vdash P_\ell(x) :: (d : C) \quad (\forall \ell \in L)}{\Gamma, c : \oplus_{\ell \in L}(\ell : A_\ell) \vdash \mathbf{recv}\ c\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} :: (d : C)}\ \oplus L$$

$$\frac{}{a : A, b : B \vdash \mathbf{send}\ c\ \langle a, b \rangle :: (c : A \otimes B)}\ \otimes X$$

$$\frac{\Gamma, x : A, y : B \vdash P(x, y) :: (d : C)}{\Gamma, c : A \otimes B \vdash \mathbf{recv}\ c\ (\langle x, y \rangle \Rightarrow P(x, y)) :: (d : C)}\ \otimes L$$

$$\frac{}{\cdot \vdash \mathbf{send}\ c\ \langle\rangle :: (c : 1)}\ \otimes X \qquad \frac{\Gamma \vdash P :: (d : C)}{\Gamma, c : 1 \vdash \mathbf{recv}\ c\ (\langle\rangle \Rightarrow P) :: (d : C)}\ 1L$$

Negatives

$$\frac{\Gamma \vdash P_\ell(x) :: (x : A_\ell) \quad (\forall \ell \in L)}{\Gamma \vdash \mathbf{recv}\ c\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} :: (c : \&_{\ell \in L}(\ell : A_\ell))}\ \&R$$

$$\frac{}{c : \&_{\ell \in L}(\ell : A_\ell) \vdash \mathbf{send}\ c\ k(a) :: (a : A_k)}\ \&X$$

$$\frac{\Gamma, x : A \vdash P :: (y : B)}{\Gamma \vdash \mathbf{recv}\ c\ (\langle x, y \rangle \Rightarrow P(x, y)) :: (c : A \multimap B)}\ \multimap R$$

$$\frac{}{a : A, c : A \multimap B \vdash \mathbf{send}\ c\ \langle a, b \rangle :: (b : B)}\ \multimap X$$

Definitions

$$\frac{(x_1 : A_1, \ldots, x_n : A_n \vdash f :: (z : C)) \in \Sigma}{a_1 : A_1, \ldots, a_n : A_n \vdash f\ c\ [a_1, \ldots, a_n] :: (c : C)}\ \mathsf{call}$$

Fig. 2. Typing for Message Passing

References

1. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3), 197–347 (1992)
2. Benton, N.: A mixed linear and non-linear logic: Proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94). pp. 121–135. Springer LNCS 933, Kazimierz, Poland (Sep 1994), an extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge
3. Blelloch, G.E., Reid-Miller, M.: Pipeling with futures. Theory of Computing Systems **32**, 213–239 (1999)
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010). pp. 222–236. Springer LNCS 6269, Paris, France (Aug 2010)
5. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016), special Issue on Behavioural Types
6. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theoretical Computer Science **232**(1–2), 133–163 (Feb 2000), special issue on Proof Search in Type-Theoretic Languages
7. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation **207**(10), 1044–1077 (Oct 2009)
8. Curry, H.B.: Functionality in combinatory logic. Proceedings of the National Academy of Sciences, U.S.A. **20**, 584–590 (1934)
9. DeYoung, H., Pfenning, F.: Data layout from a type-theoretic perspective. In: Proceedings of the 38th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2022). Electronic Notes in Theoretical Informatics and Computer Science, vol. 1 (2022), https://arxiv.org/abs/2212.06321v6
10. DeYoung, H., Pfenning, F., Pruiksma, K.: Semi-axiomatic sequent calculus. In: Ariola, Z. (ed.) 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020). pp. 29:1–29:22. LIPIcs 167, Paris, France (Jun 2020)
11. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**, 1–102 (1987)
12. Girard, J.Y., Lafont, Y.: Linear logic and lazy computation. In: Ehrig, H., Kowalski, R., Levi, G., Montanari, U. (eds.) Proceedings of the International Joint Conference on Theory and Practice of Software Development. vol. 2, pp. 52–66. Springer-Verlag LNCS 250, Pisa, Italy (Mar 1987)
13. Halstead, R.H.: Multilisp: A language for parallel symbolic computation. ACM Transactions on Programming Languages and Systems **7**(4), 501–539 (Oct 1985)
14. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) 4th International Conference on Concurrency Theory (CONCUR 1993). pp. 509–523. Springer LNCS 715 (1993)
15. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91). pp. 133–147. Springer-Verlag LNCS 512, Geneva, Switzerland (Jul 1991)
16. Howard, W.A.: The formulae-as-types notion of construction (1969), unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980)
17. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. In: Boehm, H.J., Steele, G. (eds.) Proceedings of the 23rd Symposium on Principles

of Programming Languages (POPL'96). pp. 358–371. ACM, St. Petersburg Beach, Florida, USA (Jan 1996)
18. Laurent, O.: Syntax vs. semantics: A polarized approach. Theoretical Computer Science **343**(1–2), 177–206 (2005)
19. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)
20. OpenMP, http://openmp.org
21. Pruiksma, K., Chargin, W., Pfenning, F., Reed, J.: Adjoint logic (Apr 2018), http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf, unpublished manuscript
22. Pruiksma, K., Pfenning, F.: A message-passing interpretation of adjoint logic. Journal of Logical and Algebraic Methods in Programming **120**(100637) (2021)
23. Pruiksma, K., Pfenning, F.: Back to futures. Journal of Functional Programming **32**, e6 (2022)
24. Reed, J.: A judgmental deconstruction of modal logic (May 2009), http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf, unpublished manuscript
25. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
26. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. In: Birkedal, L. (ed.) 15th International Conference on Foundations of Software Science and Computation Structures. pp. 346–360. FoSSaCS'12, Springer LNCS, Tallinn, Estonia (Mar 2012)
27. Wadler, P.: Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In: Conference on Lisp and Functional Programming (LFP 1984). pp. 45–52. ACM, Austin, Texas (Aug 1984)
28. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th International Conference on Functional Programming (ICFP 2012). pp. 273–286. ACM Press, Copenhagen, Denmark (Sep 2012)