JFP, 33 pages, 2021. © Cambridge University Press 2021
doi:10.1017/xxxxx

Back to Futures

KLAAS PRUIKSMA

Computer Science Department
Carnegie Mellon University
(e-mail: kpruiksm@andrew.cmu.edu)

FRANK PFENNING

Computer Science Department
Carnegie Mellon University
(e-mail: fp@cs.cmu.edu)

Abstract

Common approaches to concurrent programming begin with languages whose semantics are naturally sequential and add new constructs that provide limited access to concurrency, as exemplified by *futures*. This approach has been quite successful, but often does not provide a satisfactory theoretical backing for the concurrency constructs, and it can be difficult to give a good semantics that allows a programmer to use more than one of these constructs at a time.

We take a different approach, starting with a concurrent language based on a Curry-Howard interpretation of adjoint logic, to which we add three atomic primitives that allow us to encode sequential composition and various forms of synchronization. The resulting language is highly expressive, allowing us to encode futures, fork/join parallelism, and monadic concurrency in the same framework. Notably, since our language is based on adjoint logic, we are able to give a formal account of *linear futures*, which have been used in complexity analysis by Blelloch and Reid-Miller. The uniformity of this approach means that we can similarly work with many of the other concurrency primitives in a linear fashion, and that we can mix several of these forms of concurrency in the same program to serve different purposes.

Key Words: Concurrency, substructural type systems, futures

1 Introduction

Concurrency has been a very useful tool in increasing performance of computations and in enabling distributed computation, and consequently, there are a wide variety of different approaches to programming languages for concurrency. A common pattern is to begin with a sequential language and add some form of concurrency primitive, ranging from threading libraries such as pthreads to monads to encapsulate concurrent computation, as in SILL (Toninho *et al.*, 2013; Toninho, 2015; Griffith, 2016), to futures (Halstead, 1985). Many of these approaches have seen great practical success, and yet from a theoretical perspective, they are often unsatisfying, with the concurrent portion of the language being attached to the sequential base language in a somewhat ad hoc manner, rather than having a coherent theoretical backing for the language as a whole.

In order to give a more uniform approach to concurrency, we take the opposite approach and begin with a language, Seax, whose semantics are naturally concurrent. With a minor addition to Seax, we are able to force synchronization, allowing us to encode sequentiality. In the resulting language, we can model many different concurrency primitives, including futures, fork/join, and concurrency monads. Moreover, as all of these constructs are encoded in the same language, we can freely work with any mixture and retain the same underlying semantics and theoretical underpinnings.

Two lines of prior research meet in the development of Seax. The first involves a new presentation of intuitionistic logic, called the *semi-axiomatic sequent calculus* (SAX) (DeYoung *et al.*, 2020), which combines features from Hilbert's axiomatic form (Hilbert & Bernays, 1934) and Gentzen's sequent calculus (Gentzen, 1935). Cut reduction in the semi-axiomatic sequent calculus can be put into correspondence with asynchronous communication, either via message passing (Pruiksma & Pfenning, 2019) or via shared memory (DeYoung *et al.*, 2020). We build on the latter, extending it in three major ways to get Seax. First, we extend from intuitionistic logic to a semi-axiomatic presentation of adjoint logic (Reed, 2009; Licata & Shulman, 2016; Licata *et al.*, 2017; Pruiksma & Pfenning, 2019), the second major line of research leading to Seax. This gives us a richer set of connectives as well as the ability to work with linear and other substructural types. Second, we add equirecursive types and recursively defined processes, allowing for a broader range of programs, at the expense of termination, as usual. Third, we add three new *atomic write* constructs that write a value and its tag in one step. This minor addition enables us to encode both some forms of synchronization and sequential composition of processes, despite the naturally concurrent semantics of Seax.

This resulting language is highly expressive. Using these features, we are able to model functional programming with a semantics in destination-passing style that makes memory explicit (Wadler, 1984; Larus, 1989; Cervesato *et al.*, 2002; Simmons, 2012), allowing us to write programs in more familiar functional syntax which can then be expanded into Seax. We can also encode various forms of concurrency primitives, such as fork/join parallelism (Conway, 1963) implemented by parallel pairs, futures (Halstead, 1985), and a concurrency monad in the style of SILL (Toninho *et al.*, 2013; Toninho, 2015; Griffith, 2016) (which combines sequential functional with concurrent session-typed programming). As an almost immediate consequence of our reconstruction of futures, we obtain a clean and principled subsystem of *linear futures*, already anticipated and used in parallel complexity analysis by Blelloch and Reid-Miller (Blelloch & Reid-Miller, 1999) without being rigorously developed.

Our use of adjoint logic as a base for Seax is not essential to most of the programming constructs we describe — only the concurrency monad makes use of the adjoint nature of the language in a fundamental way. However, it allows for a few useful features of Seax. The uniform nature of adjoint logic means that we can move easily from our initial discussion of futures to their linear form or to a language with both linear and non-linear futures (and, for that matter, the other constructs can also be made linear or affine or strict). Moreover, we can take advantage of the adjoint nature of Seax to combine multiple language features while maintaining some degree of isolation between them. We could, for instance, have a language where one portion is purely sequential, another adds concurrency via fork/join, and yet another adds concurrency via futures. While it is already possible to add various features to a base language in an ad hoc way (as is often done in real programming languages), the fact that these features can be encoded in Seax means that the semantics are uniform — there is no need to add extra rules to handle the new constructs. Moreover, because we are able to separate the different features syntactically by keeping them at different modes, an implementation of this language could optimize differently at each mode. A purely sequential language needs only one thread of computation, and can avoid dealing with locking on memory entirely, for instance.

The overall benefits of the adjoint approach, then, are twofold — first, since Seax is expressive enough to encode varied language features, we can combine these different features or constructs in a uniform fashion, and second, since we can keep different portions of the language (containing different features) separated, we gain all the benefits of a more restrictive language, at least locally. In each individual portion of the language, we can reason (and therefore also optimize) based on the restrictions on that part of the language, although as the restrictions on different parts of the language may vary, so too will the extra information we gain from those restrictions. Because of this, rather than looking at languages as a whole, we will focus on how individual language features can be encoded in Seax. Such features can then be combined into a single language in order to use more than one at a time.

The principal contributions of this paper are:

1. the language Seax, which has a concurrent write-once shared-memory semantics for programs based on a computational interpretation of adjoint logic;
2. a model of sequential computation using an extension of this semantics with limited atomic writes;
3. a reconstruction of fork/join parallelism;
4. a reconstruction of futures, including a rigorous definition of linear futures;
5. a reconstruction of a concurrency monad which combines functional programming with session-typed concurrency as an instance of the adjoint framework;
6. the uniform nature of these reconstructions, which allows us to work with any of these concurrency primitives and more all within the same language;
7. the ability to keep different portions of the language at different modes, enabling us to restrict part of the language for implementation or reasoning, while retaining the full-featured nature of the rest of the language.

We begin by introducing the type system and syntax for Seax, along with some background on adjoint logic (Section 2), followed by its semantics, which are naturally concurrent (Section 3). At this point, we are able to look at some examples of programs in Seax. Next, we make the critical addition of sequentiality (Section 4), examining both what changes we need to make to Seax to encode sequentiality and how we go about that encoding. Using our encoding of sequentiality, we can build a reconstruction of a standard functional language's lambda terms (Section 5), which both serves as a simple example of a reconstruction and illustrates that we need not restrict ourselves to the relatively low-level syntax of Seax when writing programs. Following this, we examine and reconstruct several concurrency primitives, beginning with futures (Section 6), before moving on to parallel pairs (an implementation of fork/join, in Section 7) and a concurrency monad that borrows heavily from SILL (Section 8). We conclude with a brief discussion of our results and future work.

2 Seax: Types and Syntax

The type system and language we present here, which we will use throughout this paper, are based on adjoint logic (Reed, 2009; Licata & Shulman, 2016; Licata *et al.*, 2017; Pruiksma & Pfenning, 2019, 2020) starting with a Curry-Howard interpretation, which we then leave behind by adding recursion, allowing a richer collection of programs. Most of the details of adjoint logic are not relevant here, and so we provide a brief overview of those that are, focusing on how they relate to our language.

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 (which forms a basis for our semantics) 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 Γ satisfies $m \geq k$. We express this concisely as $\Gamma \geq k$.

We can go back and forth between the layers using *shifts* $\uparrow_k^m A_k$ (up from k to m requiring $m \geq k$) and $\downarrow_m^{\mathsf{R}} A_r$ (down from r to m requiring $r \geq m$). A given pair $\uparrow_k^m$ and $\downarrow_k^m$ forms an adjunction,[1] justifying the name adjoint logic.

Now, our types are the propositions of adjoint logic, augmented with general equirecursive types formed via mutually recursive type definitions in a global signature — most of the basic types are familiar as propositions from intuitionistic linear logic (Girard & Lafont, 1987), or as session types (Honda, 1993; Honda *et al.*, 1998; Gay & Vasconcelos, 2010), tagged with subscripts for modes. The only new types are the shifts $\downarrow_m^{\mathsf{R}} A_r$ and $\uparrow_k^m A_k$. We do, however, change $\oplus$ and $\&$ slightly, using an n-ary form rather than the standard binary form. These are, of course, equivalent, but the n-ary form allows for more natural programming. The grammar of types (as well as processes) can be found in Fig. 1. Note that while our grammar includes mode subscripts on types, type constructors, and variables we will often omit them when they are clear from context.

The typing judgment for processes has the form

$$x_1 : A_{m_1}^1, \ldots, x_n : A_{m_n}^n \vdash P :: (x : A_k)$$

where P is a process expression and we require that each $m_i \geq k$. Given such a judgment, we say that P *provides* or *writes* x, and *uses* or *reads* $x_1, \ldots, x_n$. We may often write a superscript on the variables to indicate whether they are being used for writing or reading. For instance, we would write x^{W} in P to denote that P writes to x, and x_1^{R} to denote that P reads from x. While this information is not necessary for the semantics (and therefore we do not make it part of the formal syntax), it is convenient when writing down example

[1] See, for instance, Licata *et al.* (2017) for a more categorically-focused discussion of a slightly different form of adjoint logic, or Benton (1994) for a discussion of the adjunction in a limited case.

Types	A_m, B_m	$::=$	$\oplus_m\{\ell : A_m^\ell\}_{\ell\in L}$	internal choice
		$\mid$	$A_m \otimes_m B_m$	multiplicative conjunction
		$\mid$	$\mathbf{1}_m$	multiplicative unit
		$\mid$	$\&_m\{\ell : A_m^\ell\}_{\ell\in L}$	external choice
		$\mid$	$A_m \multimap_m B_m$	linear implication
		$\mid$	$\downarrow_m^{\mathsf{R}} A_r$	down shift, $r \geq m$
		$\mid$	$\uparrow_k^m A_k$	up shift, $m \geq k$
		$\mid$	t	type variables
Processes	P, Q	$::=$	$x_m \leftarrow P \,;\, Q$	cut: allocate x, spawn P, continue as Q
		$\mid$	$x_m \leftarrow y_m$	id: copy or move from y to x
		$\mid$	$x_m.V$	write V to x, or read K from x and pass it V
		$\mid$	$\mathbf{case}\, x_m\, K$	write K to x, or read V from x and pass it to K
		$\mid$	$x_k \leftarrow p\, \overline{y_m}$	call p with arguments $\overline{y}$ and destination x
Values	V	$::=$	$i(y_m)$	label i tagging address y ($\oplus$, $\&$)
		$\mid$	$\langle w_m, y_m\rangle$	pair of addresses w and y ($\otimes$, $\multimap$)
		$\mid$	$\langle\,\rangle$	unit value ($\mathbf{1}$)
		$\mid$	$\mathbf{shift}(y_k)$	shifted address y_k ($\downarrow$, $\uparrow$)
Continuations	K	$::=$	$(\ell(y_m) \Rightarrow P_\ell)_{\ell\in L}$	branch on $\ell \in L$ to bind y ($\oplus$, $\&$)
		$\mid$	$(\langle w_m, y_m\rangle \Rightarrow P)$	match against pair to bind $\langle w, y\rangle$ ($\otimes$, $\multimap$)
		$\mid$	$(\langle\,\rangle \Rightarrow P)$	match against unit element ($\mathbf{1}$)
		$\mid$	$(\mathbf{shift}(y_k) \Rightarrow P)$	match against **shift** to bind y_k ($\downarrow$, $\uparrow$)

Fig. 1. Types and process expressions

processes for clarity, and so we will use it both in examples and in the typing rules, where it helps to clarify a key intuition of this system, which is that *right rules write* and *left rules read*. Not all reads and writes will be visible like this, however — we may call a process or invoke a stored continuation, and the resulting process may read or write (but since it is not obligated to, we do not mark these reads/writes at the callsite). The rules for this judgment can be found in Fig. 2, where we have elided (other than in the call rule, which makes explicit use of it) a fixed signature Σ with type and process definitions explained later in this section. As usual, we require each of the x_i and x to be distinct and allow silent renaming of bound variables[2] in process expressions.

In this formulation, contraction and weakening remain *implicit*.[3] Handling contraction leads us to two versions of each of the $\oplus$, $\mathbf{1}$, $\otimes$ left rules, depending on whether the principal formula of the rule can be used again or not. The subscript α on each of these rules may be either 0 or 1, and indicates whether the principal formula of the rule is preserved in

[2] Variables are bound in two ways. The cut construct $x_m \leftarrow P \,;\, Q$ binds x_m in both P and Q, and continuations K may bind variables. For instance, $(\langle w_m, y_m\rangle \Rightarrow Q)$ binds w_m and y_m in Q, while $(\langle\rangle \Rightarrow Q)$ binds no variables in Q. Each continuation K thus resembles a closure in a functional language, specifying both what variables are bound and the process term that they are bound in.

[3] See (Pruiksma *et al.*, 2018) for a formulation of adjoint logic with *explicit* structural rules, which are less amenable to programming.

$$\frac{(\Gamma_C, \Delta \geq m \geq r) \quad \Gamma_C, \Delta \vdash P :: (x : A_m) \quad \Gamma_C, \Delta', x : A_m \vdash Q :: (z : C_r)}{\Gamma_C, \Delta, \Delta' \vdash (x \leftarrow P \,;\, Q) :: (z : C_r)}\;\mathsf{cut} \qquad \frac{}{\Gamma_W, y : A_m \vdash x^{\mathsf{W}} \leftarrow y^{\mathsf{R}} :: (x : A_m)}\;\mathsf{id}$$

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

$$\frac{(i \in L)}{\Gamma_W, y : A^i_m \vdash x^{\mathsf{W}}.i(y) :: (x : \oplus_m\{\ell : A^{\ell}_m\}_{\ell \in L})}\;\oplus R^0$$

$$\frac{\Gamma, (x : \oplus_m\{\ell : A^{\ell}_m\}_{\ell \in L})^{\alpha}, y : A^{\ell}_m \vdash Q_{\ell} :: (z : C_r) \quad (\text{for all } \ell \in L)}{\Gamma, x : \oplus_m\{\ell : A^{\ell}_m\}_{\ell \in L} \vdash \mathbf{case}\, x^{\mathsf{R}}\; (\ell(y) \Rightarrow Q_{\ell})_{\ell \in L} :: (z : C_r)}\;\oplus L_{\alpha}$$

$$\frac{\Gamma \vdash P_{\ell} :: (y : A^{\ell}_m) \quad (\text{for all } \ell \in L)}{\Gamma \vdash \mathbf{case}\, x^{\mathsf{W}}\; (\ell(y) \Rightarrow P_{\ell})_{\ell \in L} :: (x : \&_m\{\ell : A^{\ell}_m\}_{\ell \in L})}\;\& R \qquad \frac{(i \in L)}{\Gamma_W, x : \&_m\{\ell : A^{\ell}_m\}_{\ell \in L} \vdash x^{\mathsf{R}}.i(y) :: (y : A^i_m)}\;\& L^0$$

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

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

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

$$\frac{}{\Gamma_W, y : A_m \vdash x^{\mathsf{W}}_k.\mathbf{shift}(y_m) :: (x : {\downarrow}^m_k A_m)}\;{\downarrow}R^0 \qquad \frac{\Gamma, (x : {\downarrow}^m_k A_m)^{\alpha}, y : A_m \vdash Q :: (z : C_r)}{\Gamma, x : {\downarrow}^m_k A_m \vdash \mathbf{case}\, x^{\mathsf{R}}_k\; (\mathbf{shift}(y_m) \Rightarrow Q) :: (z :: C_r)}\;{\downarrow}L_{\alpha}$$

$$\frac{\Gamma \vdash P :: (y : A_k)}{\Gamma \vdash \mathbf{case}\, x^{\mathsf{W}}_m\; (\mathbf{shift}(y_k) \Rightarrow P) :: (x : {\uparrow}^m_k A_k)}\;{\uparrow}R \qquad \frac{}{\Gamma_W, x : {\uparrow}^m_k A_k \vdash x^{\mathsf{R}}_m.\mathbf{shift}(y_k) :: (y : A_k)}\;{\uparrow}L^0$$

Fig. 2. Typing rules ($\alpha \in \{0, 1\}$ with $\alpha = 1$ permitted only if $C \in \sigma(m)$)

the context. The $\alpha = 0$ version of each rule is the standard linear form, while the $\alpha = 1$ version, which requires that the mode m of the principal formula satisfies $C \in \sigma(m)$, keeps a copy of the principal formula. Note that if $C \in \sigma(m)$, we are still allowed to use the $\alpha = 0$ version of the rule. Moreover, we write Γ_C, Γ_W for contexts of variables all of which allow contraction or weakening, respectively. This allows us to freely drop weakenable variables when we reach initial rules, or to duplicate contractable variables to both parent and child when spawning a new process in the cut rule.

Note that there is no explicit rule for (possibly recursively defined) type variables t, since they can be silently replaced by their definitions. Equality between types and type-checking can both easily be seen to be decidable using a combination of standard techniques for substructural type systems (Cervesato *et al.*, 2000) and subtyping for equirecursive session types, (Gay & Hole, 2005) which relies on a coinductive interpretation of the types, but not on their linearity, and so can be adapted to the adjoint setting. Some experience with a closely related algorithm (Das & Pfenning, 2020) for type equality and type checking suggests that this is practical.

We now go on to briefly examine the terms and loosely describe their meanings from the perspective of a shared-memory semantics. We will make this more precise in Sections 3 and 4, where we develop the dynamics of such a shared-memory semantics.

Both the grammar and the typing rules show that we have five primary constructs for processes, which then break down further into specific cases.

The first two process constructs are type-agnostic. The cut rule, with term $x \leftarrow P\,; Q$, allocates a new memory cell x, spawns a new process P which may write to x, and continues as Q which may read from x. The new cell x thus serves as the point of communication between the new process P and the continuing Q. The id rule, with term $x_m \leftarrow y_m$, copies the contents of cell y_m into cell x_m. If $C \notin \sigma(m)$, we can think of this instead as *moving* the contents of cell y_m into cell x_m and freeing y_m.

The next two constructs, $x.V$ and $\textbf{case}\, x\, K$, come in pairs that perform communication, one pair for each type. A process of one of these forms will either write to or read from x, depending on whether the variable is in the succedent (write) or antecedent (read).

A write is straightforward and stores either the value V or the continuation K into the cell x, while a read pulls a continuation K' or a value V' from the cell, and combines either K' and V (in the case of $x.V$) or K and V' ($\textbf{case}\, x\, K$). The symmetry of this, in which continuations and values are both eligible to be written to memory and read from memory, comes from the duality between $\oplus$ and $\&$, between $\otimes$ and $\multimap$, and between $\downarrow$ and $\uparrow$. We see this in the typing rules, where, for instance, $\oplus R^0$ and $\& L^0$ have the same process term, swapping only the roles of each variable between read and write. As cells may contain either values V or continuations K, it will be useful to have a way to refer to this class of expression:

$$\text{Cell data} \qquad D ::= V \mid K$$

The final construct allows for calling named processes, which we use for recursion. As is customary in session types, we use *equirecursive types*, collected in a signature Σ in which we also collect recursive process definitions and their types. For each type definition $t = A$, the type A must be *contractive* so that we can treat types *equirecursively* with a straightforward coinductive definition and an efficient algorithm for type equality (Gay & Hole, 2005).

A named process p is *declared* as $B^1_{m_1}, \ldots, B^n_{m_n} \vdash p :: A_k$ which means it requires arguments of types $B^i_{m_i}$ (in that order) and provides a result of type A_k. For ease of readability, we may sometimes write in variables names as well, but they are actually only needed for the corresponding *definitions* $x \leftarrow p\, y_1, \ldots, y_n = P$. Operationally, a call $z \leftarrow p\, \overline{w}$ expands to its definition with a substitution $[\overline{w}/\overline{y}, z/x]P$, replacing variables by addresses.

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

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

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

In the remainder of this paper we assume that we have a fixed valid signature Σ, so we annotate neither the typing judgment nor the computation rules with an explicit signature, other than in the call rule, where we extract a process definition from Σ.

3 Concurrent Semantics

We will now present a concurrent shared-memory semantics for Seax, using *multiset rewriting rules* (Cervesato & Scedrov, 2009). The state of a running program is a multiset of semantic objects, which we refer to as a *process configuration*. We have three distinct types of semantic objects:

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

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

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

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

We think of the join $\mathscr{C}_1, \mathscr{C}_2$ of two configurations as a commutative and associative operation so that this grammar defines a multiset rather than a list or tree.

A multiset rewriting rule takes the collection of objects on the left-hand side of the rule, consumes them (if they are ephemeral), and then adds in the objects on the right-hand side of the rule. Rules may be applied to any subconfiguration, leaving the remainder of the configuration unchanged. This yields a naturally nondeterministic semantics, but we will see that the semantics are nonetheless confluent (Theorem 3). Additionally, while our configurations are not ordered, we will adopt the convention that the writer of an address appears to the left of any readers of that address.

Our semantic rules are based on a few key ideas:

1. Variables represent addresses in shared memory.
2. Cut/spawn is the only way to allocate a new cell.
3. Identity/forward will move or copy data between cells.

4. A process $\mathsf{thread}(c, P)$ will (eventually) write to the cell at address c and then terminate.
5. A process $\mathsf{thread}(d, Q)$ that is trying to read from $c \neq d$ will wait until the cell with address c is available (i.e. its contents is no longer _), perform the read, and then continue.

The counterintuitive part of this interpretation (when using a message-passing semantics as a point of reference) is that a process providing $c : A \mathbin{\&} B$ *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. This ability to write continuations to memory is a major feature distinguishing this from a message-passing semantics where potentially large closures would have to be captured, serialized, and deserialized, the cost of which is difficult to control (Miller *et al.*, 2016).

The final piece that we need to present the semantics is a key operation, namely that of passing a value V to a continuation K to get a new process P. This operation is defined as follows:

$$\begin{array}{lcll}
i(d) \circ (\ell(y) \Rightarrow P_\ell)_{\ell \in L} & = & [d/y]P_i & (\oplus, \mathbin{\&}) \\
\langle e, c \rangle \circ (\langle w, y \rangle \Rightarrow P) & = & [e/w, c/y]P & (\otimes, \multimap) \\
\langle\,\rangle \circ (\langle\,\rangle \Rightarrow P) & = & P & (\mathbf{1}) \\
\mathbf{shift}(d) \circ (\mathbf{shift}(y) \Rightarrow P) & = & [d/y]P & (\downarrow, \uparrow)
\end{array}$$

When any of these reductions is applied, either the value or the continuation has been read from a cell while the other is a part of the executing process. With this notation, we can give a concise set of rules for the concurrent dynamics. We present these rules in Fig. 3.

$$\begin{array}{ll}
\mathsf{thread}(c, x \leftarrow P\,;Q) \mapsto \mathsf{thread}(a, [a/x]P), \mathsf{cell}(a, \_), \mathsf{thread}(c, [a/x]Q)\ (a \text{ fresh}) & \textit{cut: allocate \& spawn} \\
!_m\mathsf{cell}(c_m, D), \mathsf{thread}(d_m, d_m \leftarrow c_m), \mathsf{cell}(d_m, \_) \mapsto\ !_m\mathsf{cell}(d_m, D) & \textit{id: move or copy} \\
\mathsf{thread}(c, c \leftarrow p\,\overline{d}) \mapsto \mathsf{thread}(c, [c/x, \overline{d/y}]P) \quad \text{for } x \leftarrow p\,\overline{y} = P \in \Sigma & \textit{call} \\
\mathsf{thread}(c_m, c_m.V), \mathsf{cell}(c_m, \_) \mapsto\ !_m\mathsf{cell}(c_m, V) & (\oplus R^0, \otimes R^0, \mathbf{1}R^0, \downarrow R^0) \\
!_m\mathsf{cell}(c_m, V), \mathsf{thread}(e_k, \mathbf{case}\ c_m\ K) \mapsto \mathsf{thread}(e_k, V \circ K) & (\oplus L, \otimes L, \mathbf{1}L, \downarrow L) \\
\mathsf{thread}(c_m, \mathbf{case}\ c_m\ K), \mathsf{cell}(c_m, \_) \mapsto\ !_m\mathsf{cell}(c_m, K) & (\multimap R, \mathbin{\&}R, \uparrow R) \\
!_m\mathsf{cell}(c_m, K), \mathsf{thread}(d_k, c_m.V) \mapsto \mathsf{thread}(d_k, V \circ K) & (\multimap L^0, \mathbin{\&}L^0, \uparrow L^0)
\end{array}$$

Fig. 3. Concurrent dynamic rules

These rules match well with our intuitions from before. In the cut rule, we allocate a new empty cell a, spawn a new thread to execute P, and continue executing Q, just as we described informally in Section 2. Similarly, in the id rule, we either move or copy (depending on whether $C \in \sigma(m)$) the contents of cell c into cell d and terminate. The rules that write values to cells are exactly the right rules for positive types ($\oplus$, $\otimes$, $\mathbf{1}$, $\downarrow$), while the right rules for negative types ($\mathbin{\&}$, $\multimap$, $\uparrow$) write continuations to cells instead. Dually, to read from a cell of positive type, we must have a continuation to pass the stored value to, while to read from a cell of negative type, we need a value to pass to the stored continuation.

3.1 Results

We have standard results for this system — a form of progress, of preservation, and a confluence result. To discuss progress and preservation, we must first extend our notion of typing for process terms to configurations. Configurations are typed with the judgment $\Gamma \vdash \mathscr{C} :: \Delta$ which means that configuration $\mathscr{C}$ may read from the addresses in Γ and write to the addresses in Δ. We can then give the following set of rules for typing configurations, which make use of the typing judgment $\Gamma \vdash P :: (c : A_m)$ for process terms in the base cases. Recall that we use Γ_C to denote a context in which all propositions are contractible, and which can therefore be freely duplicated.

$$\frac{\Gamma_C, \Gamma \vdash P :: (c : A_m)}{\Gamma_C, \Gamma, \Delta \vdash \mathsf{thread}(c, P), \mathsf{cell}(c, \_) :: (\Gamma_C, \Delta, c : A_m)}$$

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

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

Note that our invariants on configurations mean that there is no need to separately type the objects $\mathsf{thread}(c, P)$ and $\mathsf{cell}(c, \_)$, as they can only occur together. Additionally, while our configurations are multisets, and therefore not inherently ordered, observe that the typing derivation for a configuration induces an order on the configuration, something which is quite useful in proving progress. [4]

Our preservation theorem differs slightly from the standard, in that it allows the collection of typed channels Δ offered by a configuration $\mathscr{C}$ to grow after a step, as steps may introduce new persistent memory cells. Note that the Δ cannot shrink, despite the fact that affine or linear cells may be deallocated after read. This is because a linear cell that is read from never appeared in Δ in the first place — the process that reads it also consumes it in the typing derivation. Likewise, an affine cell that is read from will not appear in Δ, while an affine cell with no reader appears in Δ (but of course, since it has no reader, it will not be deallocated).

Theorem 1 (Type Preservation). *If* $\Gamma \vdash \mathscr{C} :: \Delta$ *and* $\mathscr{C} \mapsto \mathscr{C}'$ *then* $\Gamma \vdash \mathscr{C}' :: \Delta'$ *for some* $\Delta' \supseteq \Delta$.

Proof By cases on the transition relation for configurations, applying repeated inversions to the typing judgment on $\mathscr{C}$ to obtain the necessary information to assemble a typing derivation for $\mathscr{C}'$. This requires some straightforward lemmas expressing that non-interfering processes and cells can be exchanged in a typing derivation. ■

Progress is entirely standard, with configurations comprised entirely of filled cells taking the role that values play in a functional language.

[4] This technique is used in (DeYoung *et al.*, 2020) to prove progress for a similar language.

Theorem 2 (Progress). *If $\cdot \vdash \mathscr{C} :: \Delta$ then either*

(i) $\mathscr{C} \mapsto \mathscr{C}'$ for some $\mathscr{C}'$, or
(ii) for every channel $c_m : A_m \in \Delta$ there is an object $!_m\mathsf{cell}(c_m, D) \in \mathscr{C}$.

Proof We first re-associate all applications of the typing rule for joining configurations to the left. Then we perform an induction over the structure of the resulting derivation, distinguishing cases for the rightmost cell or thread and potentially applying the induction hypothesis on the configuration to its left. This structure, together with inversion on the typing of the cell or thread yields the theorem. ■

In addition to these essential properties, we also have a confluence result, for which we need to define a weak notion of equivalence on configurations. We say $\mathscr{C}_1 \sim \mathscr{C}_2$ if there is a renaming ρ of addresses such that $\rho\mathscr{C}_1 = \mathscr{C}_2$. We can then establish the following version of the diamond property:

Theorem 3 (Diamond Property). *Assume $\Delta \vdash \mathscr{C} :: \Gamma$. If $\mathscr{C} \mapsto \mathscr{C}_1$ and $\mathscr{C} \mapsto \mathscr{C}_2$ such that $\mathscr{C}_1 \not\sim \mathscr{C}_2$. Then there exist $\mathscr{C}_1'$ and $\mathscr{C}_2'$ such that $\mathscr{C}_1 \mapsto \mathscr{C}_1'$ and $\mathscr{C}_2 \mapsto \mathscr{C}_2'$ with $\mathscr{C}_1' \sim \mathscr{C}_2'$.*

Proof The proof is straightforward by cases. There are no critical pairs involving ephemeral (that is, non-persistent) objects in the left-hand sides of transition rules. ■

3.2 Examples

We present here a few examples of concurrent programs, illustrating various aspects of our language.

3.2.1 Example: Binary Numbers.

As a first simple example we consider binary numbers, defined as a type *bin* at mode *m*. The structural properties of mode *m* are arbitrary for our examples. For concreteness, assume that *m* is *linear*, that is, $\sigma(m) = \{\,\}$.

$$bin_m = \oplus_m\{\mathsf{b0} : bin_m, \mathsf{b1} : bin_m, \mathsf{e} : \mathbf{1}_m\}$$

Unless multiple modes are involved, we will henceforth omit the mode *m*. As an example, the number $6 = (110)_2$ would be represented by a sequence of labels e, b1, b1, b0, chained together in a linked list. The first cell in the list would contain the bit b0. It has some address c_0, and also contains an address c_1 pointing to the next cell in the list. Writing out the whole sequence as a configuration we have

$$\mathsf{cell}(c_4, \langle\,\rangle), \mathsf{cell}(c_3, \mathsf{e}(c_4)), \mathsf{cell}(c_2, \mathsf{b1}(c_3)), \mathsf{cell}(c_1, \mathsf{b1}(c_2)), \mathsf{cell}(c_0, \mathsf{b0}(c_1))$$

3.2.2 Example: Computing with Binary Numbers.

We implement a recursive process *succ* that reads the bits of a binary number *n* starting at address *y* and writes the bits for the binary number $n+1$ starting at *x*. This process may block until the input cell (referenced as *y*) has been written to; the output cells are allocated one by one as needed. Since we assumed the mode m is linear, the cells read by the *succ* process from will be deallocated.

```
$bin_m = \oplus_m\{\mathsf{b0} : bin_m, \mathsf{b1} : bin_m, \mathsf{e} : \mathbf{1}_m\}$
$(y : bin) \vdash succ :: (x : bin)$
$x \leftarrow succ\ y =$
    $\textbf{case}\ y^{\mathsf{R}}\ (\ \mathsf{b0}(y_1) \Rightarrow x_1 \leftarrow (x_1^{\mathsf{W}} \leftarrow y_1^{\mathsf{R}})\ ;$      % alloc $x_1$ and copy $y_1$ to $x_1$
                              $x^{\mathsf{W}}.\mathsf{b1}(x_1)$                             % write $\mathsf{b1}(x_1)$ to $x$
                  $\mid \mathsf{b1}(y_1) \Rightarrow x_1 \leftarrow (x_1 \leftarrow succ\ y_1)\ ;$   % alloc $x_1$ and spawn $succ\ y_1$
                              $x^{\mathsf{W}}.\mathsf{b0}(x_1)$                             % write $\mathsf{b0}(x_1)$ to $x$
                  $\mid \mathsf{e}(y_1) \Rightarrow x_2 \leftarrow (x_2^{\mathsf{W}} \leftarrow y_1^{\mathsf{R}})\ ;$       % alloc $x_2$ and copy $y_1$ to $x_2$
                             $x_1 \leftarrow x_1^{\mathsf{W}}.\mathsf{e}(x_2)\ ;$                   % alloc $x_1$ and write $\mathsf{e}(x_2)$ to $x_1$
                             $x^{\mathsf{W}}.\mathsf{b1}(x_1)\ )$                            % write $\mathsf{b1}(x_1)$ to $x$
```

In this example and others we find certain repeating patterns. Abbreviating these makes the code easier to read and also more compact to write. As a first simplification, we can use the following shortcuts:

$$\begin{array}{lcl} x \leftarrow y\ ; Q & \triangleq & x \leftarrow (x \leftarrow y)\ ; Q \\ x \leftarrow f\ \bar{y}\ ; Q & \triangleq & x \leftarrow (x \leftarrow f\ \bar{y})\ ; Q \end{array}$$

With these, the code for successor becomes

```
$x \leftarrow succ\ y =$
    $\textbf{case}\ y^{\mathsf{R}}\ (\ \mathsf{b0}(y_1) \Rightarrow x_1^{\mathsf{W}} \leftarrow y_1^{\mathsf{R}}\ ;$          % alloc $x_1$ and copy $y_1$ to $x_1$
                              $x^{\mathsf{W}}.\mathsf{b1}(x_1)$                 % write $\mathsf{b1}(x_1)$ to $x$
                  $\mid \mathsf{b1}(y_1) \Rightarrow x_1 \leftarrow succ\ y_1\ ;$      % alloc $x_1$ and spawn $succ\ y_1$
                              $x^{\mathsf{W}}.\mathsf{b0}(x_1)$                 % write $\mathsf{b0}(x_1)$ to $x$
                  $\mid \mathsf{e}(y_1) \Rightarrow x_2^{\mathsf{W}} \leftarrow y_1^{\mathsf{R}}\ ;$           % alloc $x_2$ and copy $y_1$ to $x_2$
                             $x_1 \leftarrow x_1^{\mathsf{W}}.\mathsf{e}(x_2)\ ;$       % alloc $x_1$ and write $\mathsf{e}(x_2)$ to $x_1$
                             $x^{\mathsf{W}}.\mathsf{b1}(x_1)\ )$                % write $\mathsf{b1}(x_1)$ to $x$
```

The second pattern we notice are sequences of allocations followed by immediate (single) uses of the new address. We can collapse these by a kind of specialized substitution. We describe the inverse, namely how the abbreviated notation is *elaborated* into the language primitives.

$$\text{Value Sequence} \quad \bar{V} \quad ::= \quad i(\bar{V}) \mid (y, \bar{V}) \mid \textbf{shift}(\bar{V}) \mid V$$

At positive types ($\oplus$, $\otimes$, $\mathbf{1}$, $\downarrow$), which write to the variable x with $x.\bar{V}$, we define:

$$\begin{array}{lcll} x^{\mathsf{W}}\ .\ i(\bar{V}) & \triangleq & x_1 \leftarrow x_1^{\mathsf{W}}\ .\ \bar{V}\ ; x^{\mathsf{W}}.i(x_1) & (\oplus) \\ x^{\mathsf{W}}\ .\ \langle y, \bar{V}\rangle & \triangleq & x_1 \leftarrow x_1^{\mathsf{W}}\ .\ \bar{V}\ ; x^{\mathsf{W}}.\langle y, x_1\rangle & (\otimes) \\ x^{\mathsf{W}}\ .\ \textbf{shift}(\bar{V}) & \triangleq & x_1 \leftarrow x_1^{\mathsf{W}}\ .\ \bar{V}\ ; x^{\mathsf{W}}.\textbf{shift}(x_1) & (\downarrow) \end{array}$$

In each case, and similar definitions below, x_1 is a fresh variable. Using these abbreviations in our example, we can shorten it further.

```
$x \leftarrow succ\ y =$
```

$$\begin{array}{lll}
\textbf{case}\ y^{\mathsf{R}}\ (\ \mathsf{b0}(y_1) \Rightarrow x^{\mathsf{W}}.\mathsf{b1}(y_1) & & \textit{\% write } \mathsf{b1}(y_1) \textit{ to } x \\
\quad\quad \mid \mathsf{b1}(y_1) \Rightarrow x_1 \leftarrow \textit{succ}\ y_1\ ; & & \textit{\% alloc } x_1 \textit{ and spawn succ } y_1 \\
\quad\quad\quad\quad\quad x^{\mathsf{W}}.\mathsf{b0}(x_1) & & \textit{\% write } \mathsf{b0}(x_1) \textit{ to } x \\
\quad\quad \mid \mathsf{e}(y_1) \Rightarrow x^{\mathsf{W}}.\mathsf{b1}(\mathsf{e}(y_1))\ ) & & \textit{\% write } \mathsf{b1}(\mathsf{e}(y_1)) \textit{ to } x
\end{array}$$

For negative types ($\&$, $\multimap$, $\uparrow$) the expansion is symmetric, swapping the left- and right-hand sides of the cut. This is because these constructs read a continuation from memory at x and pass it a value.

$$\begin{array}{lcll}
x^{\mathsf{R}} \,.\, i(\bar{V}) & \triangleq & x_1 \leftarrow x^{\mathsf{R}}.i(x_1)\ ;\ x_1^{\mathsf{R}} \,.\, \bar{V} & (\&) \\
x^{\mathsf{R}} \,.\, \langle y, \bar{V}\rangle & \triangleq & x_1 \leftarrow x^{\mathsf{R}}.\langle y, x_1\rangle\ ;\ x_1^{\mathsf{R}} \,.\, \bar{V} & (\multimap) \\
x^{\mathsf{R}} \,.\, \textbf{shift}(\bar{V}) & \triangleq & x_1 \leftarrow x^{\mathsf{R}}.\textbf{shift}(x_1)\ ;\ x_1^{\mathsf{R}} \,.\, \bar{V} & (\uparrow)
\end{array}$$

Similarly, we can decompose a continuation matching against a value sequence $(\bar{V} \Rightarrow P)$. For simplicity, we assume here that the labels for each branch of a pattern match for internal ($\oplus$) or external ($\&$) choice are distinct; a generalization to nested patterns is conceptually straightforward but syntactically somewhat complex so we do not specify it formally.

$$\begin{array}{lcll}
(\ell(\bar{V}_\ell) \Rightarrow P_\ell)_{\ell\in L} & \triangleq & (\ell(x_1) \Rightarrow \textbf{case}\, x_1\ (\bar{V}_\ell \Rightarrow P_\ell))_{\ell\in L} & (\oplus, \&) \\
(\langle y, \bar{V}\rangle \Rightarrow P) & \triangleq & (\langle y, x_1\rangle \Rightarrow \textbf{case}\, x_1\ (\bar{V} \Rightarrow P)) & (\otimes, \multimap) \\
(\textbf{shift}(\bar{V}) \Rightarrow P) & \triangleq & (\textbf{shift}(x_1) \Rightarrow \textbf{case}\, x_1\ (\bar{V} \Rightarrow P)) & (\downarrow, \uparrow)
\end{array}$$

For example, we can rewrite the successor program one more time to express that y_1 in the last case must actually contain the unit element $\langle\ \rangle$ and match against it as well as construct it on the right-hand side.

$$\begin{array}{lll}
x \leftarrow \textit{succ}\ y = & & \\
\quad \textbf{case}\ y^{\mathsf{R}}\ (\ \mathsf{b0}(y_1) \Rightarrow x^{\mathsf{W}}.\mathsf{b1}(y_1) & & \textit{\% write } \mathsf{b1}(y_1) \textit{ to } x \\
\quad\quad\quad \mid \mathsf{b1}(y_1) \Rightarrow x_1 \leftarrow \textit{succ}\ y_1\ ; & & \textit{\% alloc } x_1 \textit{ and spawn succ } y_1 \\
\quad\quad\quad\quad\quad\quad x^{\mathsf{W}}.\mathsf{b0}(x_1) & & \textit{\% write } \mathsf{b0}(x_1) \textit{ to } x \\
\quad\quad\quad \mid \mathsf{e}\,\langle\ \rangle \Rightarrow x^{\mathsf{W}}.\mathsf{b1}(\mathsf{e}\,\langle\ \rangle)\ ) & & \textit{\% write } \mathsf{b1}(\mathsf{e}\,\langle\ \rangle) \textit{ to } x
\end{array}$$

We have to remember, however, that intermediate matches and allocations still take place and the last two programs are *not equivalent* in case the process with destination y' does not terminate.

To implement *plus2* we can just compose *succ* with itself.

$$\begin{array}{l}
(z : \textit{bin}) \vdash \textit{plus2} :: (x : \textit{bin}) \\[4pt]
x \leftarrow \textit{plus2}\ z = \\
\quad y \leftarrow \textit{succ}\ z\ ; \\
\quad x \leftarrow \textit{succ}\ y
\end{array}$$

In our concurrent semantics, the two successor processes form a concurrently executing pipeline — the first reads the initial number from memory, bit by bit, and then writes a new number (again, bit by bit) to memory for the second successor process to read.

3.2.3 *Example: MapReduce.*

As a second example we consider *mapReduce* applied to a tree. We have a neutral element z (which stands in for every leaf) and a process f to be applied at every node to reduce the whole tree to a single value. This exhibits a high degree of parallelism, since the operations on the left and right subtree can be done independently. We abstract over the type of element A and the result B at the meta-level, so that tree_A is a family of types, and $mapReduce_{AB}$ is a family of processes, indexed by A and B.

$$tree_A = \oplus_m\{\mathsf{empty} : \mathbf{1}, \mathsf{node} : tree_A \otimes A \otimes tree_A\}$$

Since *mapReduce* applies reduction at every node in the tree, it is *linear* in the tree. On the other hand, the neutral element z is used for every leaf, and the associative operation f for every node, so z requires at least contraction (there must be at least one leaf) and f both weakening and contraction (there may be arbitrarily many nodes). Therefore we use three modes: the linear mode m for the tree and the result of *mapReduce*, a strict mode s for the neutral element z, and an unrestricted mode u for the operation applied at each node.

$$(z : {\uparrow}^s_m B)\ (f : {\uparrow}^u_m((B \otimes A \otimes B) \multimap B))\ (t : tree_A) \vdash mapReduce_{AB} :: (s : B)$$

```
$s \leftarrow mapReduce_{AB}\ z\ f\ t =$
    $\mathbf{case}\ t^{\mathsf{R}}\ \ (\mathsf{empty}\ \langle\,\rangle \Rightarrow z_1 \leftarrow z^{\mathsf{R}}.\mathbf{shift}(z_1)$                 $\%\ drop\ f$
                                $s^{\mathsf{W}} \leftarrow z_1^{\mathsf{R}}$
                $\mid \mathsf{node}\ \langle l, \langle x, r\rangle\rangle \Rightarrow l_1 \leftarrow mapReduce_{AB}\ z\ f\ l\ ;$
                                  $r_1 \leftarrow mapReduce_{AB}\ z\ f\ r\ ;$      $\%\ duplicate\ z\ and\ f$
                                  $p \leftarrow p^{\mathsf{W}}.\langle l_1, \langle x, r_1\rangle\rangle\ ;$
                                  $s_1 \leftarrow f^{\mathsf{R}}.\mathbf{shift}\langle p, s_1\rangle\ ;$
                                  $s^{\mathsf{W}} \leftarrow s_1^{\mathsf{R}}\ )$
```

3.2.4 *Example: λ-Calculus.*

As a third example we show an encoding of the λ-calculus using higher-order abstract syntax and parallel evaluation. We specify, at an arbitrary mode m:

$$exp_m = \oplus\{\mathsf{app} : exp \otimes exp, \mathsf{val} : val\}$$
$$val_m = \oplus\{\mathsf{lam} : val \multimap exp\}$$

An interesting property of this representation is that if we pick m to be linear, we obtain the linear λ-calculus (Lincoln & Mitchell, 1992), if we pick m to be strict ($\sigma(m) = \{C\}$) we obtain Church and Rosser's original λI calculus (Church & Rosser, 1936), and if we set $\sigma(m) = \{W, C\}$ we obtain the usual (intuitionistic) λ-calculus. Evaluation (that is, parallel reduction to a weak head-normal form) is specified by the following process, no matter which version of the λ-calculus we consider.

$$(e : exp) \vdash eval : (v : val)$$

```
$v \leftarrow eval\ e =$
    $\mathbf{case}\ e^{\mathsf{R}}\ \ (\mathsf{val}(v_1) \Rightarrow v^{\mathsf{W}} \leftarrow v_1^{\mathsf{R}}$
    $\mid \mathsf{app}\ \langle e_1, e_2\rangle \Rightarrow v_1 \leftarrow eval\ e_1$ ;
                              $v_2 \leftarrow eval\ e_2$ ;
                              **case** $v_1^{\mathsf{R}}$ ( $\mathsf{lam}(f) \Rightarrow e_3 \leftarrow f^{\mathsf{R}}.\langle v_2, e_3\rangle$ ;      % $f : val \multimap exp$
                                                 $v \leftarrow eval\ e_3$ ) )
```

In this code, v_2 acts like a *future*: we spawn the evaluation of e_2 with the promise to place the result in v_2. In our dynamics, we allocate a new cell for v_2, as yet unfilled. When we pass v_2 to f in $f.\langle v_2, e_3\rangle$ the process $eval\ e_2$ may still be computing and we will not block until we eventually try to read from v_2 (which may nor may not happen).

4 Sequential Semantics

While our concurrent semantics is quite expressive and allows for a great deal of parallelism, in a real-world setting, the overhead of spawning a new thread can make it inefficient to do so unless the work that thread does is substantial. The ability to express sequentiality is therefore convenient from an implementation standpoint, as well as for ease of reasoning about programs. Moreover, many of the patterns of concurrent computation that we would like to model involve adding some limited access to concurrency in a largely sequential language. We can address both of these issues with the concurrent semantics by adding a construct to enforce sequentiality. Here, we will take as our definition of sequentiality that only one thread (the *active thread*) is able to take a step at a time, with all other threads being blocked.

The key idea in enforcing sequentiality is to observe that only the cut/spawn rule turns a single thread into two. When we apply the cut/spawn rule to the term $x \leftarrow P \,;\, Q$, P and Q are executed concurrently. One obvious way (we discuss another later in this section) to enforce sequentiality is to introduce a *sequential cut* construct $x \Leftarrow P \,;\, Q$ that ensures that P runs to completion, writing its result into x, before Q can continue. We do not believe that we can ensure this using our existing (concurrent) semantics. However, with a small addition to the language and semantics, we are able to define a sequential cut as syntactic sugar for a Seax term that does enforce this.

Example Revisited: A Sequential Successor. Before we move to the formal definition that enforces sequentiality, we reconsider the successor example on binary numbers in its most explicit form. We make all cuts sequential.

```
$bin_m = \oplus_m\{\mathsf{b0} : bin_m, \mathsf{b1} : bin_m, \mathsf{e} : \mathbf{1}_m\}$
$(y : bin) \vdash succ :: (x : bin)$
$x \leftarrow succ\ y =$
    **case** $y^{\mathsf{R}}$ ( $\mathsf{b0}(y_1) \Rightarrow x_1 \Leftarrow (x_1^{\mathsf{W}} \leftarrow y_1^{\mathsf{R}})$ ;          % *alloc* $x_1$ *and copy* $y_1$ *to* $x_1$
                      $x^{\mathsf{W}}.\mathsf{b1}(x_1)$                    % *write* $\mathsf{b1}(x_1)$ *to* $x$
             $\mid \mathsf{b1}(y_1) \Rightarrow x_1 \Leftarrow (x_1 \leftarrow succ\ y_1)$ ;      % *alloc* $x_1$ *and spawn* $succ\ y_1$
                      $x^{\mathsf{W}}.\mathsf{b0}(x_1)$                    % *write* $\mathsf{b0}(x_1)$ *to* $x$
             $\mid \mathsf{e}(y_1) \Rightarrow x_2 \Leftarrow (x_2^{\mathsf{W}} \leftarrow y_1^{\mathsf{R}})$            % *alloc* $x_2$ *and copy* $y_1$ *to* $x_2$
                      $x_1 \Leftarrow x_1^{\mathsf{W}}.\mathsf{e}(x_2)$              % *alloc* $x_1$ *and write* $\mathsf{e}(x_2)$ *to* $x_1$
```

$$x^{\mathsf{W}}.\mathsf{b1}(x_1)\;) \qquad\qquad \textit{\% write } \mathsf{b1}(x_1) \textit{ to } x$$

This now behaves like a typical sequential implementation of a successor function, but in destination-passing style (Wadler, 1984; Larus, 1989; Cervesato *et al.*, 2002; Simmons, 2012). Much like in continuation-passing style, where each function, rather than returning, calls a continuation that is passed in, in destination-passing style, rather than returning, a function stores its result in a destination that is passed in. Likewise, our processes take in an address or destination, compute their result, and write it to that address. When there is a carry (manifest as a recursive call to *succ*), the output bit zero will not be written until the effect of the carry has been fully computed.

To implement sequential cut, we will take advantage of the fact that a shift from a mode m to itself does not affect provability, but does force synchronization. If $x : A_m$, we would like to define

$$x \Leftarrow P \,;\, Q \triangleq x_1 \leftarrow P' \,;\, \mathbf{case}\, x_1 \;(\mathbf{shift}(x) \Rightarrow Q),$$

where $x_1 : {\downarrow}_m^m A_m$, and (informally) P' behaves like P, except that wherever P would write to x, P' writes simultaneously to x and x_1. By Setting aside for now a formal definition of P', we see that Q is blocked until x_1 has been written to, and so as long as P' writes to x_1 no later than it writes to x, this ensures that x is written to before Q can continue. By doing this, we use x_1 as a form of acknowledgment which cannot be written to until P has finished its computation.[5]

We now see that in order to define P' from P, we need some way to ensure that x_1 is written to no later than x. The simplest way to do this is to add a form of *atomic write* which writes to two cells simultaneously. We define three new constructs for these atomic writes, shown here along with the non-atomic processes that they imitate. We do not show typing rules here, but each atomic write can be typed in the same way as its non-atomic equivalent.

Atomic Write	Non-atomic equivalent
$x_1^{\mathsf{W}}.\mathbf{shift}(x^{\mathsf{W}}.V)$	$x \leftarrow x^{\mathsf{W}}.V \,;\, x_1^{\mathsf{W}}.\mathbf{shift}(x)$
$x_1^{\mathsf{W}}.\mathbf{shift}(\mathbf{case}\, x^{\mathsf{W}}\; K)$	$x \leftarrow \mathbf{case}\, x^{\mathsf{W}}\; K \,;\, x_1^{\mathsf{W}}.\mathbf{shift}(x)$
$x_1^{\mathsf{W}}.\mathbf{shift}(x^{\mathsf{W}} \leftarrow y^{\mathsf{R}})$	$x \leftarrow (x^{\mathsf{W}} \leftarrow y^{\mathsf{R}}) \,;\, x_1^{\mathsf{W}}.\mathbf{shift}(x)$

Each atomic write simply evaluates in a single step to the configuration where both x and x_1 have been written to, much as if the non-atomic equivalent had taken three steps — first for the cut, second to write to x, and third to write to x_1. This intuition is formalized in the following transition rules:

$$\begin{array}{ll}
\mathsf{thread}(x_{1m}, x_{1m}^{\mathsf{W}}.\mathbf{shift}(x_k^{\mathsf{W}}.V)) \mapsto {!_k}\mathsf{cell}(x_k, V),\, {!_m}\mathsf{cell}(x_{1m}, \mathbf{shift}(x_k)) & \textit{atom-val} \\
\mathsf{thread}(x_{1m}, x_{1m}^{\mathsf{W}}.\mathbf{shift}(\mathbf{case}\, x_k^{\mathsf{W}}\; K)) \mapsto {!_k}\mathsf{cell}(x_k, K),\, {!_m}\mathsf{cell}(x_{1m}, \mathbf{shift}(x_k)) & \textit{atom-cont} \\
\mathsf{thread}(x_{1m}, x_{1m}^{\mathsf{W}}.\mathbf{shift}(x_k^{\mathsf{W}} \leftarrow y_k^{\mathsf{R}})),\, {!_k}\mathsf{cell}(y_k, D) \mapsto {!_k}\mathsf{cell}(x_k, D),\, {!_m}\mathsf{cell}(x_{1m}, \mathbf{shift}(x_k)) & \textit{atom-id}
\end{array}$$

Note that the rule for the identity case is different from the other two — it requires the cell y_k to have been written to in order to continue. This is because the $x^{\mathsf{W}} \leftarrow y^{\mathsf{R}}$ construct reads

[5] There are other ways that we could handle sequentiality, for instance by adding a new construct that blocks waiting for x to be written to, using x as its own acknowledgment. Each approach has its own advantages, and we use a separate acknowledgment via a shift because this approach generalizes more smoothly to call-by-name, which we explore later on.

from y and writes to x — if we wish to write to x and x_1 atomically, we must also perform the read from y.

Now, to obtain P' from P, we define a substitution operation $[x_1.\mathbf{shift}(x)/\!\!/x]$ that replaces writes to x with atomic writes to x and x_1 as follows:

$$\begin{array}{rcl}
(x^{\mathsf{W}}.V)[x_1.\mathbf{shift}(x)/\!\!/x] & = & x_1^{\mathsf{W}}.\mathbf{shift}(x^{\mathsf{W}}.V) \\
(\mathbf{case}\, x^{\mathsf{W}}\ K)[x_1.\mathbf{shift}(x)/\!\!/x] & = & x_1^{\mathsf{W}}.\mathbf{shift}(\mathbf{case}\, x^{\mathsf{W}}\ K) \\
(x^{\mathsf{W}} \leftarrow y^{\mathsf{R}})[x_1.\mathbf{shift}(x)/\!\!/x] & = & x_1^{\mathsf{W}}.\mathbf{shift}(x^{\mathsf{W}} \leftarrow y^{\mathsf{R}})
\end{array}$$

Extending $[x_1.\mathbf{shift}(x)/\!\!/x]$ compositionally over our other language constructs, we can define $P' = P[x_1.\mathbf{shift}(x)/\!\!/x]$, and so

$$x \Leftarrow P\,;Q \triangleq x_1 \leftarrow P[x_1.\mathbf{shift}(x)/\!\!/x]\,;\mathbf{case}\, x_1^{\mathsf{R}}\ (\mathbf{shift}(x) \Rightarrow Q).$$

We now can use the sequential cut to enforce an order on computation. Of particular interest is the case where we restrict our language so that all cuts are sequential. This gives us a fully sequential language, where we indeed have that only one thread is active at a time. We will make extensive use of this ability to give a fully sequential language, and in Sections 6 to 8, we will add back limited access to concurrency to such a sequential language in order to reconstruct various patterns of concurrent computation.

There are a few properties of the operation $[x_1.\mathbf{shift}(x)/\!\!/x]$ and the sequential cut that we will make use of in our embeddings. Essentially, we would like to know that $P[x_1.\mathbf{shift}(x)/\!\!/x]$ has similar behavior from a typing perspective to P, and that a sequential cut can be typed in a similar manner to a standard concurrent cut. We make this precise with the following lemmas:

Lemma 4. *If* $\Gamma \vdash P :: (x : A_m)$, *then* $\Gamma \vdash P[x_1.\mathbf{shift}(x)/\!\!/x] :: (x_1 : \downarrow_m^m A_m)$.

Lemma 5. *The rule*

$$\frac{(\Gamma_C, \Delta \geq m \geq r) \quad \Gamma_C, \Delta \vdash P :: (x : A_m) \quad \Gamma_C, \Delta', x : A_m \vdash Q :: (z : C_r)}{\Gamma_C, \Delta, \Delta' \vdash (x \Leftarrow P\,; Q) :: (z : C_r)}\ \mathsf{seqcut}$$

is admissible.

Lemma 4 follows from a simple induction on the structure of P, and Lemma 5 can be proven by deriving the seqcut rule using Lemma 4.

In an earlier version of this paper,[6] we developed a separate set of sequential semantics which is bisimilar to the presentation we give here in terms of sequential cuts. However, by embedding the sequential cut into the concurrent semantics as syntactic sugar, we are able to drastically reduce the conceptual and technical overhead needed to look at interactions between the two different frameworks, and simplify our encodings of various concurrency patterns.

Example Revisited: λ-Calculus. If we make all cuts in the λ-calculus interpreter sequential, we obtain a *call-by-value* semantics. In particular, it may no longer compute a weak head-normal form even if it exists. Note that just as we used syntactic sugar for standard

[6] (Pruiksma & Pfenning, 2020), version 1, found at https://arxiv.org/abs/2002.04607

cuts with the identity or call rule on the left, we will also define for convenience

$$\begin{array}{rcl} x \Leftarrow y \,;\, Q & \triangleq & x \Leftarrow (x \leftarrow y) \,;\, Q \\ x \Leftarrow f\,\overline{y} \,;\, Q & \triangleq & x \Leftarrow (x \leftarrow f\,\overline{y}) \,;\, Q \end{array}$$

```
$exp_m = \oplus\{\mathsf{app} : exp \otimes exp, \mathsf{val} : val\}$
$val_m = \oplus\{\mathsf{lam} : val \multimap exp\}$

$(e : \mathsf{exp}) \vdash eval : (v : \mathsf{val})$

$v \leftarrow eval\ e =$
    $\mathbf{case}\ e^{\mathsf{R}}\ (\ \mathsf{val}(v') \Rightarrow v^{\mathsf{W}} \leftarrow v'^{\mathsf{R}}$
              $\mid \mathsf{app}\ \langle e_1, e_2 \rangle \Rightarrow v_1 \Leftarrow eval\ e_1\ ;$
                                       $v_2 \Leftarrow eval\ e_2\ ;$
                                       $\mathbf{case}\ v_1^{\mathsf{R}}\ (\ \mathsf{lam}(f) \Rightarrow e_3 \Leftarrow f^{\mathsf{R}}.\langle v_2, e_3 \rangle\ ;$
                                                                  $v \Leftarrow eval\ e_3\ )\ )$
```

Call-by-name. As mentioned at the beginning of this section, there are multiple approaches to enforcing that only one thread is active at a time. We can think of the sequential cut defined in Section 4 as a form of *call-by-value* — P is fully evaluated before Q can continue. Here, we will define a different sequential cut $x \Leftarrow^N P \,;\, Q$, which will behave more like *call-by-name*, delaying execution of P until Q attempts to read from x. Interestingly, this construct avoids the need for atomic write operations! We nevertheless prefer the "call-by-value" form of sequentiality as our default, as it aligns better with Halstead's approach to futures Halstead (1985), which were defined in a call-by-value language, and also avoids recomputing P if x is used multiple times in Q.

As before, we take advantage of shifts for synchronization, here using an upwards shift rather than a downwards one. If $x : A_m$, we would like to define

$$x \Leftarrow^N P \,;\, Q \triangleq x_1 \leftarrow \mathbf{case}\, x_1^{\mathsf{W}}\ (\mathbf{shift}(x) \Rightarrow P) \,;\, Q',$$

where $x_1 : \uparrow_m^m A_m$ and Q' behaves as Q, except that where Q would read from x, Q' first reads from x_1 and then from x. We can formalize the operation that takes Q to Q' in a similar manner to $[x_1.\mathbf{shift}(x) /\!\!/ x]$. We will call this operation $[x_1.\mathbf{shift}(x)\%x]$, so $Q' = Q[x_1.\mathbf{shift}(x)\%x]$.

$$\begin{array}{rcl} (x^{\mathsf{R}}.V)[x_1.\mathbf{shift}(x)\%x] & = & x \leftarrow x_1^{\mathsf{R}}.\mathbf{shift}(x) \,;\, x^{\mathsf{R}}.V \\ (\mathbf{case}\, x^{\mathsf{R}}\ K)[x_1.\mathbf{shift}(x)\%x] & = & x \leftarrow x_1^{\mathsf{R}}.\mathbf{shift}(x) \,;\, \mathbf{case}\, x^{\mathsf{R}}\ K \\ (y^{\mathsf{W}} \leftarrow x^{\mathsf{R}})[x_1.\mathbf{shift}(x)\%x] & = & x \leftarrow x_1^{\mathsf{R}}.\mathbf{shift}(x) \,;\, y^{\mathsf{W}} \leftarrow x^{\mathsf{R}} \end{array}$$

Note that unlike in our "call-by-value" sequential cut, where we needed to write to two cells atomically, here, the order of reads is enforced because $x_1^{\mathsf{R}}.\mathbf{shift}(x)$ will execute the stored continuation $\mathbf{shift}(x) \Rightarrow P$, which finishes by writing to x. As such, we are guaranteed that Q' is paused waiting to read from x until P finishes executing. Moreover, P is paused within a continuation until Q' reads from x_1, after which it immediately blocks on x, so we maintain only one active thread as desired.

While we will not make much use of this form of sequentiality, we find it interesting that it is so simply encoded, and that the encoding is so similar to that of call-by-value cuts. Both constructions are also quite natural — the main decision that we make is whether to pause P or Q inside a continuation. From this, the rest of the construction follows, as there are two natural places to wake up the paused process — as early as possible or as late as possible. If we wake the paused process P immediately after the cut, as in

$$x_1 \leftarrow \mathbf{case}\, x_1^{\mathsf{W}}\ (\mathbf{shift}(x) \Rightarrow P)\ ; x \leftarrow x_1^{\mathsf{R}}.\mathbf{shift}(x)\ ; Q,$$

the result is a concurrent cut with the extra overhead of the shift. Our sequential cuts are the result of waking the paused process as late as possible — once there is no more work to be done in P in the call-by-value cut, and once Q starts to actually depend on the result of P in the call-by-name cut.

λ-Calculus Example Revisited. We can achieve a sequential interpreter for the λ-calculus with a single use of a by-name cut. This interpreter is then complete: if a weak head-normal form exists, it will compute it. We also recall that this property holds no matter which structural properties we allow for the λ-calculus (e.g., purely linear if the mode allows neither weakening nor contraction, of the λI-calculus if the mode only allows contraction).

```
$v \leftarrow eval\ e =$
    $\mathbf{case}\ e^{\mathsf{R}}\ (\ val(v_1) \Rightarrow v^{\mathsf{W}} \leftarrow v_1^{\mathsf{R}}$
              $\mid app\ \langle e_1, e_2 \rangle \Rightarrow v_1 \Leftarrow eval\ e_1\ ;$
                             $v_2 \Leftarrow^{N} eval\ e_2\ ;$
                             $\mathbf{case}\ v_1^{\mathsf{R}}\ (\ \mathsf{lam}(f) \Rightarrow e_3 \Leftarrow f^{\mathsf{R}}.\langle v_2, e_3 \rangle\ ;$
                                                   $v \Leftarrow eval\ e_3\ )\ )$
```

5 Functions

Rather than presenting an embedding or translation of a full (sequential) functional language into our system, we will focus on the case of functions. There is a standard translation of natural deduction to sequent calculus taking introduction rules to right rules, and constructing elimination rules from cut and left rules. We base our embedding of functions into our language on this translation. By following a similar process with other types, one can similarly embed other functional constructs, such as products and sums.

We will embed functions into an instance of Seax with a single mode m. For this example, we specify $\sigma(m) = \{W, C\}$ in order to model a typical functional language, but we could, for instance, take $\sigma(m) = \{\}$ to model the linear λ-calculus. We also restrict the language at mode m to only have sequential cuts, which will allow us to better model a sequential language. Note that while we only specify one mode here, we could work within a larger mode structure, as long as it contains a suitable mode m at which to implement functions — namely, one with the appropriate structural properties, and where we have the restriction of only having sequential cuts. It is this modularity that allows us to freely combine the various reconstructions presented here and in the following sections. As we are only working within a single mode in this section, we will generally omit mode subscripts, but everything is implicitly at mode m.

Now, to add functions to this language, we begin by adding a new type $A \to B$ and two new constructs — a constructor and a destructor for this type. The constructor, $z^{\mathsf{W}}.(\lambda x.P_\star)$, writes a λ-abstraction to destination z. Here, we write $P_\star$ to denote that the process expression P denotes its destination by $\star$. We will write P_y for $P[y/\star]$. The use of $\star$ makes this closer to the functional style, where the location that the result is returned to is not made explicit. The destructor, $P_\star(Q_\star)$, applies the function $P_\star$ to $Q_\star$. These can be typed using variants of the standard $\rightarrow I$ and $\rightarrow E$ rules labeled with channels:

$$\frac{\Gamma, (x : A) \vdash P_\star :: (\star : B)}{\Gamma \vdash z^{\mathsf{W}}.(\lambda x.P_\star) :: (z : A \rightarrow B)} \rightarrow I \qquad \frac{\Gamma \vdash P_\star :: (\star : A \rightarrow B) \quad \Gamma \vdash Q_\star :: (\star : A)}{\Gamma \vdash y \leftarrow (P_\star(Q_\star)) :: (y : B)} \rightarrow E$$

In order to avoid having to augment our language each time we wish to add a new feature, we will show that these new constructs can be treated as syntactic sugar for terms already in the language, and, moreover, that those terms behave as we would expect of functions and function applications.

We take the following definitions for the new type and terms:

$$\begin{aligned} A \rightarrow B &\triangleq A \multimap B \\ z^{\mathsf{W}}.(\lambda x.P_\star) &\triangleq \textbf{case}\, z^{\mathsf{W}}\ (\langle x, y\rangle \Rightarrow P_y) \\ y \leftarrow (P_\star(Q_\star)) &\triangleq f \Leftarrow P_f; x \Leftarrow Q_x; f^{\mathsf{R}}.\langle x, y\rangle \end{aligned}$$

These definitions are type-correct, as shown by the following theorem:

Theorem 6. *If we expand all new constructs using $\triangleq$, then the typing rules rules $\rightarrow I$ and $\rightarrow E$ above are admissible.*

We can prove this by deriving the typing rules, using Lemma 5 in a few places.

Now that we have established that we can expand this syntactic sugar for functions in a type-correct manner, we examine the evaluation behavior of these terms. First, we consider the lambda abstraction $z^{\mathsf{W}}.(\lambda x.P_\star)$ and its expansion $\textbf{case}\, z^{\mathsf{W}}\ (\langle x, y\rangle \Rightarrow P_y)$. A lambda abstraction should already be a value, and so we might expect that it can be written to memory immediately. Indeed, in the expansion, we immediately write the continuation $(\langle x, y\rangle \Rightarrow P_y)$, which serves as the analogue for $(\lambda x.P_\star)$. This term thus behaves as expected.

We expect that when applying a function $P_\star$ to an argument $Q_\star$, we first reduce $P_\star$ to a value, then reduce $Q_\star$ to a value, and then apply the value of $P_\star$ to the value of $Q_\star$, generally by substitution. In the term $f \Leftarrow P_f; x \Leftarrow Q_x; f^{\mathsf{R}}.\langle x, y\rangle$, we see exactly this behavior. We first evaluate P_f into f, then Q_x into x, and then apply the continuation stored in f to the pair $\langle x, y\rangle$. The addition of y allows us to specify the destination for the result of the function application, as in the *destination-passing style* (Wadler, 1984; Larus, 1989; Cervesato *et al.*, 2002; Simmons, 2012) of semantics for functional languages.

6 Futures

Futures (Halstead, 1985) are a classic example of a primitive to introduce concurrency into a sequential language. In the usual presentation, we add to a (sequential) functional language the ability to create a *future* that immediately returns a *promise* and spawns a concurrent computation. *Touching* a promise by trying to access its value blocks until that 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 various communication primitives.

The development of a sequential cut in Section 4 provides us with ways to model or reconstruct concurrency primitives, and futures are a surprisingly simple example of this. Starting with a language that only allows sequential cuts, we would like to add a new construct that serves to create a future, as we added functions to the base language in Section 5. In this case, however, we already have a construct that behaves exactly as desired. The concurrent cut $x \leftarrow P \,;\, Q$ spawns a new process P, and executes P and Q concurrently. When Q tries to read from x, it will block until P has computed a result W and written it to x. If we wish to add an explicit synchronization point, we can do so with minimal overhead by making use of identity to read from x. For instance, the process $z \Leftarrow (z^{\mathsf{W}} \leftarrow x^{\mathsf{R}}) \,;\, Q$ will first copy or move the contents of cell x to cell z, and then run Q. As such, it delays the execution of Q until x has been written to, even if Q does not need to look at the value of x until later. This is analogous to the touch construct of some approaches to futures.

In other words, in this language, futures, rather than being a construct that we need to add and examine carefully, are in fact the default. This is, in a sense, opposite to the standard approach, where sequentiality is the norm and a new construct is needed to handle futures. By instead adding sequential cut to our otherwise concurrent language, we get the same expressive power, being able to specify whenever we spawn a new computation whether it should be run concurrently with or sequentially before the continuation process.

These futures, much like those in Halstead's Multilisp, are not distinguished at the type level and do not require an explicit touch construct for synchronization, although we can add synchronization points as shown. It is possible to provide an encoding of futures with a distinct type, as they are used in many more modern languages (see Appendix 1), but we find the form presented here more natural, as it allows a great deal of flexibility to the programmer — a process using a variable x does not know and need not care whether the value of x is computed concurrently or not.

One interesting result that arises from this approach to futures, and in particular from the fact that this approach works at any mode m, regardless of what $\sigma(m)$ is, is that by considering the case where $\sigma(m) = \{\}$, we recover a definition of *linear futures*, which must be used exactly once. This is limited in that the base language at mode m will also be linear, along with its futures. However, we are not restricted to working with one mode. For instance, we may take a mode S with $\sigma(\mathsf{S}) = \{\}$, which allows for programming linearly with futures, and a mode S^* with $\sigma(\mathsf{S}^*) = \{\mathsf{W}, \mathsf{C}\}$ and $\mathsf{S} < \mathsf{S}^*$, which allows for standard functional programming. The shifts between the linear and non-linear modes allow both types of futures to be used in the same program, embedding the linear language (including its futures) into the non-linear language via the monad $\uparrow_{\mathsf{S}}^{\mathsf{S}^*} \downarrow_{\mathsf{S}}^{\mathsf{S}^*}$. Uses for linear futures (without a full formalization) in the efficient expression of certain parallel algorithms have already been explored in prior work (Blelloch & Reid-Miller, 1999), but to our knowledge, no formalization of linear futures has yet been given.

Binary Numbers Revisited. The program for *plus2* presented in Section 3.2 is a classic example of a (rather short-lived) pipeline set up with futures. For this to exhibit the expected parallelism, the individual *succ* process should also be concurrent in its recursive call.

```
$(z : bin) \vdash plus2 :: (x : bin)$
$x \leftarrow plus2\ z =$
    $y \leftarrow succ\ z\ ;$
    $x \leftarrow succ\ y$
```

Simple variations (for example, setting up a Boolean circuit on bit streams) follow the same pattern of composition using futures.

mapReduce **Revisited.** As a use of futures, consider making all cuts in *mapReduce* sequential except those representing a recursive call:

```
$(z : \uparrow_m^s B)\ (f : \uparrow_m^u((B \otimes A \otimes B) \multimap B))\ (t : tree_A) \vdash mapReduce_{AB} :: (s : B)$
$s \leftarrow mapReduce_{AB}\ z\ f\ t =$
    $\textbf{case}\ t^{\mathsf{R}}\ \ (\mathsf{empty}\ \langle\ \rangle \Rightarrow z_1 \Leftarrow z^{\mathsf{R}}.\textbf{shift}(z_1)\ ;$          $\%\ drop\ f$
                           $s^{\mathsf{W}} \leftarrow z_1^{\mathsf{R}}$
               $\mid \mathsf{node}\ \langle l, \langle x, r\rangle\rangle \Rightarrow l_1 \leftarrow mapReduce_{AB}\ z\ f\ l\ ;$
                                 $r_1 \leftarrow mapReduce_{AB}\ z\ f\ r\ ;$   $\%\ duplicate\ z\ and\ f$
                                 $p \Leftarrow p^{\mathsf{W}}.\langle l_1, \langle x, r_1\rangle\rangle\ ;$
                                 $s_1 \Leftarrow f^{\mathsf{R}}.\textbf{shift}\langle p, s_1\rangle\ ;$
                                 $s^{\mathsf{W}} \leftarrow s_1^{\mathsf{R}}\ )$
```

In this program, the computation at each node is sequential, but the two recursive calls to *mapReduce* are spawned as futures. We synchronize on these futures when they are needed in the computation of f.

7 Fork/Join Parallelism

While futures allow us a great deal of freedom in writing concurrent programs with fine-grained control, sometimes it is useful to have a more restrictive concurrency primitive, either for implementation reasons or for reasoning about the behavior of programs. Fork/join parallelism is a simple, yet practically highly successful paradigm, allowing multiple independent threads to run in parallel, and then collecting the results together after those threads are finished, using a *join* construct. Many slightly different treatments of fork/join exist. Here, we will take as the primitive construct a parallel pair $\langle P_\star \mid Q_\star \rangle$, which runs $P_\star$ and $Q_\star$ in parallel, and then stores the pair of results. Joining the computation then occurs when the pair is read from, which requires both $P_\star$ and $Q_\star$ to have terminated. This form of fork/join is common in the literature dealing with scheduling and other optimizations for parallelism, particularly *nested* parallelism (e.g. Acar *et al.* (2018)), due to its relative simplicity.

As with our reconstruction of functions in Section 5, we will use a single mode m which may have arbitrary structural properties, but only allows sequential cuts. As we are working with only a single mode, we will generally omit the subscripts that indicate mode, writing A rather than A_m.

We introduce a new type $A_m \| B_m$ of parallel pairs and new terms to create and read from such pairs. We present these terms in the following typing rules:

$$\frac{\Gamma_C, \Gamma \vdash P_\star :: (\star : A) \qquad \Gamma_C, \Delta \vdash Q_\star :: (\star : B)}{\Gamma_C, \Gamma, \Delta \vdash z^{\mathsf{W}}.\langle P_\star \mid Q_\star \rangle :: (z : A \| B)} \; \| R$$

$$\frac{\Gamma, (z : A), (w : B) \vdash R :: (c : C)}{\Gamma, (x : A \| B) \vdash \mathbf{case}\, x^{\mathsf{R}}\; (\langle z \mid w \rangle \Rightarrow R) :: (c : C)} \; \| L$$

As in Section 5 we can reconstruct these types and terms in Seax already. Here, we define:

$$\begin{aligned}
A \| B \quad &\triangleq \quad A \otimes B \\
z^{\mathsf{W}}.\langle P_\star \mid Q_\star \rangle \quad &\triangleq \quad x_1 \leftarrow P_\star[x_1.\mathbf{shift}(x) /\!\!/ \star]\;; \\
&\qquad\;\; y_1 \leftarrow Q_\star[y_1.\mathbf{shift}(y) /\!\!/ \star]\;; \\
&\qquad\;\; \mathbf{case}\, x_1^{\mathsf{R}}\; (\mathbf{shift}(x) \Rightarrow \mathbf{case}\, y_1^{\mathsf{R}}\; (\mathbf{shift}(y) \Rightarrow z^{\mathsf{W}}.\langle x, y \rangle)) \\
\mathbf{case}\, x^{\mathsf{R}}\; (\langle z \mid w \rangle \Rightarrow R) \quad &\triangleq \quad \mathbf{case}\, x^{\mathsf{R}}\; (\langle z, w \rangle \Rightarrow R)
\end{aligned}$$

This definition respects the typing as prescribed by the $\|R$ and $\|L$ rules.

Theorem 7. *If we expand all new constructs using $\triangleq$, then the $\|R$ and $\|L$ rules above are admissible.*

This theorem follows quite straightforwardly from Lemma 4.

The evaluation behavior of these parallel pairs is quite simple — we first observe that, as the derivation of $\|L$ in the theorem above suggests, the reader of a parallel pair behaves exactly as the reader of an ordinary pair. The only difference, then, is in the synchronization behavior of the writer of the pair. Examining the term

$$\begin{aligned}
&x_1 \leftarrow P_\star[x_1.\mathbf{shift}(x) /\!\!/ \star]\;; \\
&y_1 \leftarrow Q_\star[y_1.\mathbf{shift}(y) /\!\!/ \star]\;; \\
&\mathbf{case}\, x_1^{\mathsf{R}}\; (\mathbf{shift}(x) \Rightarrow \mathbf{case}\, y_1^{\mathsf{R}}\; (\mathbf{shift}(y) \Rightarrow z^{\mathsf{W}}.\langle x, y \rangle))
\end{aligned}$$

we see that it spawns two new threads, which run concurrently with the original thread. The new threads execute $P_\star[x_1.\mathbf{shift}(x) /\!\!/ \star]$ and $Q_\star[y_1.\mathbf{shift}(y) /\!\!/ \star]$ with destinations x_1 and y_1, respectively, while the original thread waits first on x_1, then on y_1, before writing the pair $\langle x, y \rangle$ to z. Because the new threads will write to x and x_1 atomically, and similarly for y and y_1, by the time $\langle x, y \rangle$ is written to z, x and y must have already been written to. However, because both cuts in this term are concurrent cuts, $P_\star$ and $Q_\star$ run concurrently, as we expect from a parallel pair.

mapReduce **Revisited.** We can use the fork/join pattern in the implementation of *mapReduce* so that we first synchronize on the results returned from the two recursive calls before we call f on them.

$$(z : \uparrow^s_m B)\; (f : \uparrow^u_m((B \otimes A \otimes B) \multimap B))\; (t : tree_A) \vdash mapReduce_{AB} :: (s : B)$$

```
$s \leftarrow \mathit{mapReduce}_{AB}\ z\ f\ t =$
      $\textbf{case}\ t^{\mathsf{R}}\ \ (\ \mathsf{empty}\ \langle\,\rangle \Rightarrow z_1 \Leftarrow z^{\mathsf{R}}.\textbf{shift}(z_1)$
                                    $s^{\mathsf{W}} \leftarrow z_1^{\mathsf{R}}$
                $\mid \mathsf{node}\ \langle l, \langle x, r \rangle\rangle \Rightarrow$
                    $rl \leftarrow rl^{\mathsf{W}}.\langle \mathit{mapReduce}_{AB}\ x\ f\ l \mid \mathit{mapReduce}_{AB}\ x\ f\ r \rangle\ ;$
                    $\textbf{case}\ rl^{\mathsf{R}}\ (\langle l_1 \mid r_1 \rangle \Rightarrow p \Leftarrow p^{\mathsf{W}}.\langle l_1, \langle x, r_1 \rangle\rangle\ ;$
                                          $s_1 \Leftarrow f^{\mathsf{R}}.\textbf{shift}\langle p, s_1 \rangle\ ;$
                                          $s^{\mathsf{W}} \leftarrow s_1^{\mathsf{R}}\ )\ )$
```

8 Monadic Concurrency

For a different type of concurrency primitive, we look at a monad for concurrency, taking some inspiration from SILL (Toninho *et al.*, 2013; Toninho, 2015; Griffith, 2016), which makes use of a contextual monad to embed the concurrency primitives of linear session types into a functional language. This allows us to have both a fully-featured sequential functional language and a fully-featured concurrent linear language, with the concurrent layer able to refer on variables in the sequential layer, but not the other way around. By keeping the layers separate in this way, we can reason about them independently. Moreover, the sequential layer could be implemented more simply than the concurrent layer — while the concurrent layer needs some form of locking or synchronization to ensure that a cell is not read from until it has been written to, the sequential layer can avoid all of this overhead. Similarly, while in the sequential layer, an implementation could avoid the extra work of thread management by maintaining a single thread.

To construct this concurrency monad, we will use two modes N and S with $\mathsf{N} < \mathsf{S}$. Intuitively, the linear concurrent portion of the language is at mode N, while the functional portion is at mode S. As in common in functional languages, S allows weakening and contraction ($\sigma(\mathsf{S}) = \{W, C\}$), but only permits sequential cuts (by which we mean that any cut whose principal formula is at mode S must be a sequential cut) so that it models a sequential functional language. By contrast, N allows concurrent cuts, but is linear ($\sigma(\mathsf{S}) = \{\}$). We will write A_{S} and A_{N} for sequential and concurrent types, respectively.

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

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

Since they live in the session-typed layer, the $\wedge$ and $\supset$ constructs fit more straightforwardly into our language. We will focus on the type $A_{\mathsf{S}} \wedge B_{\mathsf{N}}$, but $A_{\mathsf{S}} \supset B_{\mathsf{N}}$ can be handled similarly. A process of type $A_{\mathsf{S}} \wedge B_{\mathsf{N}}$ should write a pair of a functional value with type A_{S}

and a concurrent value with type B_{N}. These terms and their typing rules are shown below:

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

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

To bring these new constructs into the base language, we define

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

These definitions give us the usual type-correctness theorem:

Theorem 8. *If we expand all new constructs using $\triangleq$, then the typing rules for $\{\cdot\}$ and $\wedge$ are admissible.*

As with the previous sections, it is not enough to know that these definitions are well-typed — we would also like to verify that they have the behavior we expect for a concurrency monad. In both cases, this is relatively straightforward. Examining the term

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

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

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

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

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

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

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

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

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

These terms therefore capture the general behavior of a monad used to encapsulate concurrency inside a functional language. The details of the monad we present here are different from that of SILL's (contextual) monad, despite our use of similar notation, but the essential idea is the same.

Example: A Concurrent Counter. We continue our example of binary numbers, this time supposing that the mode $m = \mathsf{S}$, that is, our numbers and the successor function on them are sequential and allow weakening and contraction. *counter* represents a concurrently running process that can receive inc and val messages to increment or retrieve the counter value, respectively.

$$ctr_{\mathsf{N}} = \&_{\mathsf{N}}\{\mathsf{inc} : ctr_{\mathsf{N}}, \mathsf{val} : bin_{\mathsf{S}} \wedge ctr_{\mathsf{N}}\}$$

$$(x : bin_{\mathsf{S}}) \vdash counter :: (c : ctr_{\mathsf{N}})$$

```
$c \leftarrow counter\ x =$
    $\textbf{case}\ c\ \ (\ \mathsf{inc}(c_1) \Rightarrow x_1 \Leftarrow succ\ x\ ;$
                           $c_1 \leftarrow counter\ x_1$
             $\mid \mathsf{val}(c_1) \Rightarrow c_2 \leftarrow counter\ x\ ;$
                           $c_1^{\mathsf{W}}.\langle x, c_2\rangle$
```

9 Conclusion

We have presented a concurrent shared-memory semantics based on a semi-axiomatic (DeYoung *et al.*, 2020) presentation of adjoint logic (Reed, 2009; Licata & Shulman, 2016; Licata *et al.*, 2017; Pruiksma & Pfenning, 2019), for which we have usual variants of progress and preservation, as well as confluence. We then demonstrate that by adding a limited form of atomic writes, we can model *sequential* computation. Taking advantage of this, we reconstruct several patterns that provide limited access to concurrency in a sequential language, such as fork/join, futures, and monadic concurrency in the style of SILL. The uniform nature of these reconstructions means that they are all mutually compatible, and so we can freely work with any set of these concurrency primitives within the same language. Moreover, taking advantage of the adjoint nature of the language, we can have multiple modes, each with different features — for instance, one mode where computation is purely sequential, another with futures, and yet another with fork/join. The separation between these modes means that we can reason about programs at each mode separately — not needing to think about concurrency at the purely sequential mode, for example. Building on this, an actual implementation of this language could make optimizations based on the restrictions at each mode, not needing to worry about the full range of features that may exist at other modes. Seax therefore allows us to get many of the benefits of working in a restricted language (at a specific mode), without the drawbacks of only having specific tools to work with (since we can weaken those restrictions or place other restrictions at different modes).

There are several potential directions that future work in this space could take. In our reconstruction of futures, we incidentally also provide a definition of *linear* futures, which have been used in designing pipelines (Blelloch & Reid-Miller, 1999), but to our knowledge have not been examined formally or implemented. One item of future work, then, would be to further explore linear futures, now aided by a formal definition which is also amenable to implementation. We also believe that it would be interesting to explore an implementation of our language as a whole, and to investigate what other concurrency patterns arise naturally when working in it. Another item of future work is to make more precise the correctness of the encodings we describe in Sections 5, 7 and 8. For instance, for functions, we can prove beta reduction admissible already, but for the other encodings, we lack similar results, as this kind of functional correctness result appears to require a better notion of equivalence for Seax processes, allowing us to compare terms in the languages augmented with additional constructs to terms in the base language that use encodings in place of those additional constructs. Additionally, the stratification of the language into layers connected with adjoint operators strongly suggests that some properties of a language instance as a whole can be obtained modularly from properties of the sublanguages at each mode. Although based on different primitives, research on monads and comonads to capture effects and coeffects, respectively (Curien *et al.*, 2016; Gaboardi *et al.*, 2016), also points in this direction. In particular, we would like to explore a modular theory of (observational) equivalence using this approach. Some work on observational equivalence in a substructural setting already exists (Kavanagh, 2020), but works in a message-passing setting and does not seem to translate directly to the shared-memory setting of Seax.

Conflicts of Interest: None.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 1718267.

References

Acar, U. A., Charguéraud, A., Guatto, A., Rainey, M. and Sieczkowski, F. (2018) Heartbeat scheduling: Provable efficiency for nested parallelism. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* pp. 769–782.

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

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

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

Cervesato, I., Hodas, J. S. and Pfenning, F. (2000) Efficient resource management for linear logic proof search. *Theoretical Computer Science* **232**(1–2):133–163. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

Cervesato, I., Pfenning, F., Walker, D. and Watkins, K. (2002) *A Concurrent Logical Framework II: Examples and Applications*. Tech. rept. CMU-CS-02-102. Department of Computer Science, Carnegie Mellon University. Revised May 2003.

Church, A. and Rosser, J. (1936) Some properties of conversion. *Transactions of the American Mathematical Society* **39**(3):472–482.

Conway, M. E. (1963) A multiprocessor system design. *Proceedings of the Fall Joint Computer Conference (AFIPS'63)* pp. 139–146. ACM.

Curien, P.-L., Fiore, M. P. and Munch-Maccagnoni, G. (2016) A theory of effects and resources: Adjunction models and polarised calculi. Bodík, R. and Majumdar, R. (eds), *Proceedings of the 43rd Symposium on Principles of Programming Languages (POPL 2016)* pp. 44–56. ACM.

Das, A. and Pfenning, F. (2020) Rast: Resource-aware session types with arithmetic refinements. Ariola, Z. (ed), *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)* pp. 33:1–33:17. LIPIcs 167. System description.

DeYoung, H., Pfenning, F. and Pruiksma, K. (2020) Semi-axiomatic sequent calculus. Ariola, Z. (ed), *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)* pp. 29:1–29:22. LIPIcs 167.

Gaboardi, M., ya Katsumata, S., Orchard, D., Breuvart, F. and Uustalu, T. (2016) Combining effects and coeffects via grading. Garrigue, J., Keller, G. and Sumii, E. (eds), *21st International Conference on Functional Programming (ICFP 2016)* pp. 476–489. ACM, Nara, Japan.

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

Gay, S. J. and Vasconcelos, V. T. (2010) Linear type theory for asynchronous session types. *Journal of Functional Programming* **20**(1):19–50.

Gentzen, G. (1935) Untersuchungen über das logische Schließen. *Mathematische Zeitschrift* **39**:176–210, 405–431. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Girard, J.-Y. and Lafont, Y. (1987) Linear logic and lazy computation. Ehrig, H., Kowalski, R., Levi, G. and 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.

Griffith, D. (2016) *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign.

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

Hilbert, D. and Bernays, P. (1934) *Grundlagen der Mathematik*. Springer-Verlag.

Honda, K. (1993) Types for dyadic interaction. Best, E. (ed), *4th International Conference on Concurrency Theory (CONCUR 1993)* pp. 509–523. Springer LNCS 715.

Honda, K., Vasconcelos, V. T. and Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. Hankin, C. (ed), *7th European Symposium on Programming Languages and Systems (ESOP 1998)* pp. 122–138. Springer LNCS 1381.

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

Larus, J. R. (1989) *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, Berkeley.

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

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

Lincoln, P. and Mitchell, J. C. (1992) Operational aspects of linear lambda calculus. *7th Annual Symposium on Logic in Computer Science (LICS 1992)* pp. 235–246. IEEE, Santa Cruz, California.

Miller, H., Haller, P., Müller, N. and Boullier, J. (2016) Function passing: A model for typed, distributed functional programming. Visser, E., Murphy-Hill, E. and Lopes, C. (eds), *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)* pp. 82–97. ACM.

Pruiksma, K. and Pfenning, F. (2019) A message-passing interpretation of adjoint logic. Martins, F. and Orchard, D. (eds), *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)* pp. 60–79. EPTCS 291.

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

Pruiksma, K., Chargin, W., Pfenning, F. and Reed, J. (2018) *Adjoint Logic*. Unpublished manuscript.

Reed, J. (2009) *A Judgmental Deconstruction of Modal Logic*. Unpublished manuscript.

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

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

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

Wadler, P. (1984) Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-times. *Conference on Lisp and Functional Programming (LFP 1984)* pp. 45–52. ACM, Austin, Texas.

1 Typed Futures

The futures that we discuss in Section 6 behave much like Halstead's original futures in Multilisp Halstead (1985), which, rather than being distinguished at the type level, are purely operational. One side effect of this is that while we can explicitly synchronize these futures, we can also make use of implicit synchronization, where accessing the value of the future blocks until it has been computed, without the need for a touch construct.

Here, we will look at a different encoding of futures, which distinguishes futures at the type level, as they have often been presented since. As in Section 5, we will work with a single mode m, in which we will only allow sequential cuts, and which may have any set $\sigma(m)$ of structural properties. To the base language, we add the following new types and process terms for futures:

$$\begin{array}{llll}
\text{Types} & A & ::= & \ldots \mid \mathbf{fut}\, A \\
\text{Processes} & P & ::= & \ldots \mid x^{\mathsf{W}}.\langle P_\star \rangle \mid \mathbf{touch}\, y^{\mathsf{R}}\ (\langle z \rangle \Rightarrow P)
\end{array}$$

We type these new constructs as:

$$\frac{\Gamma \vdash P_\star :: (\star : A_m)}{\Gamma \vdash x^{\mathsf{W}}_m.\langle P_\star \rangle :: (x_m : \mathbf{fut}\, A_m)}\ \mathbf{fut}R$$

$$\frac{\Gamma, z_m : A_m \vdash Q :: (w_m : C_m)}{\Gamma, x_m : \mathbf{fut}\, A_m \vdash \mathbf{touch}\, x^{\mathsf{R}}_m\ (\langle z_m \rangle \Rightarrow Q) :: (w_m : C_m)}\ \mathbf{fut}L$$

We then reconstruct this in Seax by defining

$$\begin{array}{lcl}
\mathbf{fut}\, A_m & \triangleq & \downarrow^m_m \downarrow^m_m A_m \\
x^{\mathsf{W}}_m.\langle P_\star \rangle & \triangleq & y_m \leftarrow P_\star[y_m.\mathbf{shift}(z_m) /\!\!/ \star]\ ;\ x^{\mathsf{W}}_m.\mathbf{shift}(y_m) \\
\mathbf{touch}\, x^{\mathsf{R}}_m\ (\langle z_m \rangle \Rightarrow Q) & \triangleq & \mathbf{case}\, x^{\mathsf{R}}_m\ (\mathbf{shift}(y_m) \Rightarrow \mathbf{case}\, y^{\mathsf{R}}_m(\mathbf{shift}(z_m) \Rightarrow Q))
\end{array}$$

This is not the only possible reconstruction, [7] but we use it because it is the simplest one that we have found. The first property to verify is that these definitions are type-correct:

Theorem 9. *If we expand all new constructs using* $\triangleq$*, then the rules* $\mathbf{fut}L$ *and* $\mathbf{fut}R$ *are admissible.*

Proof By examining typing derivations for these processes, we see that these rules can be derived as follows:

$$\dfrac{\dfrac{\Gamma \vdash P_\star :: (\star :: A_m)}{\Gamma \vdash P_\star[y_m.\mathbf{shift}(z_m) /\!\!/ \star] :: (y_m : \downarrow^m_m A_m)}\ \textit{Lemma } 4 \qquad \dfrac{}{y_m : \downarrow^m_m A_m \vdash x^{\mathsf{W}}_m.\mathbf{shift}(y_m) :: (x_m : \downarrow^m_m \downarrow^m_m A_m)}\ \downarrow R^0}{\Gamma \vdash y_m \leftarrow P_\star[y_m.\mathbf{shift}(z_m) /\!\!/ \star]\ ;\ x^{\mathsf{W}}_m.\mathbf{shift}(y_m) :: (x_m : \downarrow^m_m \downarrow^m_m A_m)}\ \mathsf{cut}$$

[7] In particular, as the role of the outer shift is simply to allow the client of the future to proceed, we can replace the shift with any other type that forces a send but does not provide any useful information. Examples include $\uparrow^m_m \downarrow^m_m A_m$ and $\mathbf{1}_m \otimes (\downarrow^m_m A_m)$.

$$\dfrac{\dfrac{\Gamma, z_m : A_m \vdash Q :: (w : C_m)}{\Gamma, y_m : {\downarrow}^m_m A_m \vdash \mathbf{case}\; y^{\mathsf{R}}_m(\mathbf{shift}(z_m) \Rightarrow Q) :: (w : C_m)}\; {\downarrow}L_0}{\Gamma, x_m : {\downarrow}^m_m{\downarrow}^m_m A_m \vdash \mathbf{case}\; x^{\mathsf{R}}_m\; (\mathbf{shift}(y_m) \Rightarrow \mathbf{case}\; y^{\mathsf{R}}_m(\mathbf{shift}(z_m) \Rightarrow Q)) :: (w : C_m)}\; {\downarrow}L_0$$

Note that we omit mode conditions on cut because within a single mode m, they are necessarily satisfied. ■

Now, we examine the computational behavior of these terms to demonstrate that they behave as futures. The type ${\downarrow}^m_m A_m$, much like in Section 4 where we used it to model sequentiality, adds an extra synchronization point. Here, we shift twice, giving ${\downarrow}^m_m{\downarrow}^m_m A_m$, to introduce two synchronization points. The first is that enforced by our restriction to only allow sequential cuts in this language (outside of futures), while the second will become the **touch** construct. We will see both of these when we examine each process term.

We begin by examining the constructor for futures. Intuitively, when creating a future, we would like to spawn a new thread to evaluate $P_\star$ with new destination z_m, and immediately write the promise of z_m (represented by a hypothetical new value $\langle z_m \rangle$) into x_m, so that any process waiting on x_m can immediately proceed. The term

$$x^{\mathsf{W}}_m.\langle P_\star \rangle \quad \triangleq \quad y^{\mathsf{W}}_m \Leftarrow P_\star[y_m.\mathbf{shift}(z_m) /\!\!/ \star] \,;\, x^{\mathsf{W}}_m.\mathbf{shift}(y_m)$$

behaves almost exactly as expected. Rather than spawning $P_\star$ with destination z_m, we spawn $P_\star[y_m.\mathbf{shift}(z_m) /\!\!/ \star]$, which will write the result of $P_\star$ to z_m, and a synchronizing shift to y_m. Concurrently, we write the value $\mathbf{shift}(y_m)$ to x_m, allowing the client of x_m to resume execution, even if x_m was created by a sequential cut. This value $\mathbf{shift}(y_m)$ is the first half of the promise $\langle z_m \rangle$, and the second half, $\mathbf{shift}(z_m)$, will be written to y_m when P finishes executing.

If, while P continues to execute, we touch x_m, we would expect to block until the promise $\langle z_m \rangle$ has been fulfilled by P having written to z_m. Again, we see exactly this behavior from the term

$$\mathbf{touch}\; x^{\mathsf{R}}_m\; (\langle z_m \rangle \Rightarrow Q) \quad \triangleq \quad \mathbf{case}\; x^{\mathsf{R}}_m\; (\mathbf{shift}(y_m) \Rightarrow \mathbf{case}\; y^{\mathsf{R}}_m(\mathbf{shift}(z_m) \Rightarrow Q)).$$

This process will successfully read $\mathbf{shift}(y_m)$ from x_m, but will block trying to read from y_m until y_m is written to. Since z_m and y_m are written to at the same time, we block until z_m is written to, at which point the promise is fulfilled. Once a result W has been written to z_m and (simultaneously) $\mathbf{shift}(z_m)$ has been written to y_m, this process can continue, reading both y_m and z_m, and continuing as Q. Again, this is the behavior we expect a touch construct to have.

This approach does effectively model a form of typed future, which ensures that all synchronization is explicit, but comes at the cost of overhead from the additional shifts. Both this and the simpler futures that we describe in Section 6 have their uses, but we believe that the futures native to Seax are more intuitive in general.

2 Proofs of Type Correctness

In Sections 5, 7 and 8, we present type-correctness theorems for our reconstructions of various concurrency primitives, but omit the details of the proofs. Here, we present those details.

Functions. We derive the typing rules as follows, making use of Lemma 5 to use the admissible seqcut rule. We omit the conditions on modes for cut, as we only have one mode:

$$\dfrac{\Gamma, x : A \vdash P_y :: (y : B)}{\Gamma \vdash \mathbf{case}\ z\ (\langle x, y\rangle \Rightarrow P_y) :: (z : A \multimap B)}\ {\multimap}R$$

$$\dfrac{\Gamma \vdash P_f :: (f : A \multimap B) \qquad \dfrac{\Gamma \vdash Q_x :: (x : A) \quad \dfrac{}{\Gamma, (f : A \multimap B), (x : A) \vdash f.\langle x, y\rangle :: (y : B)}\ {\multimap}L^0}{\Gamma, (f : A \multimap B) \vdash x \Leftarrow Q_x; f.\langle x, y\rangle :: (y : B)}\ \mathsf{seqcut}}{\Gamma \vdash f \Leftarrow P_f; x \Leftarrow Q_x; f.\langle x, y\rangle :: (y : B)}\ \mathsf{seqcut}$$

Fork/Join. Due to the length of the process term that defines $z.\langle P_\star \mid Q_\star\rangle$, we elide portions of it throughout the derivation below, and we will write P' for $P_\star[x'.\mathbf{shift}(x) /\!\!/ \star]$, and similarly Q' for $Q_\star[y'.\mathbf{shift}(y) /\!\!/ \star]$. With these abbreviations, we have the following derivation for the $\|R$ rule, where the dashed inferences are made via Lemma 4.

$$\dfrac{\dfrac{\Gamma_C, \Gamma \vdash P_x :: (x : A)}{\Gamma_C, \Gamma \vdash P' :: (x' : {\downarrow}^m_m A)} \qquad \dfrac{\dfrac{\Gamma_C, \Delta \vdash Q_y :: (y : B)}{\Gamma_C, \Delta \vdash Q' :: (y' : {\downarrow}^m_m B)} \quad \dfrac{\dfrac{\dfrac{}{x : A, y : B \vdash z.\langle x, y\rangle :: (z : A \otimes B)}\ {\otimes}R^0}{x : A, y' : {\downarrow}^m_m B \vdash \mathbf{case}\ y'\ (\ldots) :: (z : A \otimes B)}\ {\downarrow}L_0}{x' : {\downarrow}^m_m A, y' : {\downarrow}^m_m B \vdash \mathbf{case}\ x'\ (\ldots) :: (z : A \otimes B)}\ {\downarrow}L_0}{\Gamma_C, \Delta, x' : {\downarrow}^m_m A \vdash y' \leftarrow Q'\ ; \ldots :: (z : A \otimes B)}\ \mathsf{cut}}{\Gamma_C, \Gamma, \Delta \vdash x' \leftarrow P'\ ; \ldots :: (z : A \otimes B)}\ \mathsf{cut}$$

The left rule is much more straightforward, since this encoding makes the writer of the pair rather than the reader responsible for synchronization.

$$\dfrac{\Gamma, z : A, w : B \vdash R :: (c : C)}{\Gamma, x : A \otimes B \vdash \mathbf{case}\ x\ (\langle z, w\rangle \Rightarrow R) :: (c : C)}\ {\otimes}L_0$$

Monadic Concurrency. We first construct the typing rules for $\{\cdot\}$, which are straightforward:

$$\dfrac{\Gamma_{\mathsf{S}} \vdash P_x :: (x_{\mathsf{N}} : A_{\mathsf{N}})}{\Gamma_{\mathsf{S}} \vdash \mathbf{case}\ d_{\mathsf{S}}\ (\mathbf{shift}(x_{\mathsf{N}}) \Rightarrow P_x) :: (d_{\mathsf{S}} : {\uparrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{N}})}\ {\uparrow}R$$

$$\dfrac{\Gamma_{\mathsf{S}} \geq \mathsf{S} \geq \mathsf{N} \quad \Gamma_{\mathsf{S}} \vdash P_y :: (y_{\mathsf{S}} : {\uparrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{N}}) \quad \dfrac{}{(y_{\mathsf{S}} : {\uparrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{N}}) \vdash y_{\mathsf{S}}.\mathbf{shift}(c_{\mathsf{N}}) :: (c_{\mathsf{N}} : A_{\mathsf{N}})}\ {\uparrow}L^0}{\Gamma_{\mathsf{S}} \vdash y_{\mathsf{S}} \Leftarrow P_y; y_{\mathsf{S}}.\mathbf{shift}(c_{\mathsf{N}}) :: (c_{\mathsf{N}} : A_{\mathsf{N}})}\ \mathsf{seqcut}$$

We then construct the typing rules for $\wedge$:

$$\dfrac{\dfrac{}{v_{\mathsf{S}} : A_{\mathsf{S}} \vdash x_{\mathsf{N}}.\mathbf{shift}(v_{\mathsf{S}}) :: (x_{\mathsf{N}} :: {\downarrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{S}})}{\downarrow}R^0 \quad \dfrac{}{\Gamma_W, (x_{\mathsf{N}} :: {\downarrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{S}}) \vdash d_{\mathsf{N}}.\langle x_{\mathsf{N}}, y_{\mathsf{N}}\rangle :: (d_{\mathsf{N}} :: ({\downarrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{S}}) \otimes B_{\mathsf{N}})}\otimes R^0}{\Gamma_W, (v_{\mathsf{S}} : A_{\mathsf{S}}), (y_{\mathsf{N}} : B_{\mathsf{N}}) \vdash x_{\mathsf{N}} \leftarrow x_{\mathsf{N}}.\mathbf{shift}(v_{\mathsf{S}}); d_{\mathsf{N}}.\langle x_{\mathsf{N}}, y_{\mathsf{N}}\rangle :: (d_{\mathsf{N}} :: A_{\mathsf{S}} \wedge B_{\mathsf{N}})}\mathsf{cut}$$

$$\dfrac{\dfrac{\Gamma, (v_{\mathsf{S}} : A_{\mathsf{S}}), (w_{\mathsf{N}} : B_{\mathsf{N}}) \vdash P_z :: (z_{\mathsf{N}} : C_{\mathsf{N}})}{\Gamma, (x_{\mathsf{N}} : {\downarrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{S}}), (w_{\mathsf{N}} : B_{\mathsf{N}}) \vdash \mathbf{case}\, x_{\mathsf{N}}(\mathbf{shift}(v_{\mathsf{S}}) \Rightarrow P_z) :: (z_{\mathsf{N}} : C_{\mathsf{N}})}{\downarrow}L_0}{\Gamma, (d_{\mathsf{N}} : ({\downarrow}^{\mathsf{S}}_{\mathsf{N}} A_{\mathsf{S}}) \otimes B_{\mathsf{N}}) \vdash \mathbf{case}\, d_{\mathsf{N}}\ (\langle x_{\mathsf{N}}, w_{\mathsf{N}}\rangle \Rightarrow \mathbf{case}\, x_{\mathsf{N}}(\mathbf{shift}(v_{\mathsf{S}}) \Rightarrow P_z)) :: (z_{\mathsf{N}} : C_{\mathsf{N}})}\otimes L_0$$

Note that unlike the rules for $\{\cdot\}$ or for many of the constructs in previous sections, those for $\wedge$ are not only admissible — they are *derivable*.