# Lecture Notes on Futures

15-836: Substructural Logics
Frank Pfenning

Lecture 16
November 2, 2023

## 1 Introduction

In many ways the border between message passing and shared memory concurrency is fluid. We can think of a message passing language as implemented using shared memory, or shared memory representing messages passed between threads. So far, we have taken the message passing view of communication, we will now take the shared memory view.

Shared memory comes in several forms. We strive to find the right level of abstraction to retain the close connection to logic and also illuminate the correspondence to message passing. It turns out that *futures* [Halstead, 1985] are the perfect fit. They were first developed for Lisp, a dynamically typed language, but are entirely compatible with static typing [Pruiksma and Pfenning, 2022, Somayyajula and Pfenning, 2022, 2023].

What are futures? Consider the construct

$$\textbf{let } x = \textbf{future } e_1 \textbf{ in } e_2(x)$$

in a functional language. The idea is the **future** $e_1$ immediately returns a *promise* $p$. Then we evaluate $e_1$ and $e_2(p)$ in parallel. If evaluation of $e_2(p)$ requires the value of $p$ it blocks until the evaluation of $e_1$ has fulfilled the promise by providing a value and $e_2(p)$ can continue.

The analogy with message passing should be clear: a promise acts as a channel of communication between $e_1$and $e_2$. We think of the future as being a designated shared memory location where the value of the promise can eventually be found. This point of view has several advantages. For one, it is quite close to an implementation. For another, it quite naturally lends itself to a *sequential* implementation which is less apparent under message passing. Finally, it allows us to investigate, formally, the connection to message passing [Pfenning and Pruiksma, 2023].

While futures aren't intrinsically substructural (and certainly weren't conceived as such), it turns out that a substructural version has been proposed [Blelloch and Reid-Miller, 1999] and can have advantages in asymptotic complexity over nonlinear ones. Our development in this lecture starts with the linear version and then generalizes it by adding structural types.

This form of shared memory of *write-once* shared memory: once written, it can be read by multiple consumers but it can not be modified. Allowing this would require an imperative language with mutable shared memory. Such languages (or libraries in imperative host languages) certainly exist (including, for example, Halstead's original Multilisp) and programs in them are subject to reasoning via external means. For example, we may want to reason about programs in Rust using concurrent separation logic [Brookes, 2007, O'Hearn, 2007, Jung et al., 2018], a substructural logic in a different mold from the ones we have been discussing. In this case the programming language and logic are not related by a proofs-as-programs correspondence.

## 2 Reinterpreting SAX: Positive Types

The fundamental idea is that in a sequent each variable stands for a memory address. A process $P$ reads from the addresses among the antecedents and writes to the address labeling the succedent.

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{read}} \vdash P :: \underbrace{(x : A)}_{\text{write}}$$

If everything is linear the process $P$ should definitely read from all the $x_i$ and write to $x$. However, in the presence of recursion the mere type system does not guarantee that and we need some additional reasoning [Somayyajula and Pfenning, 2022].

Under message passing, the type $A$ described the type of message exchanged. Here, it describes the contents of the memory cell. What was a continuation channel now becomes an address of further data. Cut now allocates a new shared cell, while the identity moves the contents of one cell to another. We first focus on posi-

tive types.

$$
\begin{array}{llcll}
\text{Values} & V & ::= & k(x) & (\oplus) \\
 & & \mid & (x_1, x_2) & (\otimes) \\
 & & \mid & (\,) & (\mathbf{1}) \\
\text{Continuations} & K & ::= & (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} & (\oplus) \\
 & & \mid & ((x_1, x_2) \Rightarrow P(x_1, x_2)) & (\otimes) \\
 & & \mid & ((\,) \Rightarrow P) & (\mathbf{1}) \\
\text{Processes} & P & ::= & x \leftarrow P(x)\ ;\ Q(x) & \mathsf{cut} \\
 & & \mid & \mathbf{move}\ x\ y & \mathsf{id} \\
 & & \mid & \mathbf{write}\ x\ V & \\
 & & \mid & \mathbf{read}\ x\ K & \\
 & & \mid & \mathbf{call}\ p\ x\ y_1 \ldots y_n &
\end{array}
$$

At runtime, we think of tagged value such as $k(a)$ as a pair consisting of a tag $k$ and an address $a$, a value $(a_1, a_2)$ as a pair of addresses $a_1$ and $a_2$, and $(\,)$ as a unit value. Continuations branch based on a value read from memory.

We have replaced **send** and **recv** with **read** and **write**. Also, instead of forwarding between channels we move the contents of one memory location to another.

Both statics and dynamics for the positive types are straightforward.

$$\frac{k \in L}{y : A_k \vdash \mathbf{write}\ x\ k(y) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus X$$

$$\frac{\Delta, y : A_\ell \vdash Q_\ell(y) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ k\ (\ell(y) \Rightarrow Q_\ell(y))_{\ell \in L} :: (z : C)}\ \oplus L$$

$$\frac{}{x_1 : A, x_2 : B \vdash \mathbf{write}\ x\ (x_1, x_2) :: (x : A \otimes B)}\ \otimes X$$

$$\frac{\Delta, x_1 : A, x_2 : B \vdash Q(x_1, x_2) :: \delta}{\Delta, x_1 : A \otimes B \vdash \mathbf{recv}\ x\ ((x_1, x_2) \Rightarrow Q(x_1, x_2)) :: \delta}\ \otimes L$$

$$\frac{}{\cdot \vdash \mathbf{write}\ x\ (\,) :: (x : \mathbf{1})}\ \mathbf{1}X \qquad \frac{\Delta \vdash Q :: \delta}{\Delta, x : \mathbf{1} \vdash \mathbf{read}\ x\ ((\,) \Rightarrow Q) :: \delta}\ \mathbf{1}L$$

Cut and identity do not change from the sequent calculus.

$$\frac{}{y : A \vdash \mathbf{move}\ x\ y :: (x : A)}\ \mathsf{id} \qquad \frac{\Delta \vdash P(x) :: (x : A) \quad \Delta', x : A \vdash Q(x) :: \delta}{\Delta, \Delta' \vdash x_A \leftarrow P(x)\ ;\ Q(x) :: \delta}\ \mathsf{cut}$$

The dynamics relies on the $V \triangleright K$ operation carried over from the message passing setting. However, we differentiate memory cells at address $a$ containing value $V$,

written $\mathsf{cell}(a, V)$, from processes. This will give us properties such as: a configuration is *final* if it contains only memory cells and no processes.

$$
\begin{array}{lcll}
\mathsf{proc}(x \leftarrow P(x)\,;\,Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) & (a \text{ fresh}) \\
\mathsf{cell}(b, V), \mathsf{proc}(\mathbf{move}\ a\ b) & \longrightarrow & \mathsf{cell}(a, V) & \\
\mathsf{proc}(\mathbf{write}\ a\ V) & \longrightarrow & \mathsf{cell}(a, V) & \\
\mathsf{cell}(a, V), \mathsf{proc}(\mathbf{read}\ a\ K) & \longrightarrow & \mathsf{proc}(V \triangleright K) & \\
\mathsf{proc}(\mathbf{call}\ p\ a\ b_1 \ldots b_n) & \longrightarrow & \mathsf{proc}(P(a, b_1, \ldots, b_n)) & \\
 & & \text{for } p\ x\ y_1 \ldots y_n = P(x, y_1, \ldots, y_n) \in \Sigma &
\end{array}
$$

$$
\begin{array}{lcll}
k(a) \triangleright (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} & = & P_k(a) & (k \in L) \\
(a_1, a_2) \triangleright ((x_1, x_2) \Rightarrow P(x_1, x_2)) & = & P(a_1, a_2) & \\
(\,) \triangleright ((\,) \Rightarrow P) & = & P &
\end{array}
$$

Here is a simple program we can write already, reversing a list.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'cons : bin * list, 'nil : 1}

proc rev (R : list) (L : list) (K : list) =
   read L ( 'cons(p) => read p ((x,L') =>
               p' : bin * list <- write p' (x,K) ;
               K' : list <- write K' 'cons(p') ;
               call R L' K')
          | 'nil(u) => read u (() =>
               move R K) )

proc reverse (R : list) (L : list) =
   u : 1 <- write u () ;
   K : list <- write K 'nil() ;
   call rev R L K
```

Using the equivalent of message sequences, this could be more compact—something we'll get back to in the next lecture.

## 3 Reinterpreting SAX: Negative Types

So far, things worked out as one might expect: on positive types, receives become reads and sends become writes. Negative types present a surprise because *every action on the succedent is a write!* This means that cells no longer just contain small values $V$, but they also have to contain continuations. We will shortly write this out. But first the rules: right rules write, left rules (even in the form of axioms)

read.

$$\frac{\Delta \vdash P_\ell(y) :: (y : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \mathbf{write}\ x\ (\ell(y) \Rightarrow P_\ell(y)) :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L}{x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{read}\ x\ k(y) :: (y : A_k)}\ \&X$$

$$\frac{\Delta, x_1 : A \vdash P(x_1, x_2) :: (x_2 : B)}{\Delta \vdash \mathbf{write}\ x\ ((x_1, x_2) \Rightarrow P(x_1, x_2)) :: (x : A \multimap B)}\ \multimap R$$

$$\frac{}{x_1 : A, x : A \multimap B \vdash \mathbf{read}\ x\ (x_1, x_2) :: (x_2 : B)}\ \multimap X$$

Let's take a closer look at the meaning of linear functions. **write** $a$ $((x_1, x_2) \Rightarrow P(x_1, x_2))$ will write the continuation $(x_1, x_2) \Rightarrow P(x_1, x_2)$ to the cell at address $a$.

Conversely, **read** $a$ $(a_1, a_2)$ will read the continuation and pass it $a_1$ and $a_2$, where $a_1 : A$ is the "actual argument" of the function and $a_2 : B$ is the destination for the result.

Our syntax is now:

| | | | | |
|---|---|---|---|---|
| Values | $V$ | $::=$ | $k(x)$ | $(\oplus, \&)$ |
| | | $\mid$ | $(x_1, x_2)$ | $(\otimes, \multimap)$ |
| | | $\mid$ | $()$ | $(\mathbf{1})$ |
| Continuations | $K$ | $::=$ | $(\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}$ | $(\oplus, \&)$ |
| | | $\mid$ | $((x_1, x_2) \Rightarrow P(x_1, x_2))$ | $(\otimes, \multimap)$ |
| | | $\mid$ | $(() \Rightarrow P)$ | $(\mathbf{1})$ |
| Storable | $S$ | $::=$ | $V \mid K$ | |
| Processes | $P$ | $::=$ | $x \leftarrow P(x)\ ;\ Q(x)$ | cut |
| | | $\mid$ | **move** $x\ y$ | id |
| | | $\mid$ | **write** $x\ S$ | |
| | | $\mid$ | **read** $x\ S$ | |
| | | $\mid$ | **call** $p\ x\ y_1 \ldots y_n$ | |

The dynamics also changes subtly from purely positive types. We add the following two, while the remaining ones remain the same.

$$\begin{array}{lcl} \mathsf{proc}(\mathbf{write}\ a\ K) & \longrightarrow & \mathsf{cell}(a, K) \\ \mathsf{cell}(a, K), \mathsf{proc}(\mathbf{read}\ a\ V) & \longrightarrow & \mathsf{proc}(V \rhd K) \end{array}$$

We use map as iteration as an example. First, the message passing version.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'cons : bin * list, 'nil : 1}

type iter = &{'next : bin -o bin * iter, 'done : 1}
```

```
proc map (R : list) (i : iterm) (L : list) =
  recv L ( 'cons(p) => recv p ((x,L') =>
              f : bin -o bin * iter <- send i 'next(f) ;
              p : bin * iter <- send f (x, p) ;
              recv p ((y, i') => R' <- call map R' i' L' ;
                                 q : bin * list <- send q (y, R') ;
                                 send R 'cons(q)))
         | 'nil(u) => recv u (() =>
              v : 1 <- send i 'done(v)
              send R 'nil(v)) )
```

To convert this to a program using futures, positive send/receive become read-/write, respectively, while the this correspondence is switched for negative types.

```
    read L ( 'cons(p) => read p ((x,L') =>
              f : bin -o bin * iter <- read i 'next(f) ;
              p : bin * iter <- read f (x, p) ;
              read p ((y, i') => R' <- call map R' i' L' ;
                                 q : bin * list <- write q (y, R') ;
                                 write R 'cons(q)))
         | 'nil(u) => read u (() =>
              v : 1 <- read i 'done(v)
              write R 'nil(v)) )
```

## 4 Mixed Linear/Structural Futures

We have our recipe: We combine linear and structural types by adding appropriate shifts. The upshift is intrinsically negative, while the downshift is intrinsically positive. We have already assigned a syntax to the processes for these shifts that

we reuse.

| | | | | |
|---|---|---|---|---|
| Values | $V$ | $::=$ | $k(x)$ | $(\oplus, \&)$ |
| | | $\mid$ | $(x_1, x_2)$ | $(\otimes, \multimap)$ |
| | | $\mid$ | $()$ | $(\mathbf{1})$ |
| | | $\mid$ | $\langle x \rangle$ | $(\downarrow, \uparrow)$ |
| Continuations | $K$ | $::=$ | $(\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}$ | $(\oplus, \&)$ |
| | | $\mid$ | $((x_1, x_2) \Rightarrow P(x_1, x_2))$ | $(\otimes, \multimap)$ |
| | | $\mid$ | $(() \Rightarrow P)$ | $(\mathbf{1})$ |
| | | $\mid$ | $(\langle x \rangle \Rightarrow P(x))$ | $(\downarrow, \uparrow)$ |
| Storable | $S$ | $::=$ | $V \mid K$ | |
| Processes | $P$ | $::=$ | $x \leftarrow P(x)\ ;\ Q(x)$ | cut |
| | | $\mid$ | $\mathbf{move}\ x\ y$ | id |
| | | $\mid$ | $\mathbf{write}\ x\ S$ | |
| | | $\mid$ | $\mathbf{read}\ x\ S$ | |
| | | $\mid$ | $\mathbf{call}\ p\ x\ y_1 \ldots y_n$ | |

In the typing rules we just have to replace send and receive by write and read, as appropriate.

$$\frac{}{\Delta_\mathsf{S}, y_\mathsf{S} : A_\mathsf{S} \vdash \mathbf{write}\ x_\mathsf{L}\ \langle y_\mathsf{S} \rangle :: (x_\mathsf{L} :: \downarrow A_\mathsf{S})}\ \downarrow R$$

$$\frac{\Delta, y_\mathsf{S} : A_\mathsf{S} \vdash Q(y_\mathsf{S}) :: \delta}{\Delta, x_\mathsf{L} : \downarrow A_\mathsf{S} \vdash \mathbf{read}\ x_\mathsf{L}\ (\langle y_\mathsf{S} \rangle \Rightarrow Q(y_\mathsf{S})) :: \delta}\ \downarrow L$$

$$\frac{\Delta \vdash P(y_\mathsf{L}) :: (y_\mathsf{L} : A_\mathsf{L})}{\Delta \vdash \mathbf{write}\ x_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow P(y_\mathsf{L})) :: (x_\mathsf{S} : \uparrow A_\mathsf{L})}\ \uparrow R$$

$$\frac{}{\Delta_\mathsf{S}, x_\mathsf{S} : \uparrow A_\mathsf{L} \vdash \mathbf{read}\ x_\mathsf{S}\ \langle y_\mathsf{L} \rangle :: (y_\mathsf{L} : A_\mathsf{L})}\ \uparrow L$$

In the dynamics, the changes are a little less straightforward. For addresses of structural type we need to create *persistent cells* in the dynamics. We write $!\mathsf{cell}(a_\mathsf{S}, S)$ for a persistent cell. This means when it is read it remains in configuration rather than being consumed. The rules before remain what they are, assuming all the addresses are linear. In addition we have:

$$\begin{array}{lcl}
\mathsf{proc}(\mathbf{write}\ a_\mathsf{S}\ S) & \longrightarrow & !\mathsf{cell}(a_\mathsf{S}, S) \\
!\mathsf{cell}(a_\mathsf{S}, S), \mathsf{proc}(\mathbf{read}\ a_\mathsf{S}\ S') & \longrightarrow & \mathsf{proc}(S \bowtie S') \\
!\mathsf{cell}(b_\mathsf{S}, S), \mathsf{proc}(\mathbf{move}\ a_\mathsf{S}\ b_\mathsf{S}) & \longrightarrow & !\mathsf{cell}(a_\mathsf{S}, S)
\end{array}$$

Here $S \bowtie S'$ is defined by $K \bowtie V = V \bowtie K = V \rhd K$, accounting for both positive and negative types.

As an example, consider a map over a linear list with shared binary numbers. We write $A[m]$ for a type of mode $m$ and $(S)A_s$ for $\downarrow A_s$, signifying that the scope is shared. The code uses some compound values, analogous to message sequences. We will return to them in the next lecture.

```
type bin[m] = +{'b0 : bin[m], 'b1 : bin[m], 'e : 1}
type bin_s = bin[S]
type list = +{'cons : (S)bin_s * list, 'nil : 1} % linear

proc map (R : list) (F : bin_s -> bin_s) (L : list) =
  read L ( 'cons(<x>,L') => y : bin_s <- read F (x, y) ;
                            R' <- call map R' F L' ;
                            write R 'cons(<y>,R')
         | 'nil() => write R 'nil() )
```

It is of course possible to give other modes to map.

## References


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

Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 365(1–3):227–270, 2007.

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

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation of higher-order concurrent separation logic. *Journal of Functional Programming*, 29:e20, November 2018.

Peter O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

Frank Pfenning and Klaas Pruiksma. Relating message passing and shared memory, proof-theoretically. In S. Jongmans and A. Lopes, editors, *25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, pages 3–27, Lisbon, Portugal, June 2023. Springer LNCS 13908. Notes to an invited talk.

Klaas Pruiksma and Frank Pfenning. Back to futures. *Journal of Functional Programming*, 32:e6, 2022.

Siva Somayyajula and Frank Pfenning. Type-based termination for futures. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, pages 12:1–12:21, Haifa, Israel, August 2022. LIPIcs 228.

Siva Somayyajula and Frank Pfenning. Dependent type refinements for futures. In M. Kerjean and P. Levy, editors, *39th International Conference on Mathematical Foundations of Programming Semantics (MFPS 2023)*, Bloomington, Indiana, USA, June 2023. Preliminary version.