Relating Message Passing and Shared Memory, Proof-Theoretically

Frank Pfenning
Klaas Pruiksma

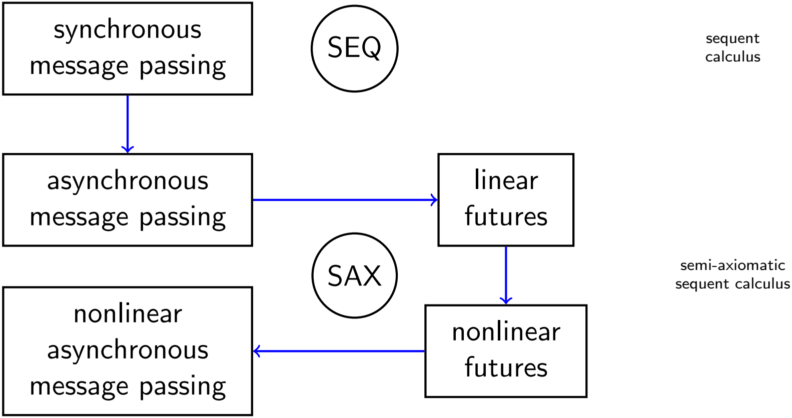
Computer Science Department
Carnegie Mellon University

DisCoTec'23
June 21, 2023
Invited Talk

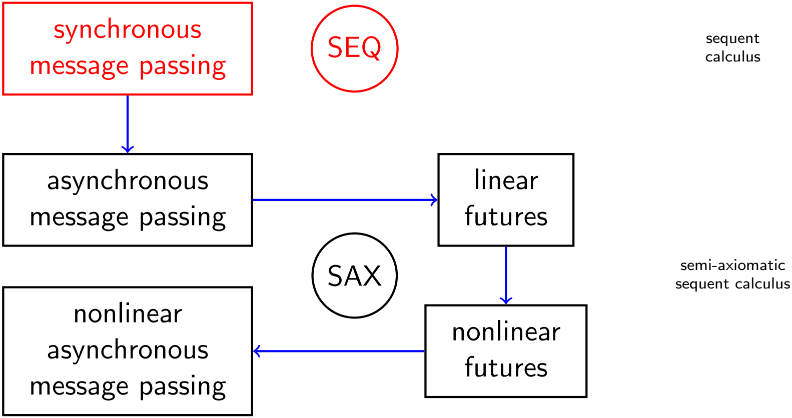
Goals

- High level abstractions for parallel/concurrent programming
- Elegant intrinsically safe programming
  - Session fidelity / type preservation
  - Deadlock freedom / progress
- Reasoning about
  - correctness
  - efficiency (work, span, messages/space)
  - timing
- Subgoal: relating message passing to shared memory
- "Secret weapon": proof theory

Our Journey


synchronous message passing
SEQ
sequent calculus
asynchronous message passing
linear futures
SAX
semi-axiomatic sequent calculus
nonlinear asynchronous message passing
nonlinear futures

Our Journey


synchronous message passing
SEQ
sequent calculus
asynchronous message passing
linear futures
SAX
semi-axiomatic sequent calculus
nonlinear asynchronous message passing
nonlinear futures

Synchronous Message Passing Example

```
server :: (c : int -o (int -o int)) =
  recv c (x =>
  recv c (y =>
  send c (x-y)))
```

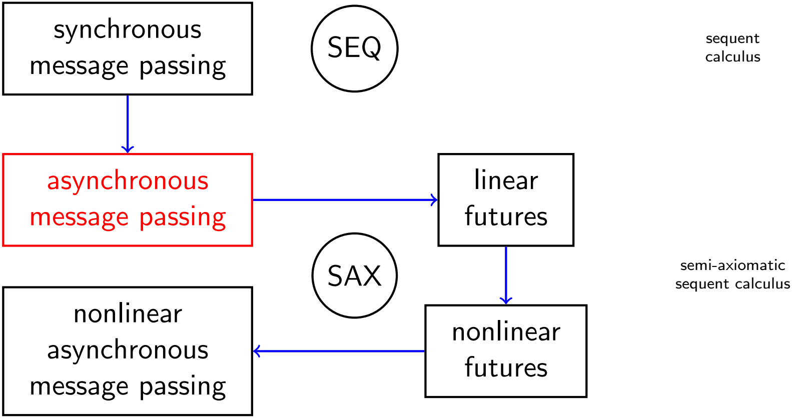
```
client (c : int -o (int -o int)) :: (a : int) =
  send c 35 ;
  send c 17 ;
  recv c (z =>
  send a z)
```

```
proc (server c)             , proc (client (c) a)      % c : int -o (int -o int)
proc (recv c (x => ...))    , proc (send c 35 ; ...)   % c :         int -o int
proc (recv c (y => ...))    , proc (send c 17 ; ...)   % c :                int
proc (send c (35-17))       , proc (recv c (z => ...)) % (c closed)
                              proc (send a 18)         % (a closed)
```

Synchronous Message Passing

- Session types [Honda'93] [Honda et al.'98]
- Curry-Howard correspondence with sequent calculus for linear logic [Caires & Pf'10] [Wadler'12] [Caires et al.'16]
  - Propositions as session types
  - Sequent calculus proofs a processes
  - Cut reduction as synchronous communication
- Can simulate typed asynchronous communication [Griffith & Pf'16]

Our Journey



synchronous
message passing
SEQ
sequent
calculus
asynchronous
message passing
linear
futures
SAX
semi-axiomatic
sequent calculus
nonlinear
asynchronous
message passing
nonlinear
futures

Asynchronous Message Passing

- Fundamentally: sender does not block
- Dynamics [Boudol'92]
  - Key idea: a message is a process
- Statics [Honda'91] [Kobayashi'98] [Kobayashi et al.'99] [Gay & Vasconcelos'10]
  - Key idea: continuation channels
- Can simulate typed synchronous message passing
- Can we establish a Curry-Howard correspondence?
  - Propositions as session types (no change)
  - Proofs as processes?
  - Cut reduction as asynchronous communication?

Problem: Ordering of Messages

- Messages may be received out of order

```
client (c : int -o (int -o int)) :: (a : int) =
    send c 35 ;
    send c 17 ;
    recv c (z => % z = 18 or -18?
    send a z)
```

- Jeopardizes type safety

```
client (c:int -o (bool -o int)) :: (a:int) =
    send c 35 ;    % must be first
    send c true ; % must be second
    ...
```

- Solution: continuation channels!

Continuation Channels

- First approximation

```
client (c : int -o (int -o int)) :: (a : int) =
    send c  (35,c1) ;  % c1 : int -o int
    send c1 (17,c2) ;  % c2 :        int
    recv c2 (z =>      % z : int
    send a z)
```

- With allocation of continuation channels

```
client (c:int -o (int -o int*1)) :: (a:int*1) =
  c1 <- send c  (35,c1) ; % c1 :  int -o int*1
  c2 <- send c1 (17,c2) ; % c2 :         int*1
  recv c2 ((z,c3) =>      % z : int, c3 : 1
  send a (z,c3))
```

Continuation Channels

- Client (repeat)

```
client (c:int -o (int -o int*1)) :: (a:int*1) =
  c1 <- send c  (35,c1) ; % c1 :  int -o int*1
  c2 <- send c1 (17,c2) ; % c2 :         int*1
  recv c2 ((z,c3) =>      % z : int, c3 : 1
  send a (z,c3))
```

- Matching server

```
server :: (c : int -o (int -o int * 1)) =
  recv c  ((x, c1) => % c1 : int -o int * 1
  recv c1 ((y, c2) => % c2 :        int * 1
  c3 <- send c3 () ;  % c3 : 1
  send c2 (x-y, c3)))
```

Asynchronous Communication: Statics

- Judgment

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{use}} \vdash P :: \underbrace{(z : C)}_{\text{provide}}$$

  - Channels x_i and z define interface to P
  - Process P is client of $x_i : A_i$, provides $z : C$
  - Session types A_i and C prescribe communication protocols
  - Communication is bidirectional

- Allocating a fresh channel / spawning a new process

$$\frac{\overbrace{\Gamma \vdash P(x) :: (x : A)}^{\text{provider of } x} \quad \overbrace{\Delta, x : A \vdash Q(x) :: (d : D)}^{\text{client of } x}}{\Gamma, \Delta \vdash (x \leftarrow P(x)\, ;\, Q(x)) :: (d : D)}\ \text{alloc/spawn}$$

Asynchronous Communication: Dynamics

- A configuration is described by a multiset of semantic objects

$$\begin{array}{lcll} \text{Objects} & \phi & ::= & \mathsf{proc}\ P \mid \ldots \\ \text{Configurations} & \mathcal{C} & ::= & \phi \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \end{array}$$

- Dynamics is described by multiset rewriting rules, for example:

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

  - Match left-hand side against part of configuration
  - Replace by right-hand side

Asynchronous Communication: Logic

- Recall alloc/spawn

$$\dfrac{\overbrace{\Gamma \vdash P(x) :: (x : A)}^{\text{provider of } x} \quad \overbrace{\Delta, x : A \vdash Q(x) :: (d : D)}^{\text{client of } x}}{\Gamma, \Delta \vdash (x \leftarrow P(x) \,;\, Q(x)) :: (d : D)}\ \text{alloc/spawn}$$

- Erase computational decorations: cut

$$\dfrac{\Gamma \vdash A \quad \Delta, A \vdash D}{\Gamma, \Delta \vdash D}\ \text{cut}$$

- Same as for synchronous communication

Asynchronous Communication: Meaning of Connectives

- Types prescribe protocols
- Polarities determine direction of communication
  - Negatives $A \multimap B$, $A \mathbin{\&} B$: provider receives, client sends
  - Positives $A \otimes B$, $\mathbf{1}$, $A \oplus B$: provider sends, client receives
- Basic principles:
  - Messages are processes
  - Messages have continuation channels

Receiving a Channel / Type $A \multimap B$

- Provider view: receive channel x along c

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

  - x stands for channel of type A
  - y stands for a continuation channel of type B
- Client view: send channel a along c

$$\frac{}{a : A, c : A \multimap B \vdash \textbf{send}\ c\ \langle a, b\rangle :: (b : B)} \multimap L^0 \quad \frac{}{A, A \multimap B \vdash B} \multimap L^0$$

  - **send** $c\ \langle a, b\rangle$: sending a with continuation channel b along c
  - $\multimap L$ rule of sequent calculus becomes an axiom $\multimap L^0$

Communication / Cut Reduction

- Multiset rewriting rule

$$\begin{array}{l} \text{proc}\ (\textbf{recv}\ c\ (\langle x, y\rangle \Rightarrow P(x,y))), \\ \text{proc}\ (\textbf{send}\ c\ \langle a, b\rangle) \\ \mapsto \\ \text{proc}\ P(a,b) \end{array}$$

- Mirrors cut reduction

$$\dfrac{\dfrac{\begin{array}{c}P(x,y)\\ \Gamma, x : A \vdash y : B\end{array}}{\Gamma \vdash c : A \multimap B}\multimap R \quad \dfrac{}{a : A, c : A \multimap B \vdash b : B}\multimap L^0}{\Gamma, a : A \vdash b : B}\text{cut} \quad \mapsto \quad \begin{array}{c}P(a,b)\\ \Gamma, a : A \vdash b : B\end{array}$$

Sending a Channel / Type $A \otimes B$

- Like $A \multimap B$, swapping sending/receiver roles
- Provider view: send channel a with cont. channel b along c

$$\frac{}{a : A, b : B \vdash \textbf{send}\ c\ \langle a, b\rangle :: (c : A \otimes B)}\ \otimes R^0 \qquad \frac{}{A, B \vdash A \otimes B}\ \otimes R^0$$

- Client view: receive channel x with cont. channel y along c

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

- The same communication rule applies!

Termination / Type 1

- Only message without a continuation
- Provider view ($1R^0 = 1R$)

$$\frac{}{\cdot \vdash \textbf{send}\ c\ \langle\,\rangle :: (c : 1)}\ 1R^0 \qquad \frac{}{\cdot \vdash 1}\ 1R^0$$

- Client view

$$\frac{\Gamma \vdash P :: (d : D)}{\Gamma, c : 1 \vdash \textbf{recv}\ c\ (\langle\,\rangle \Rightarrow P) :: (d : D)}\ 1L \qquad \frac{\Gamma \vdash D}{\Gamma, 1 \vdash D}\ 1L$$

- Dynamics

$$\mathsf{proc}\ (\textbf{send}\ c\ \langle\,\rangle), \mathsf{proc}\ (\textbf{recv}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ P$$

External and Internal Choice

- External (client) choice $\&_{\ell \in L}\{\ell : A_\ell\}$
- Internal (provider) choice $\oplus_{\ell \in L}\{\ell : A_\ell\}$
- Each alternative labeled uniquely from a finite set L
- Example:

```
arith = &{diff : int -o int -o int * 1,
          sqrt : int -o +{none : 1,
                          some : int * 1}}

server :: (c : arith) =
recv c ( diff(c1) => ...
       | sqrt(c1) => recv c1 ((x, c2) =>
          if x < 0
          then send c2 (none())
          else c3 <- send c2 (some(c3)) ;
               send c3 (isqrt(x), ())) )
```

External Choice / $A \mathbin{\&} B$

- Provider view: receive and branch on label ℓ

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

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mathbin{\&} B}\ \&R$$

- Client view: send label k

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

$$\frac{}{A \mathbin{\&} B \vdash A}\ \&L_1^0 \qquad \frac{}{A \mathbin{\&} B \vdash B}\ \&L_2^0$$

External Choice / Dynamics

- Multiset rewriting rule

$$\begin{array}{l} \text{proc}\ (\textbf{recv}\ c\ (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}), \\ \text{proc}\ (\textbf{send}\ c\ k(a)) \\ \mapsto \\ \text{proc}\ P_k(a) \qquad (k \in L) \end{array}$$

- Internal choice uses the same computation rule

Internal Choice / $A \oplus B$

- Like external choice, reversing provider/client roles
- Computation rule remains the same
- Typing rules

$$\frac{(k \in L)}{a : A_k \vdash \textbf{send}\ c\ k(a) :: \oplus_{\ell \in L}\{\ell : A_\ell\}} \oplus R$$

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

- Logically

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

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

Recursion

- Add equirecursive types
- Add recursively defined processes
- Depart from strict Curry-Howard correspondence
  - Consider circular/infinitary proofs

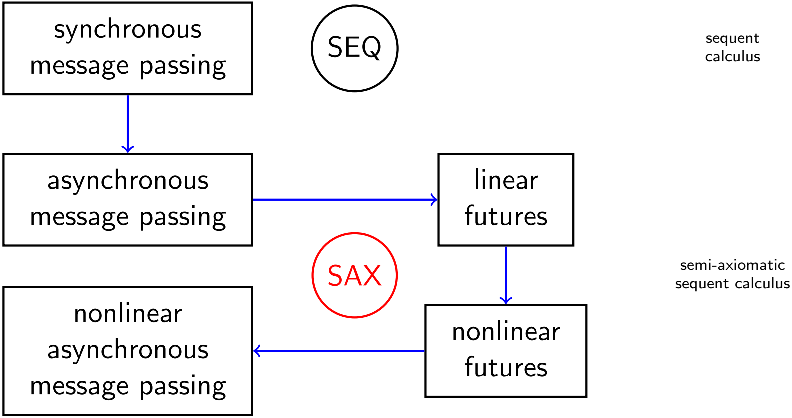
Example: Storage Server

```
store A = &{insert : A -o store A,
            delete : +{none : 1,
                       some : A * store A}}

% treating L as a local variable
server (L : list A) :: (s : store A) =
recv s ( insert(s1) =>
           recv s1 ((x,s2) => call server (x::L) s2)

         | delete(s1) =>
           case L ( nil => send s1 none()
                  | x::xs => s2 <- send s1 some(s2) ;
                             s3 <- send s2 (x,s3) ;
                             call server (xs) s3 ))
```

Our Journey


synchronous message passing
SEQ
sequent calculus
asynchronous message passing
linear futures
SAX
semi-axiomatic sequent calculus
nonlinear asynchronous message passing
nonlinear futures

The (Linear) Semi-Axiomatic Sequent Calculus (SAX)

$$\frac{\Gamma \vdash A \quad \Delta, A \vdash D}{\Gamma, \Delta \vdash D}\ \text{cut} \qquad \frac{}{A \vdash A}\ \text{id}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ {\multimap}R \qquad \frac{}{A, A \multimap B \vdash B}\ {\multimap}L^0$$

$$\frac{}{A, B \vdash A \otimes B}\ {\otimes}R^0 \qquad \frac{\Gamma, A, B \vdash D}{\Gamma, A \otimes B \vdash D}\ {\otimes}L$$

$$\frac{}{\cdot \vdash 1}\ 1R^0 \qquad \frac{\Gamma \vdash D}{\Gamma, 1 \vdash D}\ 1L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&R \qquad \frac{}{A \,\&\, B \vdash A}\ \&L_1^0 \quad \frac{}{A \,\&\, B \vdash B}\ \&L_2^0$$

$$\frac{}{A \vdash A \oplus B}\ {\oplus}R_1^0 \quad \frac{}{B \vdash A \oplus B}\ {\oplus}R_2^0 \qquad \frac{\Gamma, A \vdash D \quad \Gamma, B \vdash D}{\Gamma, A \oplus B \vdash D}\ {\oplus}L$$

SAX

- SAX replaces all noninvertible rules of the sequent calculus by axioms
- Add weakening and contraction for (nonlinear) SAX

$$\frac{\Gamma \vdash D}{\Gamma, A \vdash D}\ \text{weaken} \qquad \frac{\Gamma, A, A \vdash D}{\Gamma, A \vdash D}\ \text{contract}$$

- Mixed linear/nonlinear (= adjoint) SAX [Pruiksma'23]
- SAX satisfies a form of cut elimination [DeYoung et al.'20]

Syntax Summary

Values	V	$::=$	$\langle\,\rangle$	$(\bot, 1)$
		$\mid$	$\langle a, b \rangle$	$(\multimap, \otimes)$
		$\mid$	$k(a)$	$(\&, \oplus)$
Continuations	K	$::=$	$\langle\,\rangle \Rightarrow P$	$(\bot, 1)$
		$\mid$	$\langle x, y \rangle \Rightarrow P(x, y)$	$(\multimap, \otimes)$
		$\mid$	$(\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}$	$(\&, \oplus)$
Processes	P	$::=$	$x \leftarrow P(x)$ **;** $Q(x)$	allocate/spawn
		$\mid$	**send** $c\ V$	send V along c
		$\mid$	**recv** $c\ K$	receive along c, pass to K
		$\mid$	**fwd** $a\ b$	forward (see paper)
		$\mid$	**call** $p\ (a_1, \ldots, a_n)\ c$	call process (see paper)

Refactoring Computation Rules

- Recall basic principles of typed asynchronous communication
  - Messages are processes
  - Message ordering via continuation channels
- New semantic objects msg $c\ V$ and cont $c\ K$

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

$$
\begin{array}{lclcl}
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\
\langle a, b\rangle & \triangleright & (\langle x, y\rangle \Rightarrow P(x,y)) & = & P(a,b) \\
k(a) & \triangleright & (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} & = & P_k(a)
\end{array}
$$

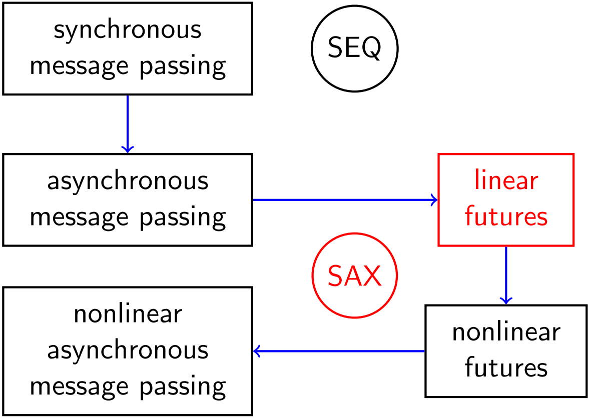
Polarity of Propositions / Types

- Proof theory (sequent calculus): invertible rules
  - Negatives: right rules are invertible ($A \mathbin{\&} B$, $A \multimap B$, $\bot$)
  - Positives: left rules are invertible ($A \oplus B$, $A \otimes B$, 1)
  - Invertible rules carry no information
  - Corresponding processes receive
  - In SAX, these rules remain unchanged
- Proof theory: noninvertible rules
  - Negatives: left rules are noninvertible
  - Positives: right rules are noninvertible
  - In SAX, these become axioms
  - Corresponding processes send
  - In SAX, these rules become axioms (= represent messages)
- Computational summary
  - Negatives: provider sends, client receives
  - Positives: provider receives, client sends

Summary of Asynchronous Message Passing

- Language as an intermediate point between a source level notation and a low level implementation
- Elegant proof-theoretic foundation in the semi-axiomatic sequent calculus SAX
  - Propositions as types
  - Proofs as programs
  - Cut reduction as asynchronous communication
- Consequently, for configurations:
  - Theorem: type preservation (= session fidelity)
  - Theorem: progress (= deadlock freedom)

Our Journey



sequent calculus

semi-axiomatic sequent calculus

Futures

- What is a future in a functional language? [Halstead'85]

$$\textbf{let future } x = e \textbf{ in } e'(x)$$

  - Allocate a new future d
  - Evaluate e with destination d
  - In parallel, evaluate $e'(d)$
  - If $e'(d)$ touches d, it blocks until d is written
- A parallel construct in a (by default) sequential language
- A future is a write-once form of shared memory
- Four steps
  - Step 0: introduce types
  - Step 1: make memory explicit
  - Step 2: make futures explicit
  - Step 3: change default from sequential to parallel

Futures / Statics

- Variables now stand for addresses
- Every expression (= thread) executes with a destination [Wadler'84]
- Typing judgment

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

- A thread P terminates as it writes to its destination z
- A thread P reads from cells at addresses x_i
- Translate **let future** $x = e$ **in** $e'(x)$ as

$$x \leftarrow P(x) \,;\, Q(x)$$

  where $P(x)$ has destination x and $Q(x)$ reads from x

Futures / Dynamics

- Semantic objects
  - thread P — thread P is executing
  - cell c S — memory cell c holds storable S
  - susp c S — suspension S
  - Storable $S ::= K \mid V$
- Processes P now with **read**/**write** instead of **send**/**receive**
- Dynamics

$$
\begin{array}{lcl}
\text{thread } (x \leftarrow P(x) \,;\, Q(x)) & \mapsto & \text{thread } P(a), \text{thread } Q(a) \\
\text{thread } (\textbf{write } c\ S) & \mapsto & \text{cell } c\ S \\
\text{thread } (\textbf{read } c\ S) & \mapsto & \text{susp } c\ S \\
\text{cell } c\ V, \text{susp } c\ K & \mapsto & \text{thread } (V \triangleright K) \\
\text{cell } c\ K, \text{susp } c\ V & \mapsto & \text{thread } (V \triangleright K)
\end{array}
$$

Memory Example

- Memory model example: binary 6 at address c_0

$$\begin{array}{l} \text{cell } c_0\ b0(c_1), \\ \text{cell } c_1\ b1(c_2), \\ \text{cell } c_2\ b1(c_3), \\ \text{cell } c_3\ e(c_4), \\ \text{cell } c_4\ \langle\,\rangle \end{array}$$

Example: Binary Successor

```
% binary numbers with least significant bit first
% labels b0, b1, e are now tags
bin = +{b0 : bin, b1 : bin, e : 1}

zero :: (y : bin) = write y e()

succ (x : bin) :: (y : bin) =
read x ( b0(x') => write y b1(x')
       | b1(x') => y' <- call succ (x') y' ;
                   write y b0(y')
       | e() => y' <- write y' e() ;
                write y b1(y') )

% a pipeline with two succ threads
plus2 (x : bin) :: (z : bin) =
  y <- call succ (x) y ;
  call succ (y) z
```

Correspondence with Asynchronous Message Passing

- Channels become memory addresses
- Allocate/spawn remains unchanged
- For positives: **write** $\sim$ **send**, **read** $\sim$ **recv**
- Example:

```
zero :: (y : bin) = send y e()

succ (x : bin) :: (y : bin) =
recv x ( b0(x') => send y b1(x')
       | b1(x') => y' <- call succ (x') y' ;
                   send y b0(y')
       | e() => y' <- send y' e() ;
                send y b1(y') )

% a pipeline with two succ processes
plus2 (x : bin) :: (z : bin) =
  y <- call succ (x) y ;
  call succ (y) z
```

Example: Lists

```
list A = &{nil : 1, cons : A * list A}

nil :: (L : list A) = write L nil()

cons (x : A, xs : list A) :: (L : list A) =
  p <- write p (x, xs) ;
  write L cons(p)
```

Examples: Storage Server

```
store A = &{insert : A -o store A,
            delete : +{none : 1,
                       some : A * store A}}

server (L : list A) :: (s : store A) =
write s ( insert(s1) =>
          write s1 ((x,s2) =>
          L' <- call cons (x, L) L' ;
          call server L' s2)
        | delete(s1) =>
          read L ( nil() => send s1 none()
                 | cons(p) => read p (x,xs) =>
                   s2 <- write s1 some(s2) ;
                   s3 <- write s2 (x,s3) ;
                   call server (xs) s3 ))
```

Positive Correspondences

- Recall $V ::= \langle a, b \rangle \mid \langle\,\rangle \mid k(a)$
- Recall positives $A \oplus B$, $A \otimes B$, 1
- Syntax

Message Passing	**Futures**
$x \leftarrow P(x) \,;\, Q(x)$	$x \leftarrow P(x) \,;\, Q(x)$
$\textbf{send}^{+}\ c\ V$	$\textbf{write}\ c\ V$
$\textbf{recv}^{+}\ c\ K$	$\textbf{read}\ c\ K$
$\textbf{fwd}^{+}\ c\ a$	$\textbf{move}\ c\ a$

- Dynamics

$$\begin{array}{lcl}
\mathsf{thread}\ (x \leftarrow P(x) \,;\, Q(x)) & \mapsto & \mathsf{thread}\ P(a), \mathsf{thread}\ Q(a) \\
\mathsf{thread}\ (\textbf{write}\ c\ V) & \mapsto & \mathsf{cell}\ c\ V \\
\mathsf{thread}\ (\textbf{read}\ c\ K) & \mapsto & \mathsf{susp}\ c\ K \\
\mathsf{cell}\ c\ V, \mathsf{susp}\ c\ K & \mapsto & \mathsf{thread}\ (V \triangleright K)
\end{array}$$

Negative Correspondences

- Recall

```
diff :: (c : int -o (int -o int * 1)) =
  recv c ((x,c1) =>
  recv c1 ((y,c2) =>
  send c2 (x-y,()))))
```

- According to typing `diff` should write to *c*!
- Idea: We write a continuation to *c*!

```
diff :: (c : int -o (int -o int * 1)) =
  write c  ((x,c1) =>
  write c1 ((y,c2) =>
  write c2 (x-y,()))))
```

Negative Correspondences

- Server (repeat)

```
diff :: (c : int -o (int -o int * 1)) =
  write c  ((x,c1) =>
  write c1 ((y,c2) =>
  write c2 (x-y,())))
```

- Matching client reads continuations and passes them values

```
client (c:int -o (int -o int*1))::(a : int*1) =
  c1 <- read c (35, c1) ;
  c2 <- read c1 (17, c2) ;
  read c2 ((z,c3) =>
  write a (z,c3))
```

Negative Correspondences

- Recall continuations for negatives

$$
\begin{array}{lllll}
K & ::= & \langle x, y\rangle \Rightarrow P(x, y) & (\multimap) & x \text{ is argument} \\
 & & & & y \text{ is destination} \\
 & \mid & (\ell(x) \Rightarrow P_\ell(x))_{\ell\in L} & (\&) & k \text{ is label/method} \\
 & & & & x \text{ is destination} \\
 & \mid & \langle\,\rangle \Rightarrow P & (\bot) &
\end{array}
$$

- Syntax

Message Passing	**Futures**
send$^-$ $c\ V$	**read** $c\ V$
recv$^-$ $c\ K$	**write** $c\ K$
fwd$^-$ $c\ a$	**move** $c\ a$

- Dynamics

$$
\begin{array}{lcl}
\text{thread}\ (\textbf{write}\ c\ K) & \mapsto & \text{cell}\ c\ K \\
\text{thread}\ (\textbf{read}\ c\ V) & \mapsto & \text{susp}\ c\ V \\
\text{cell}\ c\ K, \text{susp}\ c\ V & \mapsto & \text{thread}\ (V \rhd K)
\end{array}
$$

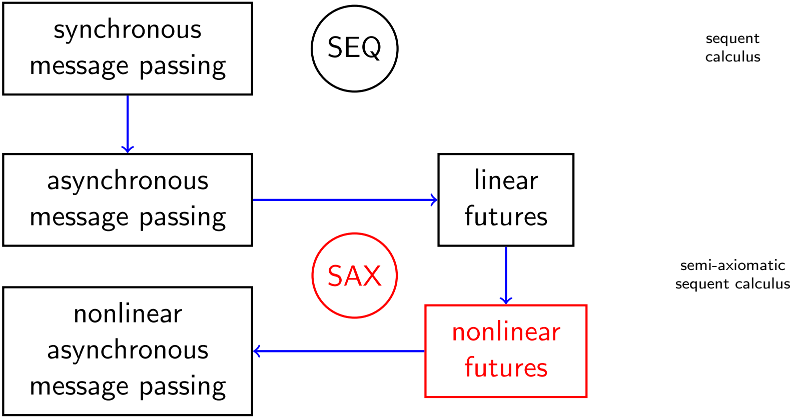
An Exact Correspondence

- On syntax and dynamic objects

Message Passing	**Futures**
send$^+$ c V	**write** c V
recv$^+$ c K	**read** c K
recv$^-$ c K	**write** c K
send$^-$ c V	**read** c V
proc P	thread P
msg$^+$ c V	cell c V
cont$^+$ c K	susp c K
cont$^-$ c K	cell c K
msg$^-$ c V	susp c V

- All messages are small (msg$^+$ c V, msg$^-$ c V)
- Storables are small values or continuations (cell c V, cell c K)

Our Journey


synchronous message passing
SEQ
sequent calculus
asynchronous message passing
linear futures
SAX
semi-axiomatic sequent calculus
nonlinear asynchronous message passing
nonlinear futures

Relation to Traditional Futures

- Futures are a single parallel construct in an otherwise sequential language
  - Just a matter of scheduling!
  - Sequential $x \xleftarrow{cbv} P(x)\ ;\ Q(x)$ for "call-by-value"
  - Block $Q(a)$ until $P(a)$ has written to new future a
  - Sequential $x \xleftarrow{cbn} P(x)\ ;\ Q(x)$ for "call-by-need"
  - Block $P(a)$ until $Q(a)$ touches new future a
- Futures are not linear
  - Proof theory: add (implicit or explicit) weakening and contraction
  - Dynamics: allow zero or multiple readers for every cell
  - Linear futures can be asymptotically more efficient than nonlinear futures [Blelloch & Reid-Miller'99]
  - Mixed linear/nonlinear futures [Pruiksma'23]

Nonlinear Futures

- Easy to accommodate (in fact, discovered first)
- Semantics objects $!\phi$ are persistent
  - Not removed from the configuration when matched

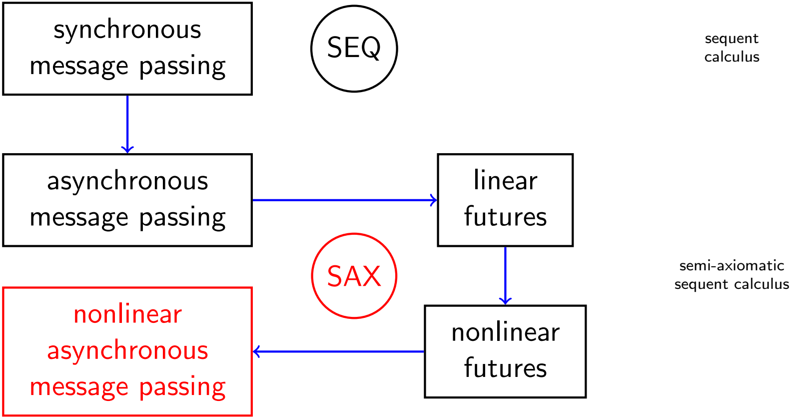
$$
\begin{array}{lcl}
\text{thread}\ (\textbf{write}\ c\ S) & \mapsto & !\text{cell}\ c\ S \\
\text{thread}\ (\textbf{read}\ c\ S) & \mapsto & \text{susp}\ c\ S \\
!\text{cell}\ c\ V, \text{susp}\ c\ K & \mapsto & \text{thread}\ c\ (V \rhd K) \\
!\text{cell}\ c\ K, \text{susp}\ c\ V & \mapsto & \text{thread}\ c\ (V \rhd K)
\end{array}
$$

- Can make a cell ephemeral or persistent, depending on its mode [Pruiksma'23]
- Requires garbage collection unless weakening (drop) and contraction (duplicate) are explicit operations [Girard & Lafont'87] [Gupta'22]

Summary: Futures

- Still just a proof term assignment for SAX
- Theorem: Type preservation
- Theorem: Progress
- Typed traditional futures a simple fragment
- Economical, intermediate-level language
  - **alloc**, **read**, **write**, **copy**, **call**
  - Sequential prototype implementation in progress

Our Journey


synchronous message passing
SEQ
sequent calculus
asynchronous message passing
linear futures
SAX
semi-axiomatic sequent calculus
nonlinear asynchronous message passing
nonlinear futures

From Nonlinear Futures to Nonlinear Message Passing

- Synchronous (untimed) message passing inherently linear?
- What about asynchronous message passing?
- Exploit the correspondence with futures to derive nonlinear asynchronous message passing!

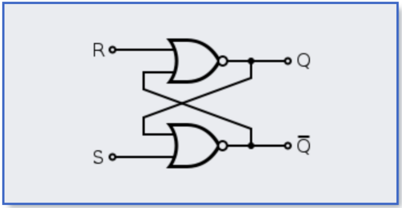
Example: Nor Gate

- "Nor" of two bits is linear

```
bit = +{b0 : 1, b1 : 1}

nor (x : bit) (y : bit) :: (z : bit) =
recv x ( b0() => recv y ( b0() => send z b1()
                        | b1() => send z b0() )
       | b1() => recv y ( b0() => send z b0()
                        | b1() => send z b0() ) )
```

Example: A Latch




```
bit = +{b0 : 1, b1 : 1}
bits2 = (bit * bit) * bits2

latch (q:bit, qbar:bit, in:bits2) :: (out:bits2) =
recv in (((r,s),in') =>
   q'    <- call nor (r, qbar) q' ;
   qbar' <- call nor (s, q) qbar' ;
   out'  <- call latch (q', qbar', in') out' ;
   send out ((q', qbar'), out'))
```

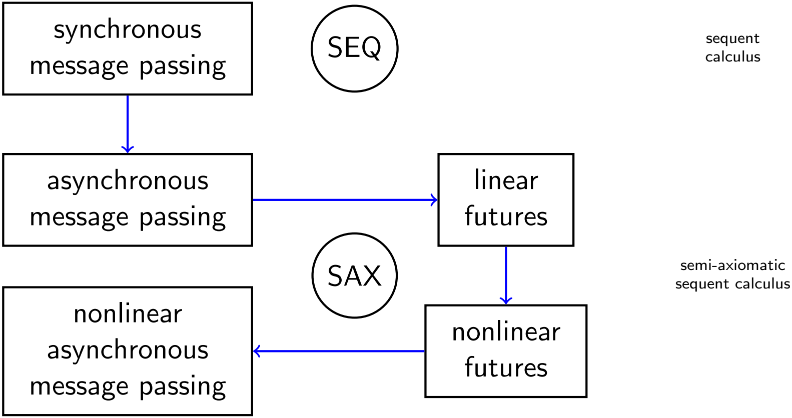
Nonlinear Asynchronous Message Passing

- A provider has multiple clients
  - Messages of positive type from provider to client are modeled as persistent objects $!\mathsf{msg}^+ \; c \; V$
  - Continuations of negative type expecting messages from client are modeled as persistent objects $!\mathsf{cont}^- \; c \; V$
- Dynamics

$$
\begin{array}{lcl}
\mathsf{proc}\;(x \leftarrow P(x) \; ; \; Q(x)) & \mapsto & \mathsf{proc}\;P(a), \mathsf{proc}\;Q(a) \\
\mathsf{proc}\;(\mathbf{send}^+ \; c \; V) & \mapsto & !\mathsf{msg}^+ \; c \; V \\
\mathsf{proc}\;(\mathbf{recv}^+ \; c \; K) & \mapsto & \mathsf{cont}^+ \; c \; K \\
!\mathsf{msg}^+ \; c \; V, \mathsf{cont}^+ \; c \; K & \mapsto & \mathsf{proc}\;(V \triangleright K) \\
\mathsf{proc}\;(\mathbf{send}^- \; c \; V) & \mapsto & \mathsf{msg}^- \; c \; V \\
\mathsf{proc}\;(\mathbf{recv}^- \; c \; K) & \mapsto & !\mathsf{cont}^- \; c \; K \\
!\mathsf{cont}^- \; c \; K, \mathsf{msg}^- \; c \; V, & \mapsto & \mathsf{proc}\;(V \triangleright K)
\end{array}
$$

- Implicitly exploits continuation channels for soundness

Our Journey


synchronous message passing
SEQ
sequent calculus
asynchronous message passing
linear futures
SAX
semi-axiomatic sequent calculus
nonlinear asynchronous message passing
nonlinear futures

Summary

- Analyzed typed asynchronous message passing and futures-based shared memory from a proof-theoretic perspective
- Perfect correspondence between message passing and futures
  - The difference lies in the interpretation of SAX
  - Using adjoint construction, we can freely combine
- Linear correspondences extend to nonlinear and mixed ones
  - Consequence of proof-theoretic approach
- There are at least two natural sequential schedulers that can be exposed in the syntax ("by value" and "by need")

Excursion: Logic Styles and Computation

- All logics below intuitionistic (and may be linear)
- Hilbert-style
  - Form: one rule (modus ponens), many axioms
  - Computationally: combinatory reduction [Curry'34]
- Natural deduction [Gentzen'35]
  - Form: introduction and elimination rules
  - Computationally: λ-calculus [Howard'69]
- Sequent calculus (linear only?)
  - Form: right and left rules
  - Computationally: synchronous message passing
- Semi-axiomatic sequent calculus
  - Form: right and left rules and axioms
  - Computationally: asynchronous message passing
  - Computationally: futures

Exploiting the Proof-Theoretic Perspective

- Sized types for reasoning about termination [Somayyajula & Pf'22]
- Dependent types for reasoning about partial correctness [Caires et al.'12] [Somayyajula & Pf'23]
- Logical relations [Pérez et al.'12] [Pruiksma'23]
- Efficient data layout for SAX [DeYoung & Pf'22]
- Proof-theoretic compilation from functional notation (natural deduction) to adjoint SAX [DeYoung, Ng, Roshal]
- Subtyping and polymorphism [DeYoung, Mordido, Pf, Das]

Thanks!

Klaas Pruiksma (coauthor)

Stephanie Balzer, Henry DeYoung, Daniel Ng, Sophia Roshal, Siva Somayyajula (closely related)

Luís Caires, Ankush Das, Dennis Griffith, Andreia Mordido, Jorge Pérez, Bernardo Toninho (somewhat related)

Organizers and programm committees for the invitation!