A Shared Memory Semantics for Session Types

Frank Pfenning
joint work with Klaas Pruiksma

Department of Computer Science
Carnegie Mellon University

Invited Talk, Linearity/TLLA 2018
July 8, 2018

Adventures with Curry and Howard

- Deep connections between logic and computation
- Depend on the logic but also the deductive system
- All logics and systems here are intuitionistic

Logic	System	Computation	Programming
structural	axiomatic	combinatory reduction	functional
structural	nat. ded.	substitution	functional
structural	seq. calc.*	explicit substitution	functional
linear	seq. calc.	message passing	concurrent (synch)
linear	seq. calc.†	message passing	concurrent (asynch)
linear	seq. calc.†	shared memory	“parallel” functional (lin)
multistructural	seq. calc.†	shared memory	“parallel” functional

*: (partially) focused

†: (partially) axiomatic

Outline

- Linear logic, sequent calculus, and synchronous communication
- A calculus for asynchronous communication
- A shared memory interpretation
- Outlook (ongoing work)

Linear Propositions as Session Types

- A Curry-Howard interpretation linear logic
  [Honda'93][Bellin & Scott'94][Honda et al.'98]...
  [Caires & Pf.'10][Wadler'12][Toninho et al.'13]...
- Linear propositions $\Leftrightarrow$ session types
- Sequent proofs $\Leftrightarrow$ message-passing concurrent programs
- Cut reduction $\Leftrightarrow$ communication

Cut as Parallel Composition

- Linear sequents

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

- Typing process P with channels x_i and z

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash P :: (z : C)$$

- P is client to $x_1, \ldots, x_n$, provides z
- Cut as parallel composition with a shared private channel

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

Substructural Operational Semantics

- Process configuration consists of semantic objects proc(c, P): process P provides along channel c
  - Every channel c has a unique provider, unique client*
  - Order is irrelevant
  - By convention, a provider precedes its client
- Transition rules of the operational semantics match the left-hand side against a subset of the objects and replace them by the right-hand side (multiset rewriting)
- Example: cut executes by spawning a new process

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

- Rewriting is highly nondeterministic, but confluent with session types

Cut Reduction as Communication

- Consider internal choice $A \oplus B$

$$
\dfrac{\dfrac{\dfrac{P}{\Delta \vdash A}}{\Delta \vdash A \oplus B}\ \oplus R_1 \qquad \dfrac{\dfrac{Q_1}{\Delta', A \vdash C} \quad \dfrac{Q_2}{\Delta', B \vdash C}}{\Delta', A \oplus B \vdash C}\ \oplus L}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_{A \oplus B} \quad \Longrightarrow \quad \dfrac{\dfrac{P}{\Delta \vdash A} \quad \dfrac{Q_1}{\Delta', A \vdash C}}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_A
$$

- Here: the first premise of the cut has the information
- Here: the second premise of the cut waits for it

Process Expressions for Internal Choice

- Process expressions

Expression	**Action**	**Continuation**
$c.\pi_1 \,;\, P$	send label π_1 along c	P
$c.\pi_2 \,;\, P$	send label π_2 along c	P
$\mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)$	receive π_1 or π_2 along c	Q_1 or Q_2

- Operational semantics

$$\mathsf{proc}(c, c.\pi_1 \,;\, P), \mathsf{proc}(e, \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) \longrightarrow \mathsf{proc}(c, P), \mathsf{proc}(e, Q_1)$$
$$\mathsf{proc}(c, c.\pi_2 \,;\, P), \mathsf{proc}(e, \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) \longrightarrow \mathsf{proc}(c, P), \mathsf{proc}(e, Q_2)$$

Typing Process Expressions

- Assign process expressions to usual right and left rules of sequent calculus

$$\frac{\Delta \vdash P :: (x : A)}{\Delta \vdash (x.\pi_1 \,;\, P) :: (x : A \oplus B)} \oplus R_1 \qquad \frac{\Delta \vdash P :: (x : B)}{\Delta \vdash (x.\pi_2 \,;\, P) :: (x : A \oplus B)} \oplus R_2$$

$$\frac{\Delta', x : A \vdash Q_1 :: (z : C) \quad \Delta', x : B \vdash Q_2 :: (z : C)}{\Delta', x : A \oplus B \vdash \mathsf{case}\, x\, (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) :: (z : C)} \oplus L$$

General Observations

- In a pair of matching right and left rules
  - the invertible rule carries no information, so it receives
  - the noninvertible rules makes a choice, so it sends
- From the perspective of the provider
  - positive connectives send ($\oplus, \mathbf{1}, \otimes, \exists$)
  - negative connectives receive ($\&, \multimap, \forall$)
- Client will carry out complementary action

Identity as Forwarding

- Identity identifies two channels ("forwarding")

$$\frac{}{A \vdash A}\ \text{id} \qquad \frac{}{y : A \vdash (x \leftarrow y) :: (x : A)}\ \text{id}$$

- Read: "x is implemented by y"
- Two alternative operational readings

$$\mathsf{proc}(d, P[d]), \mathsf{proc}(c, c \leftarrow d) \longrightarrow \mathsf{proc}(c, P[c])$$
$$\mathsf{proc}(c, c \leftarrow d), \mathsf{proc}(e, Q[c]) \longrightarrow \mathsf{proc}(e, Q[d])$$

- Arise from two different cut reductions, with id first or second premise

$$\frac{\overset{P}{\Delta \vdash A} \quad \frac{}{A \vdash A}\ \text{id}}{\Delta \vdash A}\ \text{cut} \implies \overset{P}{\Delta \vdash A} \qquad \frac{\frac{}{A \vdash A}\ \text{id} \quad \overset{Q}{\Delta', A \vdash C}}{\Delta', A \vdash C}\ \text{cut} \implies \overset{Q}{\Delta', A \vdash C}$$

Unit as Termination

- $\mathbf{1}$ is positive:
  - Right rule sends $(\mathsf{close}\,x)$
  - Left rule receives $(\mathsf{wait}\,x\,;\,Q)$
- Typing rules for new process expressions

$$\frac{}{\cdot \vdash \mathsf{close}\,x :: (x : \mathbf{1})}\;\mathbf{1}R \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash (\mathsf{wait}\,x\,;\,Q) :: (z : C)}\;\mathbf{1}L$$

- Operational reading from cut reduction

$$\mathsf{proc}(c, \mathsf{close}\,c), \mathsf{proc}(e, \mathsf{wait}\,c\,;\,Q) \longrightarrow \mathsf{proc}(e, Q)$$

Example: Bit Streams

- Generalize internal choice $A \oplus B$ to $\oplus\{\ell : A_\ell\}_{\ell \in L}$
  - Then $A \oplus B = \oplus\{\pi_1 : A, \pi_2 : B\}$
- Allow equirecursively defined types and processes

  $\mathit{bits} = \oplus\{\mathrm{b0} : \mathit{bits}, \mathrm{b1} : \mathit{bits}, \$: \mathbf{1}\}$

  $\cdot \vdash \mathit{six} :: (x : \mathit{bits})$

  $x \leftarrow \mathit{six} = x.\mathrm{b0}\ ;\ x.\mathrm{b1}\ ;\ x.\mathrm{b1}\ ;\ x.\$\ ;\ \mathrm{close}\, x$
- "Little endian": least significant bit comes first
- $\mathrm{proc}(c, c \leftarrow \mathit{six})$ does not reduce
  - Need client for interaction
  - Communication based on cut reduction is synchronous!

Example: Incrementing a Bit Stream

- Transduce bits representing n to those representing $n + 1$

$\mathit{bits} = \oplus\{\mathrm{b0} : \mathit{bits}, \mathrm{b1} : \mathit{bits}, \$: \mathbf{1}\}$

$y : \mathit{bits} \vdash \mathit{plus1} :: (x : \mathit{bits})$

$$
\begin{array}{l}
x \leftarrow \mathit{plus1} \leftarrow y = \\
\quad \mathrm{case}\ y\ (\ \mathrm{b0} \Rightarrow x.\mathrm{b1}\ ;\ x \leftarrow y \\
\qquad\qquad\ \ \mathrm{b1} \Rightarrow x.\mathrm{b0}\ ;\ x \leftarrow \mathit{plus1} \leftarrow y \\
\qquad\qquad\ \ \$ \Rightarrow x.\mathrm{b1}\ ;\ x.\$\ ;\ \mathrm{wait}\ y\ ;\ \mathrm{close}\ x\ )
\end{array}
$$

External Choice

- Provider receives for all negative type
- Example: external choice $A \mathbin{\&} B$

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

$$\frac{\Delta', A \vdash C}{\Delta', A \mathbin{\&} B \vdash C} \; \& L_1 \qquad \frac{\Delta', B \vdash C}{\Delta', A \mathbin{\&} B \vdash C} \; \& L_2$$

Computation of External Choice

- Information now flows from client to provider

$$\dfrac{\dfrac{\dfrac{P_1}{\Delta \vdash A} \quad \dfrac{P_2}{\Delta \vdash B}}{\Delta \vdash A \mathbin{\&} B}\ \&R \qquad \dfrac{\dfrac{Q}{\Delta', A \vdash C}}{\Delta', A \mathbin{\&} B \vdash C}\ \&L_1}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_{A \& B} \quad \Longrightarrow \quad \dfrac{\dfrac{P_1}{\Delta \vdash A} \quad \dfrac{Q}{\Delta', A \vdash C}}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_A$$

- Use the same process expressions

$$\mathsf{proc}(c, \mathsf{case}\ c\ (\pi_1 \Rightarrow P_1 \mid \pi_2 \Rightarrow P_2)), \mathsf{proc}(e, c.\pi_1 \ ;\ Q) \longrightarrow \mathsf{proc}(c, P_1), \mathsf{proc}(e, Q)$$
$$\mathsf{proc}(c, \mathsf{case}\ c\ (\pi_1 \Rightarrow P_1 \mid \pi_2 \Rightarrow P_2)), \mathsf{proc}(e, c.\pi_2 \ ;\ Q) \longrightarrow \mathsf{proc}(c, P_2), \mathsf{proc}(e, Q)$$

Example: A Binary Counter

- Show only part of the interface

  $ctr = \&\{\mathrm{inc} : ctr, \ldots\}$
- Messages in bit streams now become processes

  $y : ctr \vdash bit0 :: (x : ctr)$
  $y : ctr \vdash bit1 :: (x : ctr)$
  $\cdot \vdash zero :: (x : ctr)$
- Implementations

  $x \leftarrow bit0 \leftarrow y = \mathrm{case}\, x\, (\mathrm{inc} \Rightarrow x \leftarrow bit1 \leftarrow y)$
  $x \leftarrow bit1 \leftarrow y = \mathrm{case}\, x\, (\mathrm{inc} \Rightarrow y.\mathrm{inc}\, ;\, x \leftarrow bit0 \leftarrow y)$
  $x \leftarrow zero = \mathrm{case}\, x\, (\mathrm{inc} \Rightarrow y \leftarrow zero\, ;\, x \leftarrow bit1 \leftarrow y)$

Example: A Binary Counter

- Counting to two

  $\cdot \vdash \mathit{two} :: (x : \mathit{ctr})$

  $x \leftarrow \mathit{two} = x \leftarrow \mathit{zero}\ ;\ x.\mathrm{inc}\ ;\ x.\mathrm{inc}$

- This does compute since *zero* has a client

  $$\begin{aligned}\mathrm{proc}(c_0, c_0 \leftarrow \mathit{two}) \longrightarrow^* \ & \mathrm{proc}(c_2, c_2 \leftarrow \mathit{zero}),\\ & \mathrm{proc}(c_1, c_1 \leftarrow \mathit{bit1} \leftarrow c_2),\\ & \mathrm{proc}(c_0, c_0 \leftarrow \mathit{bit0} \leftarrow c_1)\end{aligned}$$

Session Type Summary

- Judgmental constructs, independent of type
  - Spawn (cut) $x \leftarrow P[x]$; $Q[x]$
  - Forward (id) $x \leftarrow y$
- Communication is synchronous
- From the perspective of the provider

Type	**Action**	**Continuation**
$A_1 \oplus A_2$	send π_i	A_i
$\mathbf{1}$	send close	*none*
$A \otimes B$	send $d : A$	B
$\exists x{:}\tau.\, B$	recv $v : \tau$	[v/x]B
$A_1 \,\&\, A_2$	recv π_i	A_i
$A \multimap B$	recv $d : A$	B
$\forall x{:}\tau.\, B$	recv $v : \tau$	[v/x]B
$!A$	recv $d : A$	fresh instance of A

Metatheory

- Type configurations $\Delta \vdash \mathcal{C} : \Delta'$
  - $\mathcal{C}$ uses all channels in Δ
  - $\mathcal{C}$ provides all channels in Δ'
- Allow recursive types and recursion

Theorem (Session Fidelity)

If $\Delta \vdash \mathcal{C} : \Delta'$ and $\mathcal{C} \longrightarrow \mathcal{C}'$ then $\Delta \vdash \mathcal{C}' : \Delta'$.

Theorem (Deadlock Freedom)

If $\cdot \vdash \mathcal{C} : \Delta'$ then either (i) all processes $\mathrm{proc}(c, P) \in \mathcal{C}$ are blocked on c, or (ii) $\mathcal{C} \longrightarrow \mathcal{C}'$ for some $\mathcal{C}'$.

Outline

- Linear logic, sequent calculus, and synchronous communication
- A calculus for asynchronous communication
- A shared memory interpretation
- Outlook (ongoing work)

Asynchronous Communication

- Synchronous
  - Derived from cut reduction
  - Sender and receiver proceed together
  - As in synchronous π-calculus
- Asynchronous
  - Sender dispatches message, proceeds immediately
  - Message is entered into channel buffer
  - Message order is guaranteed (unlike asynchronous π-calculus), to ensure session fidelity
  - Operational semantics uses two forms of semantic objects, $\mathrm{proc}(c, P)$ and $\mathrm{msg}(c, M)$
- Is there a proof-theoretic explanation for asynchronous communication?

π-Calculus

- Synchronous π-calculus (side remark: no forwarding!)

$$P ::= a\langle b\rangle.P \mid a(x).P \mid (P \mid Q) \mid (\nu x)P \mid 0 \mid !P$$

- Asynchronous π-calculus

$$P ::= a\langle b\rangle \mid a(x).P \mid (P \mid Q) \mid (\nu x)P \mid 0 \mid !P$$

- Asynchronous output action has no continuation

$$a\langle b\rangle.P \quad \simeq \quad a\langle b\rangle \mid P$$

- Employ the same observation in the logical setting!
- Continuation is proof of the premise
- Rules with no premise have no continuation!

Noninvertible Rules as Axioms

- Right rule example

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

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

- Left rule example

$$\frac{\Delta', A \vdash C}{\Delta', A \,\&\, B \vdash C}\ \& L_1 \qquad \frac{}{A \,\&\, B \vdash A}\ \& L_1^0$$

$$\frac{\Delta', B \vdash C}{\Delta', A \,\&\, B \vdash C}\ \& L_2 \qquad \frac{}{A \,\&\, B \vdash B}\ \& L_2^0$$

Simulating the Ordinary Rule

- Requires an analytic cut

$$\dfrac{\Delta \vdash A \quad \dfrac{}{A \vdash A \oplus B}\oplus R_1^0}{\Delta \vdash A \oplus B}\text{cut}_A \qquad \dfrac{\dfrac{}{A \mathbin{\&} B \vdash A}\mathbin{\&} L_1^0 \quad \Delta, A \vdash C}{\Delta, A \mathbin{\&} B \vdash C}\text{cut}_A$$

- With process expressions

$$\dfrac{}{y : A \vdash x.\pi_1(y) :: (x : A \oplus B)}\oplus R_1^0 \qquad \dfrac{}{y : A \mathbin{\&} B \vdash y.\pi_1(x) :: (x : A)}\mathbin{\&} R_1^0$$

- Replace output prefix by spawn

$$\begin{array}{rcll} x.\pi_1 \,;\, P[x] & \simeq & y \leftarrow P[y] \,;\, x.\pi_1(y) & (P[x] \text{ provides } x) \\ y.\pi_1 \,;\, Q[y] & \simeq & x \leftarrow y.\pi_1(x) \,;\, Q[x] & (Q[x] \text{ is client of } x) \end{array}$$

Multiplicative Axioms

- Multiplicative conjunction (sending a channel)

$$\dfrac{}{A, B \vdash A \otimes B}\ \otimes R^0 \qquad \dfrac{\Delta' \vdash B \quad \dfrac{\Delta \vdash A \quad \dfrac{}{A, B \vdash A \otimes B}\ \otimes R^0}{\Delta, B \vdash A \otimes B}\ \mathsf{cut}_A}{\Delta, \Delta' \vdash A \otimes B}\ \mathsf{cut}_B$$

- Linear implication (receiving a channel)

$$\dfrac{}{A, A \multimap B \vdash B}\ \multimap L^0 \qquad \dfrac{\dfrac{\Delta \vdash A \quad \dfrac{}{A, A \multimap B \vdash B}\ \multimap L^0}{\Delta, A \multimap B \vdash B}\ \mathsf{cut}_A \quad \Delta', B \vdash C}{\Delta, \Delta', A \multimap B \vdash C}\ \mathsf{cut}_B$$

Updating the Operational Semantics

- Sending is accomplished by a spawn
- Receiving selects continuation

$$\dfrac{\dfrac{}{A \vdash A \oplus B}\ \oplus R_1^0 \quad \dfrac{\begin{matrix} Q_1 \\ \Delta', A \vdash C \end{matrix} \quad \begin{matrix} Q_2 \\ \Delta', B \vdash C \end{matrix}}{\Delta', A \oplus B \vdash C}\ \oplus L}{\Delta', A \vdash C}\ \mathsf{cut}_{A \& B} \quad \Longrightarrow \quad \begin{matrix} Q_1 \\ \Delta', A \vdash C \end{matrix}$$

- Computationally, select branch and substitute continuation channel

$$\mathrm{proc}(c, c.\pi_1(d)), \mathrm{proc}(e, \mathrm{case}\ c\ (\pi_1(y) \Rightarrow Q_1[y] \mid \pi_2(y) \Rightarrow Q_2[y])) \longrightarrow \mathrm{proc}(e, Q_1[d])$$

$$\mathrm{proc}(c, c.\pi_2(d)), \mathrm{proc}(e, \mathrm{case}\ c\ (\pi_1(y) \Rightarrow Q_1[y] \mid \pi_2(y) \Rightarrow Q_2[y])) \longrightarrow \mathrm{proc}(e, Q_2[d])$$

Example Revisited: Bit Streams

- Recall

  $bits = \oplus\{b0 : bits, b1 : bits, \$: \mathbf{1}\}$

  $\cdot \vdash six :: (x : bits)$

  $x \leftarrow six = x.b0 \,;\, x.b1 \,;\, x.b1 \,;\, x.\$ \,;\, \mathrm{close}\, x$

- Asynchronously (writing cuts in reverse)

$$
\begin{aligned}
x \leftarrow six = {} & x_1 \leftarrow x.b0(x_1) \,; \\
& x_2 \leftarrow x_1.b1(x_2) \,; \\
& x_3 \leftarrow x_2.b1(x_3) \,; \\
& x_4 \leftarrow x_3.\$(x_4) \,; \\
& \mathrm{close}\, x_4
\end{aligned}
$$

- Execution

$$
\begin{aligned}
\mathrm{proc}(c_0, c_0 \leftarrow six) \longrightarrow^* {} & \mathrm{proc}(c_4, \mathrm{close}\, c_4), \\
& \mathrm{proc}(c_3, c_3.\$(c_4)), \\
& \mathrm{proc}(c_2, c_2.b1(c_3)), \\
& \mathrm{proc}(c_1, c_1.b1(c_2)), \\
& \mathrm{proc}(c_0, c_0.b0(c_1))
\end{aligned}
$$

Example Revisited: Binary Counter

- Recall

  $ctr = \&\{\text{inc} : ctr, \ldots\}$

  $y : ctr \vdash bit0 :: (x : ctr)$
  $y : ctr \vdash bit1 :: (x : ctr)$
  $\cdot \vdash zero :: (x : ctr)$

- With asynchronous message passing

  $x \leftarrow bit0 \leftarrow y = \text{case}\, x\, (\text{inc}(x') \Rightarrow x' \leftarrow bit1 \leftarrow y)$

  $$\begin{aligned} x \leftarrow bit1 \leftarrow y = \text{case}\, x\, (\text{inc}(x') \Rightarrow\ & y' \leftarrow y.\text{inc}(y')\ ;\\ & x' \leftarrow bit0 \leftarrow y') \end{aligned}$$

  $$\begin{aligned} x \leftarrow zero = \text{case}\, x\, (\text{inc}(x') \Rightarrow\ & y \leftarrow zero\ ;\\ & x' \leftarrow bit1 \leftarrow y) \end{aligned}$$

Summary: Asynchronous Semantics

- Process expressions and actions

Rules	Proc. Exp.	Action	Cont. Channel
$\oplus R^0_k$, $\& L^0_k$	$c.\pi_k(d)$	send label π_k	d
$\oplus L$, $\& R$	$\text{case } c\ (\pi_i(y) \Rightarrow P_i[y])_i$	recv label π_k	d
$\otimes R^0$, $\multimap L^0$	$\text{send } c\ \langle e, d\rangle$	send channel e	d
$\otimes L$, $\multimap R$	$\langle z, y\rangle \leftarrow \text{recv } c\ ;\ Q[z, y]$	recv channel e	d
$\mathbf{1}R$	$\text{close } c$	send close msg	*none*
$\mathbf{1}L$	$\text{wait } c\ ;\ Q$	recv close msg	*none*
cut	$x \leftarrow P[x]\ ;\ Q[x]$	spawn $P[a]$ (a fresh)	
id	$x \leftarrow y$	forward x to y	

Key Points: Asynchronous Semantics

- Force communication to be asynchronous by taking away continuation process from messages
- Logically, this means messages correspond to 0-premise rules ("axioms")
- Operationally, sending messages is accomplished by spawning a message process
- New form of cut reduction translates to asynchronous semantics
- Lose traditional cut elimination

Outline

- Linear logic, sequent calculus, and synchronous communication
- A calculus for asynchronous communication
- A shared memory interpretation
- Outlook (ongoing work)

Channels as Memory Addresses

- Previous implementations (Concurrent C0, SILL) use ad hoc queues to implement buffered channels
- Develop provable(?) implementation from first principles
- Concurrency/parallelism should be preserved
- Derived from substructural operational semantics [Pf'04]
- Now proc(c, P) — evaluate P with destination c
- New semantic artifact cell(c, V)
  - cell(c, V) — cell c holds value V
  - Values V to be defined

Memory Allocation

- Only cut (spawn) creates fresh channels

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

- Only cut (spawn) allocates memory
- Every address a has a unique $\text{proc}(a, P)$ or $\text{cell}(a, V)$
- Implementation would allocate an uninitialized $\text{cell}(a, \_)$

Internal Choice $A \oplus B$

- Process $c.\pi_k(d)$ writes $\pi_k(d)$ to destination c

$$\mathsf{proc}(c, c.\pi_k(d)) \longrightarrow \mathsf{cell}(c, \pi_k(d))$$

- Writing process terminates
- Process $\mathsf{case}\ c\ (\pi_i(y) \Rightarrow Q_i[y])_i$ reads contents

$$\mathsf{cell}(c, \pi_k(d)), \mathsf{proc}(e, \mathsf{case}\ c\ (\pi_i(y) \Rightarrow Q_i[y])_i) \longrightarrow \mathsf{proc}(e, Q_k[d])$$

- The reading process may block if there is no value!
- Due to linearity (uniqueness of client), cell is deallocated when read

Termination 1

- We replace "close" by $\langle\,\rangle$

$$\mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{cell}(c, \langle\,\rangle)$$
$$\mathsf{cell}(c, \langle\,\rangle), \mathsf{proc}(e, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(e, Q)$$

Example Revisited: Bit Streams

- Recall

  $bits = \oplus\{\mathrm{b0} : bits, \mathrm{b1} : bits, \$: \mathbf{1}\}$

  $\cdot \vdash six :: (x : bits)$

$$
\begin{aligned}
x \leftarrow six = \; & x_1 \leftarrow x.\mathrm{b0}(x_1) \; ; \\
& x_2 \leftarrow x_1.\mathrm{b1}(x_2) \; ; \\
& x_3 \leftarrow x_2.\mathrm{b1}(x_3) \; ; \\
& x_4 \leftarrow x_3.\$(x_4) \; ; \\
& \mathrm{close}\; x_4
\end{aligned}
$$

- Execution produces a simple linked list memory structure

$$
\begin{aligned}
\mathrm{proc}(c_0, c_0 \leftarrow six) \longrightarrow^* \; & \mathrm{cell}(c_4, \langle\,\rangle), \\
& \mathrm{cell}(c_3, \$(c_4)), \\
& \mathrm{cell}(c_2, \mathrm{b1}(c_3)), \\
& \mathrm{cell}(c_1, \mathrm{b1}(c_2)), \\
& \mathrm{cell}(c_0, \mathrm{b0}(c_1))
\end{aligned}
$$

Example Revisited: Incrementing a Bit Stream

- Recall

  $bits = \oplus\{\mathrm{b0} : bits, \mathrm{b1} : bits, \$: \mathbf{1}\}$

  $y : bits \vdash plus1 :: (x : bits)$

  $$
  \begin{array}{l}
  x \leftarrow plus1 \leftarrow y = \\
  \quad \mathrm{case}\, y\, (\ \mathrm{b0} \Rightarrow x.\mathrm{b1} \,;\, x \leftarrow y \\
  \qquad\qquad\ \ \mathrm{b1} \Rightarrow x.\mathrm{b0} \,;\, x \leftarrow plus1 \leftarrow y \\
  \qquad\qquad\ \ \$ \Rightarrow x.\mathrm{b1} \,;\, x.\$ \,;\, \mathrm{wait}\, y \,;\, \mathrm{close}\, x\ )
  \end{array}
  $$

- Asynchronous syntax

  $$
  \begin{array}{l}
  x \leftarrow plus1 \leftarrow y = \\
  \quad \mathrm{case}\, y\, (\ \mathrm{b0}(y') \Rightarrow x' \leftarrow x.\mathrm{b1}(x') \,;\, x' \leftarrow y' \\
  \qquad\qquad\ \ \mathrm{b1}(y') \Rightarrow x' \leftarrow x.\mathrm{b0}(x') \,;\, x' \leftarrow plus1 \leftarrow y' \\
  \qquad\qquad\ \ \$(y') \Rightarrow x' \leftarrow x.\mathrm{b1}(x') \,;\, x'' \leftarrow x'.\$(x'') \,;\, \mathrm{wait}\, y' \,;\, \mathrm{close}\, x''\ )
  \end{array}
  $$

- Forwarding

$$
\begin{array}{ll}
 & \mathrm{proc}(c_0, c_0 \leftarrow six), \mathrm{proc}(d_0, d_0 \leftarrow plus1 \leftarrow c_0) \\
\longrightarrow^{*} & \mathrm{cell}(c_4, \langle\,\rangle), \ldots, \mathrm{cell}(c_1, \mathrm{b1}(c_2)), \mathrm{cell}(c_0, \mathrm{b0}(c_1)), \mathrm{proc}(d_0, d_0 \leftarrow plus1 \leftarrow c_0) \\
\longrightarrow^{2} & \mathrm{cell}(c_4, \langle\,\rangle), \ldots, \mathrm{cell}(c_1, \mathrm{b1}(c_2)), \mathrm{proc}(d_1, d_1 \leftarrow c_1), \mathrm{cell}(d_0, \mathrm{b1}(d_1)) \\
\longrightarrow & \mathrm{cell}(c_4, \langle\,\rangle), \ldots, \mathrm{cell}(d_1, \mathrm{b1}(c_2)), \mathrm{cell}(d_0, \mathrm{b1}(d_1))
\end{array}
$$

Identity (Forwarding)

- Two immediately plausible implementations
- Copying values

$$\mathsf{cell}(d, V), \mathsf{proc}(c, c \leftarrow d) \longrightarrow \mathsf{cell}(c, V)$$

- Forwarding references with new form of cell contents

$$\mathsf{proc}(c, c \leftarrow d) \longrightarrow \mathsf{cell}(c, \mathsf{FWD}(d))$$
$$\mathsf{cell}(c, \mathsf{FWD}(d)), \mathsf{proc}(e, P[c]) \longrightarrow \mathsf{proc}(e, P[d])$$

Negative Propositions (Surprise!)

- Recall: $\mathrm{proc}(c, P)$ executes P with destination c
- With positive propositions ($\oplus$, $\mathbf{1}$, $\otimes$)
  - the provider writes to memory instead of sending
  - the client reads from memory instead of receiving
- With negative propositions ($\&$, $\multimap$)
  - the provider writes a continuation instead of receiving
  - the client reads and jumps to the continuation

External Choice

- Operationally

$$\mathsf{proc}(c, \mathsf{case}\, c\, (\pi_i(y) \Rightarrow Q_i[y])_i) \longrightarrow \mathsf{cell}(c, (\pi_i(y) \Rightarrow Q_i[y])_i)$$
$$\mathsf{cell}(c, (\pi_i(y) \Rightarrow Q_i[y])_i), \mathsf{proc}(d, c.\pi_k(d)) \longrightarrow \mathsf{proc}(d, Q_k[d])$$

- Process $\mathsf{proc}(d, c.\pi_k(d))$ may have to wait until cell is initialized
- New value corresponds to a jump table with an entry for every method π_i

Example Revisited: Binary Counter

- Recall

  $ctr = \&\{\mathrm{inc} : ctr, \ldots\}$

  $y : ctr \vdash bit0 :: (x : ctr)$
  $y : ctr \vdash bit1 :: (x : ctr)$
  $\cdot \vdash zero :: (x : ctr)$

- With asynchronous message passing

  $x \leftarrow bit0 \leftarrow y = \mathrm{case}\, x\, (\mathrm{inc}(x') \Rightarrow x' \leftarrow bit1 \leftarrow y)$

  $x \leftarrow bit1 \leftarrow y = \mathrm{case}\, x\, (\mathrm{inc}(x') \Rightarrow y' \leftarrow y.\mathrm{inc}(y')\, ;\, x' \leftarrow bit0 \leftarrow y')$

  $x \leftarrow zero = \mathrm{case}\, x\, (\mathrm{inc}(x') \Rightarrow y \leftarrow zero\, ;\, x' \leftarrow bit1 \leftarrow y)$

- Execution

  $\mathrm{proc}(c_0, c_0 \leftarrow zero), \mathrm{proc}(c_1, c_0.\mathrm{inc}(c_1)), \mathrm{proc}(c_2, c_1.\mathrm{inc}(c_2)) \longrightarrow^*$

  $\mathrm{proc}(d_1, d_1 \leftarrow zero), \mathrm{proc}(d_2, d_2 \leftarrow bit1 \leftarrow d_1), \mathrm{proc}(c_2, c_2 \leftarrow bit0 \leftarrow d_2)$

- Each process writes a continuation to memory next

Summary of Shared Memory Semantics

- Use locks or condition variables to implement blocking read?
- Operational semantics in tabular form

Rule	From		To
cut^a	$\mathrm{proc}(c, x \leftarrow P[x] \ ; \ Q[x])$	$\longrightarrow$	$\mathrm{proc}(a, P[a]), \mathrm{proc}(c, Q[a])$
id	$\mathrm{cell}(d, V), \mathrm{proc}(c, c \leftarrow d)$	$\longrightarrow$	$\mathrm{cell}(c, V)$
$\oplus R_i^0$	$\mathrm{proc}(c, c.\pi_k(d))$	$\longrightarrow$	$\mathrm{cell}(c, \pi_k(d))$
$\oplus L$	$\mathrm{cell}(c, \pi_k(d)), \mathrm{proc}(e, \mathrm{case}\ c\ (\pi_i(y) \Rightarrow Q_i[y])_i)$	$\longrightarrow$	$\mathrm{proc}(e, Q_k[d])$
$\& R$	$\mathrm{proc}(c, \mathrm{case}\ c\ (\pi_i(y) \Rightarrow P_i[y])_i)$	$\longrightarrow$	$\mathrm{cell}(c, (\pi_i(y) \Rightarrow P_i[y])_i)$
$\& L_i^0$	$\mathrm{cell}(c, (\pi_i(y) \Rightarrow P_i[y])_i), \mathrm{proc}(e, c.\pi_k(d))$	$\longrightarrow$	$\mathrm{proc}(e, P_k[d])$
$\mathbf{1}R$	$\mathrm{proc}(c, \mathrm{close}\ c)$	$\longrightarrow$	$\mathrm{cell}(c, \langle\,\rangle)$
$\mathbf{1}L$	$\mathrm{cell}(c, \langle\,\rangle), \mathrm{proc}(e, \mathrm{wait}\ c \ ; \ Q)$	$\longrightarrow$	$\mathrm{proc}(e, Q)$
$\otimes R^0$	$\mathrm{proc}(c, \mathrm{send}\ c\ \langle e, d \rangle)$	$\longrightarrow$	$\mathrm{cell}(c, \langle e, d \rangle)$
$\otimes L$	$\mathrm{cell}(c, \langle e, d \rangle), \mathrm{proc}(f, \langle z, y \rangle \leftarrow \mathrm{recv}\ c \ ; \ Q[z, y])$	$\longrightarrow$	$\mathrm{proc}(f, Q[e, d])$
$\multimap R$	$\mathrm{proc}(c, \langle z, y \rangle \leftarrow \mathrm{recv}\ c \ ; \ P[z, y])$	$\longrightarrow$	$\mathrm{cell}(c, \langle z, y \rangle.\ P[z, y])$
$\multimap L^0$	$\mathrm{cell}(c, \langle z, y \rangle.\ P[z, y]), \mathrm{proc}(f, \mathrm{send}\ c\ \langle e, d \rangle)$	$\longrightarrow$	$\mathrm{proc}(f, P[e, d])$

Metatheory

- Values

$$
\begin{array}{rcll}
V & ::= & \pi_k(d) & (\oplus) \\
& & \mid (\pi_i(y) \Rightarrow P_i[y])_i & (\&) \\
& & \mid \langle\,\rangle & (\mathbf{1}) \\
& & \mid \langle c, d\rangle & (\otimes) \\
& & \mid \langle x, y\rangle.\, P[x, y] & (\multimap)
\end{array}
$$

- Session fidelity and deadlock freedom continue to hold
- Bisimulation between asynchronous message-passing and shared memory semantics*

Beyond Linearity (Work in Progress)

- Allow controlled application of structural rules using modes of truth, arranged in a preorder [Benton'94][Reed'09]
- Adjunctions connect the different modes
- Example modes: L (linear), U (weakening & contraction)
- Logically, we have multicut

$$\frac{\Delta_{\mathsf{U}} \vdash A_{\mathsf{U}} \quad \Delta', A_{\mathsf{U}}, \ldots, A_{\mathsf{U}} \vdash C_m}{\Delta, \Delta' \vdash C_m}\ \mathsf{mcut}$$

- Operationally, a provider may have multiple clients
- Magically, the substructural operational semantics appears to continue to work!

Adjoint Sketch (Work in Progress)

- Every channel/address has an intrinsic mode
- Process objects remain ephemeral so they can evolve
- Cells inherit structural properties from channel/address
- Persistent semantic objects $!\phi$ are not consumed
- For example

$$
\begin{array}{ll}
\oplus R_i^0 & \mathrm{proc}(c_{\mathsf{U}}, c.\pi_k(d)) \longrightarrow !\mathrm{cell}(c_{\mathsf{U}}, \pi_k(d)) \\
\oplus L & !\mathrm{cell}(c_{\mathsf{U}}, \pi_k(d)), \mathrm{proc}(e_m, \mathrm{case}\; c\; (\pi_i(y) \Rightarrow Q_i[y])_i) \longrightarrow \mathrm{proc}(e_m, Q_k[d])
\end{array}
$$

$$
\begin{array}{ll}
\& R & \mathrm{proc}(c_{\mathsf{U}}, \mathrm{case}\; c\; (\pi_i(y) \Rightarrow P_i[y])_i) \longrightarrow !\mathrm{cell}(c_{\mathsf{U}}, (\pi_i(y) \Rightarrow P_i[y])_i) \\
\& L_i^0 & !\mathrm{cell}(c_{\mathsf{U}}, (\pi_i(y) \Rightarrow P_i[y])_i), \mathrm{proc}(e, c.\pi_k(d)) \longrightarrow \mathrm{proc}(e, P_k[d])
\end{array}
$$

- Provides a new shared memory semantics for a mixed linear/non-linear concurrent programming language

Outlook

- Is there a form of cut elimination for $\text{SEQ}^{\dagger}$?
- Reimplement session types on shared memory based on proof theoretic principles
- Forwarding? Optimizations? Scheduling?
- Relation to futures? [Halstead'85]
- Incorporating sharing [Balzer & Pf'17]

Summary

- Linear logic, sequent calculus, and synchronous communication
  - Provider/client distinction (intuitionistic)
  - Provider: positive types send, negative types receive
- A calculus for asynchronous communication
  - Sequent calculus with axioms for positive-right/negative-left rules
  - Send implemented via cut (spawn)
- A shared memory interpretation
  - Linear destination-passing style
  - Synchronization on memory read
  - Right rules write, left rules read
- Outlook (ongoing work)
  - Extend to structural session types
  - Incorporate mutable shared memory