Message-Passing Concurrency and Substructural Logics

Frank Pfenning

Department of Computer Science
Carnegie Mellon University

Tutorial, POPL 2018
January 8, 2018
Tutorial Objectives

- High-level abstractions for message-passing concurrent programming
- Session types as robust and expressive organizing force
- Substructural logics as a foundation for concurrency
- Concrete instantiation of ideas in one retro language, Concurrent C0
- Entry to literature
- Solved problems and current questions
Tutorial Approach

- Organized around **specification and programming**
- Three examples
  - Message streams (prime number sieve)
  - Concurrent data structure (queue)
  - Shared service (message buffer)
- Arrive at working code
- Extract essence and relate to logic
Tutorial Outline

■ Part I: Programming in Concurrent C0
  ■ Message streams (prime number sieve)
  ■ Concurrent data structure (queue)

■ Part II: Substructural Logics
  ■ Linear sequent calculus
  ■ Correspondence with message-passing concurrency

■ Part III: Sharing
  ■ Stratified session types
  ■ Manifest sharing via adjunctions
A process \textit{count} produces the stream of numbers 2, 3, 4, 5, \ldots up to some limit.

A process \textit{primes} receives the first number $p$ and passes it on, since it must be prime.

Then \textit{primes} spawns a new filter process which removes all multiples of $p$ from its input stream and recurses.

In steady state we have:

- one producer process (\textit{count})
- one filter process for each prime number already output (\textit{filter $p_i$})
- one process (\textit{primes}) that outputs only primes
A Session Type for Streams

- A data structure of lists might be described as
  \[ \text{list} = \{ \text{cons} : \text{int} \times \text{list}, \text{nil} : 1 \} \]
  \[ \text{cons}(2, \text{cons}(3, \ldots, \text{nil}())) : \text{list} \]

- We describe a \textit{stream} of integer messages along some communication channel analogously
  \[ \text{stream} = \bigoplus \{ \text{next} : \langle \text{!int} ; \text{stream} \rangle, \text{empty} : \langle \rangle \} \]
  next, 2, next, 3, \ldots, empty

  - \[ \bigoplus \{ l_1 : A_1, \ldots, l_n : A_n \} \text{ sends one of the } l_i \text{ and continues according to } A_i \]
  - \[ \langle A_1 ; \ldots ; A_n \rangle \text{ describes a sequence of interactions} \]
  - \text{!int sends an integer}
  - \langle \rangle \text{ closes the channels}
Creating a Stream (live: primes.c1)

```c
choice stream {
    <!int ; !choice stream> Next;
    < > Empty;
};
typedef <!choice stream> stream;

stream $c$ count(int n) {
    for (int i = 2; i < n; i++)
        //invariant $c : stream$
        {
            $c$.Next; /* $c : <!int ; stream> */
            send($c, i); /* $c : stream */
        }
    $c$.Empty; /* $c : < > */
    close($c);
}
```
Takeaways

- !<tp> sends a value $v : <tp>$
- !choice <name> sends a label (internal choice)
- $<ch>$ represents channel variables
- stream $l$ count(...) {...} forks a new process and provides a fresh channel $l : \text{stream}$ each time it is called
- Session type of $l$ changes during communication
- Channel types must be loop invariant
- Closing a channel terminates the providing process
Using a Stream (live: primes.c1)

```c
void print_stream(stream $s) {
    while (true) {
        switch ($s) {
            case Empty: { /* $s : < > */
                wait($s);
                print("\n");
                return;
            }
            case Next: { /* $s : <!int ; stream> */
                int x = recv($s); /* $s : stream */
                printint(x); print(" ");
                break;
            }
        }
    }
}

int main() {
    stream $nats = count(100);
    print_stream($nats);
    return 0;
}
```
Takeaways

- Client performs complementary actions to provider
- `switch ($<ch>) {...} receives and branches on label`
- `<tp> x = recv($<ch>); receives a basic data value`
- Channels behave **linearly**:
  - Guarantees **session fidelity**
  - All messages must be consumed
stream $t$ filter(int $p$, stream $s$) {
    switch ($s$) {
        case Empty: {
            wait($s$);
            $t$.Empty; close($t$);
        }
        case Next: {
            int $x$ = recv($s$);
            if ($x$ % $p$ != 0) {
                $t$.Next;
                send($t$, $x$);
            }
            $t$ = filter($p$, $s$); /* tail-call */
        }
    }
}
Takeaways

- Processes always **provide** channels
- Process may also **use** channels
- Provider/client send/receive actions are complementary
- Used channels must close before provided channels
- Tail calls can be used instead of loops
Generating Primes (live: primes.c1)

```c
stream $p$ primes(stream $s$) {
    switch ($s$) {
        case Empty: {
            wait($s$); $p$.Empty; close($p$);
        }
        case Next: {
            int $x$ = recv($s$);
            $p$.Next; send($p$, $x$);
            stream $t$ = filter($x$, $s$);
            $p$ = primes($t$);
        }
    }
}

int main() {
    stream $nats$ = count(100);
    stream $primes$ = primes($nats$);
    print_stream($primes$);
    return 0;
}
```
Takeaways

- `$<ch> = <proc>(...);` (spawn) creates fresh channel provided by new process instance
- `$<ch1> = $<ch2>` (forwarding)
  - Identifies channels `$<ch1>` and `$<ch2>`
  - Terminates provider of `$<ch1>`
  - Converse of spawn
- Strong identification between a process and the channel it provides
- Prime sieve creates $n + 2$ (lightweight) processes to produce the $n$th prime
- Implementation uses threads (C) or goroutines (Go)
Part I: Programming in Concurrent C0
- Message streams (prime number sieve)
- Concurrent data structure (queue)

Part II: Substructural Logics
- Linear sequent calculus
- Correspondence with message-passing concurrency

Part III: Sharing
- Stratified session types
- Manifest sharing via adjunctions
A Simple Buffer

- So far, all messages flow in the same direction through the network of processes.
- In contrast, a simple buffer process is responsive.

receive Ins, 1, Ins, 7, Del,
send Some, 1
receive Ins, 8, Del,
send Some, 7,
receive Del,
send Some, 8,
receive Del
send None, (close)

- Labels received signify an external choice.
External Choice

- External choice \( \&\{\ell_1 : A_1, \ldots, \ell_n : A_n\} \) receives one of the \( \ell_i \) and continues according to \( A_i \).
- \(?\text{int}\) receives an integer.
- The buffer interface:
  
  \[
  \text{buffer} = \&\{\text{Ins} : \langle ?\text{int} ; \text{buffer} \rangle, \text{Del} : \text{buffer_response}\} 
  \]

  \[
  \text{buffer_response} = \oplus\{\text{Some} : \langle !\text{int} ; \text{buffer} \rangle, \text{None} : \langle \rangle\} 
  \]

- Internal to the process, use a sequential imperative queue.
choice buffer {
  <!int ; ?choice buffer> Ins;
  <!choice buffer_response> Del;
};
choice buffer_response {
  <!int ; ?choice buffer> Some;
  < > None;
};
typedef struct queue* queue_t;
queue_t new_queue(int capacity)
//@requires 1 <= capacity && capacity < (1<<20);
//@ensures \result != NULL;
;
bool is_empty(queue_t q)
//@requires q != NULL;
;
bool is_full(queue_t q)
//@requires q != NULL;
;
/* enqueing will drop x if q full */
void enq(queue_t q, int x)
//@requires q != NULL;
;
/* dequeing will return 0 if q empty */
int deq(queue_t q)
//@requires q != NULL;
;
buffer $b$ new_buffer(int capacity) {
    queue_t $q$ = new_queue(capacity);
    while (true) {
        switch ($b$) {
            case Ins: { /* $b$ : ![int ; buffer> */
                int $x$ = recv($b$); /* $b$ : buffer */
                enq($q$, $x$);
                break;
            }
            case Del: { /* $b$ : ![choice buffer_response > */
                if (is_empty($q$)) {
                    $b$.None; close($b$);
                } else {
                    int $x$ = deq($q$);
                    $b$.Some; send($b$, $x$);
                }
                break;
            }
        }
    }
}
Takeaways

- Local process state may be complex
- Responsive systems rely on interaction between external and internal choice
- Processes offering an external choice have a concurrent object-oriented flavor
```c
int main () {
    buffer $b = new_buffer(10);
    $b.Ins; send($b,1);
    // $b.Ins; send($b,7);
    $b.Del;
    switch ($b) {
        case None: error("bad!");
        case Some: {
            assert(1 == recv($b));
            break;
        }
    }
    $b.Del;
    switch ($b) {
        case None: {
            wait($b);
            break;
        }
        case Some: error("very bad!");
    }
    print("Yes!\n");
    return 0;
}
```
Tutorial Outline

- **Part I: Programming in Concurrent C0**
  - Message streams (prime number sieve)
  - Concurrent data structure (queue)
- **Part II: Substructural Logics**
  - Linear sequent calculus
  - Correspondence with message-passing concurrency
- **Part III: Sharing**
  - Stratified session types
  - Manifest sharing via adjunctions
What Does This Have To Do With Substructural Logic?
Linear Sequent Calculus

- Linear sequent: from antecedents $\Delta$ prove succedent $C$

$A_1, \ldots, A_n \vdash C$

- Substructural: each antecedent must be used exactly once in proof (no weakening or contraction)
Judgmental Rules

- **Identity**: From antecedent $A$ we can prove succedent $A$

  $\Gamma \vdash A \quad \text{id}_A$

- **Cut**: If we can prove succedent $A$ we are allowed to assume antecedent $A$

  $\Delta \vdash A, \Delta', A \vdash C \quad \text{cut}_A$

  $\Delta', \Delta \vdash C$

- **Harmony**: identity* and cut are admissible
Each antecedent and the succedent represent a channel for communication

\[
x_1 : A_1, \ldots, x_n : A_n \vdash P :: (z : C)
\]

- Process \( P \) represents the proof of \( \Delta \vdash C \)
- Process \( P \) provides channel \( z : C \)
- Process \( P \) uses channels \( x_i : A_i \)
Annotate rule with process expressions

\[ \Delta \vdash P :: (x : A) \quad \Delta', x : A \vdash Q :: (z : C) \]

\[ \Delta', \Delta \vdash (x = P ; Q) :: (z : C) \] cut

Spawned process \( P \) provides along fresh channel \( x \)

Continuation \( Q \) is client of \( P \), using \( x \)

Other available channels (in \( \Delta', \Delta \)) are distributed between \( P \) and \( Q \).

Example (from prime sieve):

```plaintext
stream $nats = count(100);
stream $primes = primes($nats);
```
Identity as Forward

- Annotate rule with process expressions

\[
y : A \vdash (x = y) :: (x : A)
\]

- Forwarding process \((x = y)\) identifies \(x\) and \(y\)

- Example (stream constructor):

```plaintext
stream $l$ cons(int $x$, stream $k$) {
    $l$.Next;  /* $k : stream \vdash \downarrow$ $l : <!int ; stream> */
    send($l, $x);  /* $k : stream \vdash \downarrow$ $l : stream */
    $l = $k
}
```
Aside: $\pi$-Calculus

- Spawn $x = P ; Q$ corresponds to parallel composition with a private channel
  \[(\nu x)(P \mid Q)\]

- But the $\pi$-calculus does not express threads of control
- Identification $x = y$ does not have a direct analogue
As right and left rules of the sequent calculus

\[
\frac{\Delta \vdash A}{\Delta \vdash A \oplus B} \quad \lor R_1 \quad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B} \quad \lor R_2
\]

\[
\frac{\Delta', A \vdash C \quad \Delta', B \vdash C}{\Delta', A \oplus B \vdash C} \quad \lor L
\]
Key step in showing harmony is cut reduction

Replaces a cut at a compound proposition by cut(s) at smaller propositions

For example:

\[ \frac{D}{\Delta \vdash A} \quad \frac{E_1}{\Delta', A \vdash C} \quad \frac{E_2}{\Delta', B \vdash C} \implies \frac{\Delta', A \lor B \vdash C}{\Delta', \Delta \vdash C} \]

\[ \frac{D}{\Delta \vdash A} \quad \frac{E_1}{\Delta', A \vdash C} \implies \frac{\Delta', \Delta \vdash C}{\Delta', \Delta \vdash C} \]

\[ \frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta', \Delta \vdash C} \]

\[ \frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta', \Delta \vdash C} \]
Cut Reduction as the Engine of Computation

- Cut reduction is sequent calculus counterpart of substitution
- Cut reduction is more fine-grained than substitution
- Cut reduction is communication
- One premise of the cut has information to impart to the other premise

\[
\begin{align*}
\Delta \vdash A & \quad \Delta \vdash A \lor B \\
\Delta \vdash A \lor B & \quad \Delta', A \vdash C \\
\Delta', A \lor B \vdash C & \quad \Delta', B \vdash C \\
\Delta', A \lor B \vdash C & \quad \Delta', A \lor B \vdash C \\
\Delta', A \lor B \vdash C & \quad \Delta', A \lor B \vdash C
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash A & \quad \Delta' \vdash A \\
\Delta, A \vdash C & \quad \Delta, A \vdash C \\
\Delta, A \vdash C & \quad \Delta, A \vdash C
\end{align*}
\]
Internal Choice as Sending a Label

- As right and left rules of the sequent calculus
  \[
  \begin{align*}
  \Delta \vdash P :: (x : A) & \quad \forall R_1 \quad \Delta \vdash P :: (x : B) \quad \forall R_2 \\
  \Delta \vdash (x.\pi_1 ; P) :: (x : A \oplus B) & \quad \Delta \vdash (x.\pi_2 ; P) :: (x : A \oplus B)
  \end{align*}
  \]

  \[
  \begin{align*}
  \Delta', x : A \vdash Q_1 :: (z : C) & \quad \Delta', x : B \vdash Q_2 :: (z : C) \quad \forall L \\
  \Delta', x : A \oplus B \vdash \text{case } x (\pi_1 \Rightarrow Q_1 | \pi_2 \Rightarrow Q_2) :: (z : C)
  \end{align*}
  \]

- Observe how the type of the channel \(x\) changes

- Cut reduction as communication

  \[
  \begin{align*}
  (x.\pi_1 ; P) | (\text{case } x (\pi_1 \Rightarrow Q_1 | \pi_2 \Rightarrow Q_2)) \quad \rightarrow \quad P | Q_1 \\
  (x.\pi_2 ; P) | (\text{case } x (\pi_1 \Rightarrow Q_1 | \pi_2 \Rightarrow Q_2)) \quad \rightarrow \quad P | Q_2
  \end{align*}
  \]

- Concrete syntax in CC0 uses switch
Generalize to Labeled Internal Choice

- \( A \oplus B \triangleq \oplus\{\pi_1 : A, \pi_2 : B\} \)
- Generalized left and right rules

\[
\begin{align*}
\Delta \vdash (k \in L) \quad & \quad \Delta \vdash P :: (x : A_k) \quad \lor R_k \\
\Delta \vdash (x.k ; P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})
\end{align*}
\]

\[
\begin{align*}
(\forall \ell \in L) \quad & \quad \Delta', x : A_\ell \vdash Q_\ell :: (z : C) \quad \lor L \\
\Delta', x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \text{case } x (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)
\end{align*}
\]

- Generalized cut reduction

\[
(x.k ; P) \mid (\text{case } x (\ell \Rightarrow Q_\ell)_{\ell \in L}) \quad \rightarrow \quad P \mid Q_k
\]
External Choice

- Switches role of succedent (provider) and antecedent (client)
- As right and left rules of the sequent calculus

\[
\begin{align*}
\Delta &\vdash A & \Delta &\vdash B \\
&\quad \implies \quad & \Delta &\vdash A \land B & \quad \&R
\end{align*}
\]

\[
\begin{align*}
\Delta, A &\vdash C \\
&\quad \implies \quad & \Delta, A \land B &\vdash C & \quad \&L_1
\end{align*}
\]

\[
\begin{align*}
\Delta, B &\vdash C \\
&\quad \implies \quad & \Delta, A \land B &\vdash C & \quad \&L_2
\end{align*}
\]

- This time, the left rule has the information
External Choice as Receiving a Label

- Generalize to labeled external choice
- \(A \& B \triangleq \&\{\pi_1 : A, \pi_2 : B\}\)
- Generalized left and right rules

\[
\begin{align*}
(\forall \ell \in L) & \quad \Delta \vdash P_\ell :: (x : A_\ell) & & \& R \\
\Delta \vdash \text{case } x (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})& & \& R \\
(k \in L) & \quad \Delta, x : A_k \vdash Q :: (z : C) & & \& L_k \\
\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash (x.k ; Q) :: (z : C) & & \& L_k
\end{align*}
\]

- Same reduction!

\[
(\text{case } x (\ell \Rightarrow P_\ell)_{\ell \in L}) \mid (x.k ; Q) \quad \rightarrow \quad P_k \mid Q
\]

- Sending from client to provider
Multiplicative Unit

■ In sequent calculus

\[
\frac{}{1 \vdash 1_R} \quad \frac{\Delta' \vdash C}{\Delta', 1 \vdash C} 1_L
\]

■ Cut reduction

\[
\frac{1 \vdash 1_R \quad \Delta' \vdash C}{\Delta', 1 \vdash C} \quad \frac{\Delta', 1 \vdash C}{\Delta' \vdash C} \quad \Delta' \vdash C
\]

\[
\frac{1 \vdash 1_R \quad \Delta' \vdash C}{\Delta', 1 \vdash C} \quad \frac{\Delta', 1 \vdash C}{\Delta' \vdash C} \quad \Delta' \vdash C
\]
Process assignment to proofs

\[
\begin{align*}
\Delta' & \vdash Q :: (z : C) \\
\Delta', x : 1 & \vdash (\text{wait}(x) ; Q) :: (z : C)
\end{align*}
\]

\[1R\]

\[1L\]

Cut reduction to close channel and terminate process

\[
\text{close}(x) \mid (\text{wait}(x) ; Q) \rightarrow Q
\]
Existential Quantification

- In sequent calculus, for data types $\tau$

\[
\frac{\nu : \tau \quad \Delta \vdash A(\nu) \quad \exists R \quad \Delta' \quad A(c) \vdash C \quad \exists L_c}{\Delta \vdash \exists n : \tau. A(n) \quad \Delta', \exists n : \tau. A(n) \vdash C}
\]

- The $\exists R$ rule has information and sends

\[
\frac{\nu : \tau \quad \Delta \vdash P :: (x : A(\nu)) \quad \exists R \quad \Delta' \quad x : A(c) \vdash Q :: (z : C)}{\Delta \vdash (\text{send}(x, \nu) ; P) :: (x : \exists n : \tau. A(n)) \quad \Delta', x : \exists n : \tau. A(n) \vdash (c = \text{recv}(x) ; Q) :: (z : C) \quad \exists L_c}
\]

- Straightforward reduction

\[(\text{send}(x, \nu) ; P) \mid (c = \text{recv}(x) ; Q) \rightarrow P \mid [\nu/c]Q\]
Universal Quantification

- Dual to existential quantification
- Provider will receive a basic value
- Client will send a basic value
- In CC0, neither $\exists x: \tau. A$ nor $\forall x: \tau. A$ supports type dependence, that is, occurrence of $x$ in $A$
Summary of Correspondence

- Curry-Howard Isomorphism

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Session Type</th>
<th>Action</th>
<th>Cont</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \oplus B$</td>
<td>( \oplus { l : A_l }_{l \in L} )</td>
<td>send a label ( k \in L )</td>
<td>( A_k )</td>
</tr>
<tr>
<td>$A &amp; B$</td>
<td>( &amp; { l : A_l }_{l \in L} )</td>
<td>branch on received ( k \in L )</td>
<td>( A_k )</td>
</tr>
<tr>
<td>$1$</td>
<td>( \langle \rangle )</td>
<td>end session</td>
<td>–</td>
</tr>
<tr>
<td>$\exists x : \tau. A$</td>
<td>$\langle !\tau ; A \rangle$</td>
<td>send a value ( v : \tau )</td>
<td>$A$</td>
</tr>
<tr>
<td>$\forall x : \tau. A$</td>
<td>$\langle ?\tau ; A \rangle$</td>
<td>receive a value ( v : \tau )</td>
<td>$A$</td>
</tr>
</tbody>
</table>

- Cut is spawn (parallel composition)
- Identity is forward (channel identification)
- Logical connectives, from the provider point of view
Delegation: Sending Channels along Channels

- Extend Curry-Howard interpretation of multiplicative linear connectives $A \otimes B$ and $A \multimap B$

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Session Type</th>
<th>Action</th>
<th>Cont</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \otimes B$</td>
<td>$\langle !A ; B \rangle$</td>
<td>send a channel $y : A$</td>
<td>$B$</td>
</tr>
<tr>
<td>$A \multimap B$</td>
<td>$\langle ?A ; B \rangle$</td>
<td>receive a channel $y : A$</td>
<td>$B$</td>
</tr>
<tr>
<td>$A \oplus B$</td>
<td>$\oplus { \ell : A_\ell }_{\ell \in L}$</td>
<td>send a label $k \in L$</td>
<td>$A_k$</td>
</tr>
<tr>
<td>$A &amp; B$</td>
<td>$&amp; { \ell : A_\ell }_{\ell \in L}$</td>
<td>branch on received $k \in L$</td>
<td>$A_k$</td>
</tr>
<tr>
<td>$1$</td>
<td>$\langle \rangle$</td>
<td>end session</td>
<td>$-$</td>
</tr>
<tr>
<td>$\exists x : \tau. \ A$</td>
<td>$\langle !\tau ; A \rangle$</td>
<td>send a value $v : \tau$</td>
<td>$A$</td>
</tr>
<tr>
<td>$\forall x : \tau. \ A$</td>
<td>$\langle ?\tau ; A \rangle$</td>
<td>receive a value $v : \tau$</td>
<td>$A$</td>
</tr>
</tbody>
</table>
**Theorem:** (session fidelity / type preservation) All processes in a configuration remain well-typed and agree on the types of the channels connecting them.

**Theorem:** (deadlock freedom / global progress) If all linear processes are blocked then the computation is complete.

**Conjecture:** (local progress) [ongoing work] If all recursive types are inductive or coinductive

(i) communication along channels of inductive type will terminate, and

(ii) communication along channels of coinductive type will be productive
Part I: Programming in Concurrent C0
- Message streams (prime number sieve)
- Concurrent data structure (queue)

Part II: Substructural Logics
- Linear sequent calculus
- Correspondence with message-passing concurrency

Part III: Sharing
- Stratified session types
- Manifest sharing via adjunctions
Missing so far, logically: !A

Missing so far, operationally: sharing

Could we have a shared buffer with multiple producers and consumers?

So far all channels are linear: one provider, one client

Examples abound: key/value store, database, output device, input device, . . .
Stratification

- Stratify session types into **linear** and **shared**

```
ongoing research

Shared S ::= \( \uparrow A \mid S_1 \rightarrow S_2 \mid S_1 \times S_2 \mid \ldots \)

Linear A ::= \( \bigoplus \{ \ell : A_\ell \}_{\ell \in L} \mid \& \{ \ell : A_\ell \}_{\ell \in L} \mid \langle !\tau ; A \rangle \mid \langle ?\tau ; A \rangle \mid \langle !A ; B \rangle \mid \langle ?A ; B \rangle \mid \downarrow S \)
```

- Distinguish linear and shared channels
- Modeled on LNL [Benton’94]
- Traditional linear logic \( !A = \downarrow \uparrow A \)
Shared Buffer Interface

- Sharing is manifest in the type!
- The **linear** buffer interface:
  \[
  \text{buffer} = \&\{\text{Ins} : \langle ?\text{int} ; \text{buffer} \rangle, \text{Del} : \text{buffer\_response} \}
  \]
  \[
  \text{buffer\_response} = \oplus\{\text{Some} : \langle !\text{int} ; \text{buffer} \rangle, \text{None} : \langle \rangle \}
  \]
- The **shared** buffer interface:
  \[
  \text{sbuffer} = \uparrow\&\{\text{Ins} : \langle ?\text{int} ; \downarrow\text{sbuffer} \rangle, \text{Del} : \text{buffer\_response} \}
  \]
  \[
  \text{buffer\_response} = \oplus\{\text{Some} : \langle !\text{int} ; \downarrow\text{sbuffer} \rangle, \text{None} : \downarrow\text{sbuffer} \} \]
Operational Interpretation of Shifts (Provider)

- Process and channels go through shared and linear phases

$$x_S : \uparrow A$$, from the provider perspective
  - Multiple clients along shared channel $$x_S$$
  - Accept request to be acquired by one client along $$x_S$$
  - Interact exclusively according to linear session $$x_L : A$$

$$x_L : \downarrow S$$, from provider perspective
  - Detach from single client
  - Provide along resulting shared channel $$x_S : S$$

- The linear protocol between $$X = \uparrow \ldots \downarrow X$$ models a critical region with exclusive access to a shared resource
Operational Interpretation of Shifts (Client)

- Client performs matching interactions
- $x_S : \uparrow A$, from client perspective
  - Acquire exclusive access along $x_S$
  - Interact exclusively according to linear session $x_L : A$
- $x_L : \downarrow S$, from client perspective
  - Release provider
  - Revert to becoming one of many clients of $x_S : S$
choice buffer {
    <!int ; #; ?choice buffer> Ins;
    <!choice buffer_response> Del;
};

choice buffer_response {
    <!int ; #; ?choice buffer> Some;
    <# ; ?choice buffer> None;
};

typedef <!choice buffer> lbuffer;
typedef <# ; ?choice buffer> sbuffer;
• In concrete syntax, we only articulate $\uparrow A$ as `<# ; A>`
• $\downarrow S$ is implicit
sbuffer #b new_buffer(int capacity) {
    queue_t q = new_queue(capacity);
    while (true) {
        lbuffer $b = (lbuffer)#b;  /* accept */
        switch ($b) {
            case Ins: {  /* $b : <?int ; buffer> */
                int x = recv($b);  /* $b : buffer */
                enq(q, x);
                #b = (sbuffer)$b;  /* detach */
                break;
            }
            case Del: {  /* $b : !choice buffer_response */
                if (is_empty(q)) {
                    $b.None;
                    #b = (sbuffer)$b;  /* detach */
                } else {
                    int x = deq(q);
                    $b.Some; send($b, x);  /* detach */
                    #b = (sbuffer)$b;
                }
                break;
            }
        }
    }
}
Takeaways

- Shared channels have form #<ch>
- **Accept** is implemented as a cast $<ch> = (<tp>)#<ch>;
- **Detach** is implemented as a cast #<ch> = (<tp>)$<ch>;}
/* producer, from init to limit by step */
<> $c producer(int init, int step, int limit, sbuffer #b) {
    for (int i = init; i < limit; i = i+step)
        //invariant #b : sbuffer
        {
            lbuffer $b = (lbuffer)#b; /* acquire */
            $b.Ins; send($b, i);
            #b = (sbuffer)$b; /* release */
        }
    close($c);
}
/* consumer, of n messages */
<> $c consumer(int n, sbuffer #b) {
    while (n > 0)
        //invariant #b : sbuffer
        {
            lbuffer $b = (lbuffer)#b;
            $b.Del;
            switch ($b) {
                case None: {
                    print("."); flush();
                    #b = (sbuffer)$b;
                    break;
                }
                case Some: {
                    int x = recv($b);
                    print("<"); printint(x); flush();
                    n = n-1;
                    #b = (sbuffer)$b;
                    break;
                }
            }
        }
    print("\n"); close($c);
int main() {
    sbuffer #b = new_buffer(1000);
    <> $p1 = producer(0, 3, 30, #b);
    /* next line to sequentialize producers/consumers */
    // wait($p1);
    <> $p2 = producer(1, 3, 30, #b);
    // wait($p2);
    <> $p3 = producer(2, 3, 30, #b);
    // wait($p3);
    <> $c = consumer(30, #b);
    // wait($c);
    wait($p1);
    wait($p2);
    wait($p3);
    wait($c);
    return 0;
}
Takeaways

- Shared buffers are not treated linearly
- For session fidelity (type safety), type must be **equisynchronizing**
  - If released, must be at the same type at which it was acquired
  - Otherwise, waiting clients and provider may disagree on the shared channels type
- Could relax the restriction, with runtime type checking
Logical Interpretation

- $\uparrow$ and $\downarrow$ form an adjunction [Benton’94]
- $\downarrow\uparrow A$ is a comonad ($!A$)
- $\uparrow\downarrow S$ is a strong monad ($\bigcirc A$)
- Generalized in adjoint logic [Reed’09][Chargin et al.’17]
  - Adjoint propositions as stratified session types
  - Adjoint proofs as concurrent program
  - But: computation is not just proof reduction
Proof Construction and Deconstruction

- Matching accept/acquire is seen as constructing a proof by cut
- This proof will be reduced with cut reduction until . . .
- Matching detach/release is seen as deconstructing a cut into two separate proofs
- Shared channels limit nondeterminism in proof construction
- Shared processes are garbage-collected (reference counting clients)
- Deadlock is now possible!
Theorem: (session fidelity / type preservation) All processes in a configuration remain well-typed and agree on the types of the channels connecting them.

Theorem: (characterizing deadlocks / “progress”) If all linear processes are blocked then
  (i) either computation is complete, or
  (ii) all linear processes are waiting for a response to an acquire request (deadlock)
Dining Philosophers (files: dining_philosophers*.c1)
Summary: Linear Logic and Message-Passing

- Curry-Howard interpretation of intuitionistic linear logic [Caires & Pf’10]
  - Cut as parallel composition with private channel (spawn)
  - Identity as channel identification (forward)
  - Linear propositions as session types
  - Sequent proofs as process expressions
  - Cut reduction as communication
  - Guarantees session fidelity (preservation), local progress, and termination

- Extend to recursive types and processes [Toninho et al.’13]
  - Guarantee session fidelity and deadlock freedom (global progress)
  - Inductive and coinductive types [ongoing work]
Extend further to permit sharing [Balzer & Pf’17]
- Many more practical programs
- Interleave proof construction, reduction, deconstruction
- Proof construction may fail (deadlock)
Summary: Concurrent C0

- C0: type-safe and memory-safe subset of C
  - Extended with a layer of contracts
  - Using in first-year imperative programming course at CMU
  - Complemented by functional programming course in ML
  - See http://c0.typesafety.net

- Concurrent C0: session-type message-passing concurrency [Willsey et al.'16]
  - Examples from this tutorial
  - Many more examples, plus others in progress
  - svn co https://svn.concert.cs.cmu.edu/c0
    - User guest, pwd c0coffee
    - See c0/cc0-concur/README-concur.txt
    - Requires Standard ML (SML/NJ or mlton)

- Compiles to C (or Go)
Other Ongoing Research

- SILL: functional instantiation of ideas [Toninho et al.’13] [Toninho’15] [Griffith & Pf’15]
  - Includes polymorphism and subtyping, not yet sharing
- Adjoint logic [Reed’09]
  - Allows linear, affine, strict, and structural modes
  - Uniform concurrent semantics without sharing [Chargin et al.’17]
- Concurrent contracts [Gommerstadt et al.’18]
- Concurrent type theory [Caires et al.’12]
- A new foundation of object-oriented programming [Balzer & Pf’15]
- Automata and transducers in subsingleton fragment [DeYoung & Pf’16]
- Fault tolerance
Related Work (Small Sample)

- Seminal work on session types
  [Honda’93] [Honda, Vasconcelos & Kubo’98]
- Subtyping [Gay & Hole’05]
- Refinement types [Griffith & Gunter’13]
- Classical linear logic and session types [Wadler’12]
  [Toninho et al.’16]
- Links language [Lindley et al.’06–]
- Multiparty session types [Honda,Yoshida et al.’07–]
- Scribble protocol language [Yoshida et al.’09–]
- ABCD project [Gay, Wadler & Yoshida’13–’18]
Conclusion

- From (linear) logical origins to a new foundation for statically typed message-passing concurrency
- Primitives are not quite those of the $\pi$-calculus
- Simple, expressive, elegant, easy to use
- Robust across paradigms
  - Functional (SILL, Links)
  - Imperative (Concurrent C0)
  - Object-oriented (Mungo)
  - Language agnostic (Scribble)