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

Prime Number Sieve

- A process *count* produces the stream of numbers $2, 3, 4, 5, \ldots$ up to some limit
- A process *primes* receives the first number p and passes it on, since it must be prime
- Then *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 (*count*)
  - one filter process for each prime number already output (*filter* p_i)
  - one process (*primes*) that outputs only primes

A Session Type for Streams

- A data structure of lists might be described as

  $\mathit{list} = \{\mathsf{cons} : \mathsf{int} \times \mathit{list}, \mathsf{nil} : 1\}$

  $\mathsf{cons}(2, \mathsf{cons}(3, \ldots, \mathsf{nil}())) : \mathit{list}$

- We describe a *stream* of integer messages along some communication channel analogously

  $\mathit{stream} = \oplus\{\mathsf{next} : \langle !\mathsf{int} \,;\, \mathit{stream}\rangle, \mathsf{empty} : \langle\ \rangle\}$

  $\mathsf{next}, 2, \mathsf{next}, 3, \ldots, \mathsf{empty}$

  - $\oplus\{\ell_1 : A_1, \ldots, \ell_n : A_n\}$ sends one of the ℓ_i and continues according to A_i
  - $\langle A_1 \,;\, \ldots \,;\, A_n\rangle$ describes a sequence of interactions
  - $!\mathsf{int}$ sends an integer
  - $\langle\ \rangle$ closes the channels

Creating a Stream (live: primes.c1)

```
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 $1 count(...) {...}` forks a new process and provides a fresh channel `$1 : stream` each time it is called
- Session type of `$1` changes during communication
- Channel types must be loop invariant
- Closing a channel terminates the providing process

Using a Stream (live: primes.c1)

```
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

Filtering a Stream (live: `primes.c1`)

```
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`)

```
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 nth prime
- Implementation uses threads (C) or goroutines (Go)

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 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 ℓ_i and continues according to A_i
- ?int receives an integer
- The buffer interface:

  $\mathit{buffer} = \&\{\mathsf{Ins} : \langle ?\mathsf{int}\ ;\ \mathit{buffer}\rangle, \mathsf{Del} : \mathit{buffer\_response}\}$

  $\mathit{buffer\_response} = \oplus\{\mathsf{Some} : \langle !\mathsf{int}\ ;\ \mathit{buffer}\rangle, \mathsf{None} : \langle\ \rangle\}$
- Internal to the process, use a sequential imperative queue

Buffer Session Type (live: `lbuffer.c1`)

```
choice buffer {
  <?int ; ?choice buffer>   Ins;
  <!choice buffer_response> Del;
};
choice buffer_response {
  <!int ; ?choice buffer> Some;
  < >                     None;
};
```

Sequential Queue Interface (live: queue.h0)

```
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 Implementation (live: `lbuffer.c1`)

```
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

Buffer Client (live: `lbuffer.c1`)

```
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 Δ prove succedent C

$$\underbrace{A_1, \ldots, A_n}_{\Delta} \vdash C$$

- 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

$$\frac{}{A \vdash A}\ \mathsf{id}_A$$

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

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

- Harmony: identity* and cut are admissible

A Process Interpretation of Proofs

- Each antecedent and the succedent represent a channel for communication

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

- Process P represents the proof of $\Delta \vdash C$
- Process P provides channel $z : C$
- Process P uses channels $x_i : A_i$

Cut as Spawn

- Annotate rule with process expressions

$$\frac{\Delta \vdash P :: (x : A) \quad \Delta', x : A \vdash Q :: (z : C)}{\Delta', \Delta \vdash (x = P \,;\, Q) :: (z : C)}\ \mathrm{cut}$$

- Spawned process P provides along fresh channel x
- Continuation Q is client of P, using x
- Other available channels (in Δ', Δ) are distributed between P and Q.
- Example (from prime sieve):

```
stream $nats = count(100);
stream $primes = primes($nats);
```

Identity as Forward

- Annotate rule with process expressions

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

- Forwarding process $(x = y)$ identifies x and y
- Example (stream constructor):

```
stream $l cons(int x, stream $k) {
  $l.Next;       /* $k : stream |- $l : <!int ; stream> */
  send($l, x);   /* $k : stream |- $l : stream          */
  $l = $k
}
```

Aside: π-Calculus

- Spawn $x = P$ **;** Q corresponds to parallel composition with a private channel

$$(\nu x)(P \mid Q)$$

- But the π-calculus does not express threads of control
- Identification $x = y$ does not have a direct analogue

Internal Choice

- As right and left rules of the sequent calculus

$$\frac{\Delta \vdash A}{\Delta \vdash A \oplus B} \vee R_1 \qquad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B} \vee R_2$$

$$\frac{\Delta', A \vdash C \quad \Delta', B \vdash C}{\Delta', A \oplus B \vdash C} \vee L$$

Cut Reduction

- Key step in showing harmony is cut reduction
- Replaces a cut at a compound proposition by cut(s) at smaller propositions
- For example:

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta \vdash A\end{matrix}}{\Delta \vdash A \vee B}\ \vee R_1 \qquad \dfrac{\begin{matrix}\mathcal{E}_1\\ \Delta', A \vdash C\end{matrix} \quad \begin{matrix}\mathcal{E}_2\\ \Delta', B \vdash C\end{matrix}}{\Delta', A \vee B \vdash C}\ \vee L}{\Delta', \Delta \vdash C}\ \text{cut}_{A \vee B}$$

$$\longrightarrow \qquad \dfrac{\begin{matrix}\mathcal{D}\\ \Delta \vdash A\end{matrix} \quad \begin{matrix}\mathcal{E}_1\\ \Delta', A \vdash C\end{matrix}}{\Delta', \Delta \vdash C}\ \text{cut}_A$$

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

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Delta \vdash A\end{array}}{\Delta \vdash A \vee B}\,\vee R_1 \qquad \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Delta', A \vdash C\end{array} \quad \begin{array}{c}\mathcal{E}_2\\ \Delta', B \vdash C\end{array}}{\Delta', A \vee B \vdash C}\,\vee L}{\Delta', \Delta \vdash C}\,\mathsf{cut}_{A \vee B}$$

$$\longrightarrow \qquad \dfrac{\begin{array}{c}\mathcal{D}\\ \Delta \vdash A\end{array} \quad \begin{array}{c}\mathcal{E}_1\\ \Delta', A \vdash C\end{array}}{\Delta', \Delta \vdash C}\,\mathsf{cut}_A$$

Internal Choice as Sending a Label

- As right and left rules of the sequent calculus

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

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

- Observe how the type of the channel x changes
- Cut reduction as communication

$$\begin{array}{lcl} (x.\pi_1 \mathbin{;} P) \mid (\mathsf{case}\, x\, (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) & \longrightarrow & P \mid Q_1 \\ (x.\pi_2 \mathbin{;} P) \mid (\mathsf{case}\, x\, (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) & \longrightarrow & P \mid Q_2 \end{array}$$

- 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

$$\frac{(k \in L) \quad \Delta \vdash P :: (x : A_k)}{\Delta \vdash (x.k \,;\, P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})} \vee R_k$$

$$\frac{(\forall \ell \in L) \quad \Delta', x : A_\ell \vdash Q_\ell :: (z : C)}{\Delta', x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\, x\, (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)} \vee L$$

- Generalized cut reduction

$$(x.k \,;\, P) \mid (\mathsf{case}\, x\, (\ell \Rightarrow Q_\ell)_{\ell \in L}) \quad \longrightarrow \quad P \mid Q_k$$

External Choice

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

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

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

- This time, the left rule has the information

External Choice as Receiving a Label

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

$$\frac{(\forall \ell \in L) \quad \Delta \vdash P_\ell :: (x : A_\ell)}{\Delta \vdash \mathsf{case}\, x\, (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})} \;\&R$$

$$\frac{(k \in L) \quad \Delta, x : A_k \vdash Q :: (z : C)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash (x.k \,;\, Q) :: (z : C)} \;\&L_k$$

- Same reduction!

$$(\mathsf{case}\, x\, (\ell \Rightarrow P_\ell)_{\ell \in L}) \mid (x.k \,;\, Q) \quad \longrightarrow \quad P_k \mid Q$$

- Sending from client to provider

Multiplicative Unit

- In sequent calculus

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta' \vdash C}{\Delta', \mathbf{1} \vdash C}\ \mathbf{1}L$$

- Cut reduction

$$\frac{\dfrac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \quad \dfrac{\begin{array}{c}\mathcal{E}\\ \Delta' \vdash C\end{array}}{\Delta', \mathbf{1} \vdash C}\ \mathbf{1}L}{\Delta' \vdash C}\ \mathrm{cut}_{\mathbf{1}} \qquad \longrightarrow \qquad \begin{array}{c}\mathcal{E}\\ \Delta' \vdash C\end{array}$$

Unit as End of Session

- Process assignment to proofs

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

- Cut reduction to close channel and terminate process

$$\mathsf{close}(x) \mid (\mathsf{wait}(x)\ \mathbf{;}\ Q) \quad \longrightarrow \quad Q$$

Existential Quantification

- In sequent calculus, for data types τ

$$\frac{v : \tau \quad \Delta \vdash A(v)}{\Delta \vdash \exists n{:}\tau.\, A(n)} \; \exists R \qquad \frac{\Delta', A(c) \vdash C}{\Delta', \exists n{:}\tau.\, A(n) \vdash C} \; \exists L^c$$

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

$$\frac{v : \tau \quad \Delta \vdash P :: (x : A(v))}{\Delta \vdash (\mathsf{send}(x, v) \,;\, P) :: (x : \exists n{:}\tau.\, A(n))} \; \exists R$$

$$\frac{\Delta', x : A(c) \vdash Q :: (z : C)}{\Delta', x : \exists n{:}\tau.\, A(n) \vdash (c = \mathsf{recv}(x) \,;\, Q) :: (z : C)} \; \exists L^c$$

- Straightforward reduction

$$(\mathsf{send}(x, v) \,;\, P) \mid (c = \mathsf{recv}(x) \,;\, Q) \;\longrightarrow\; P \mid [v/c]Q$$

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

Linear Propositions	Session Types
Sequent Proofs	Process Expressions
Cut Reduction	Computation

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

Proposition	**Session Type**	**Action**	**Cont**
$A \oplus B$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send a label $k \in L$	A_k
$A \mathbin{\&} B$	$\&\{\ell : A_\ell\}_{\ell \in L}$	branch on received $k \in L$	A_k
$\mathbf{1}$	$\langle\,\rangle$	end session	–
$\exists x{:}\tau.\, A$	$\langle !\tau \,;\, A\rangle$	send a value $v : \tau$	A
$\forall x{:}\tau.\, A$	$\langle ?\tau \,;\, A\rangle$	receive a value $v : \tau$	A

Delegation: Sending Channels along Channels

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

Proposition	Session Type	Action	Cont
$A \otimes B$	$\langle !A \,;\, B \rangle$	send a channel $y : A$	B
$A \multimap B$	$\langle ?A \,;\, B \rangle$	receive a channel $y : A$	B
$A \oplus B$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$	send a label $k \in L$	A_k
$A \mathbin{\&} B$	$\&\{\ell : A_\ell\}_{\ell \in L}$	branch on received $k \in L$	A_k
$\mathbf{1}$	$\langle\, \rangle$	end session	–
$\exists x{:}\tau.\, A$	$\langle !\tau \,;\, A \rangle$	send a value $v : \tau$	A
$\forall x{:}\tau.\, A$	$\langle ?\tau \,;\, A \rangle$	receive a value $v : \tau$	A

Metatheoretic Properties

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

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

Sharing

- 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

$$\begin{array}{llll} \text{Shared} & S & ::= & \uparrow A \quad \overbrace{\mid S_1 \to S_2 \mid S_1 \times S_2 \mid \ldots}^{\text{ongoing research}} \\ \text{Linear} & A & ::= & \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \\ & & & \mid \langle !\tau \,;\, A \rangle \mid \langle ?\tau \,;\, A \rangle \\ & & & \mid \langle !A \,;\, B \rangle \mid \langle ?A \,;\, B \rangle \\ & & & \mid \downarrow S \end{array}$$

- 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:

  $\mathit{buffer} = \&\{\mathsf{Ins} : \langle ?\mathsf{int}\ ;\ \mathit{buffer}\rangle, \mathsf{Del} : \mathit{buffer\_response}\}$

  $\mathit{buffer\_response} = \oplus\{\mathsf{Some} : \langle !\mathsf{int}\ ;\ \mathit{buffer}\rangle, \mathsf{None} : \langle\ \rangle\}$
- The shared buffer interface:

  $\mathit{sbuffer} = \uparrow\&\{\mathsf{Ins} : \langle ?\mathsf{int}\ ;\ \downarrow\mathit{sbuffer}\rangle, \mathsf{Del} : \mathit{buffer\_response}\}$

  $\mathit{buffer\_response} = \oplus\{\mathsf{Some} : \langle !\mathsf{int}\ ;\ \downarrow\mathit{sbuffer}\rangle, \mathsf{None} : \downarrow\mathit{sbuffer}\}$

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 $x_S : S$

Shared Buffer Interface (live: `sbuffer.c1`)

```
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;
```

Takeaways

- In concrete syntax, we only articulate $\uparrow A$ as `<# ; A>`
- $\downarrow S$ is implicit

Shared Buffer Implementation (live: sbuffer.c1)

```
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>;`

Shared Buffer Clients (file: sbuffer-test.c1)

```
/* 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);
}
```

Testing a Shared Buffer (file: `sbuffer-test.c1`)

```
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!

Metatheoretic Properties, Including Sharing

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]

Summary: Linear Logic and Message-Passing

- 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 `c0c0ffee`
    - 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 π-calculus
- Simple, expressive, elegant, easy to use
- Robust across paradigms
  - Functional (SILL, Links)
  - Imperative (Concurrent C0)
  - Object-oriented (Mungo)
  - Language agnostic (Scribble)