Polarized Substructural Session Types

Frank Pfenning & Dennis Griffith

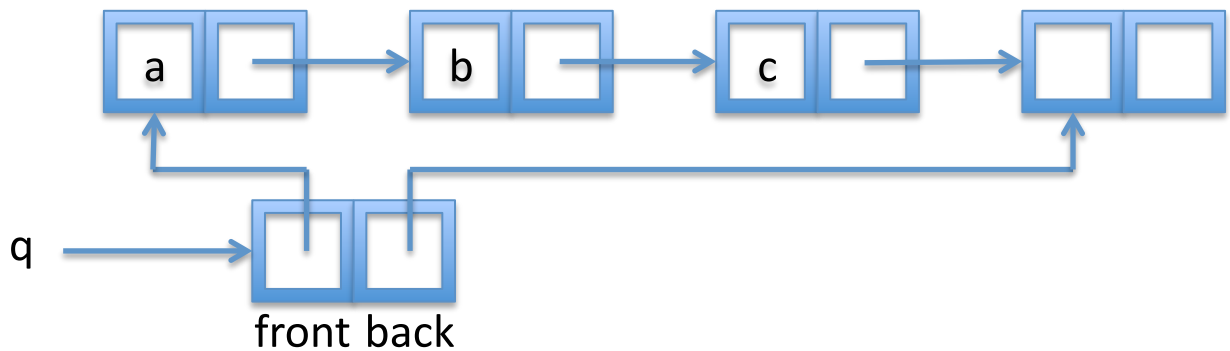
[Bernardo Toninho, Luís Caires]

Outline

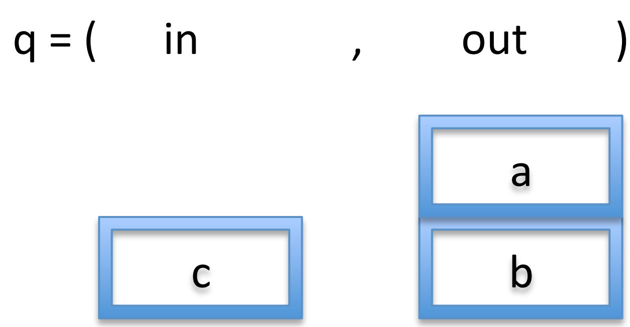
- Example: implementing queues
- Linear session types
  - A Curry-Howard correspondence
- Linear, affine, and shared channels
  - Substructural adjoint logic
- Synchronous & asynchronous communication
  - Polarization
- Synthesis in polarized adjoint logic
- Conclusion

Example: Implementing Queues

- Queues, imperatively

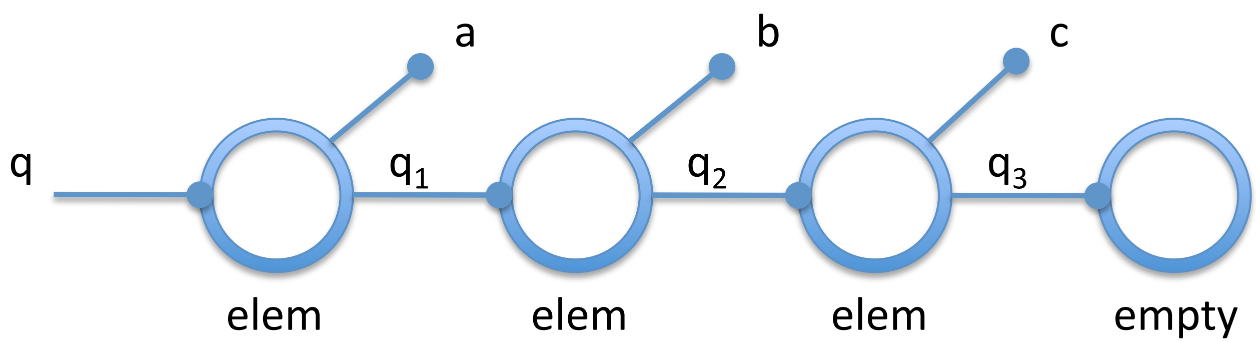


- Queues, functionally

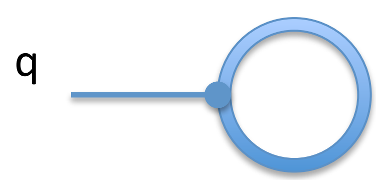


Example: Implementing Queues

- Queues, concurrently

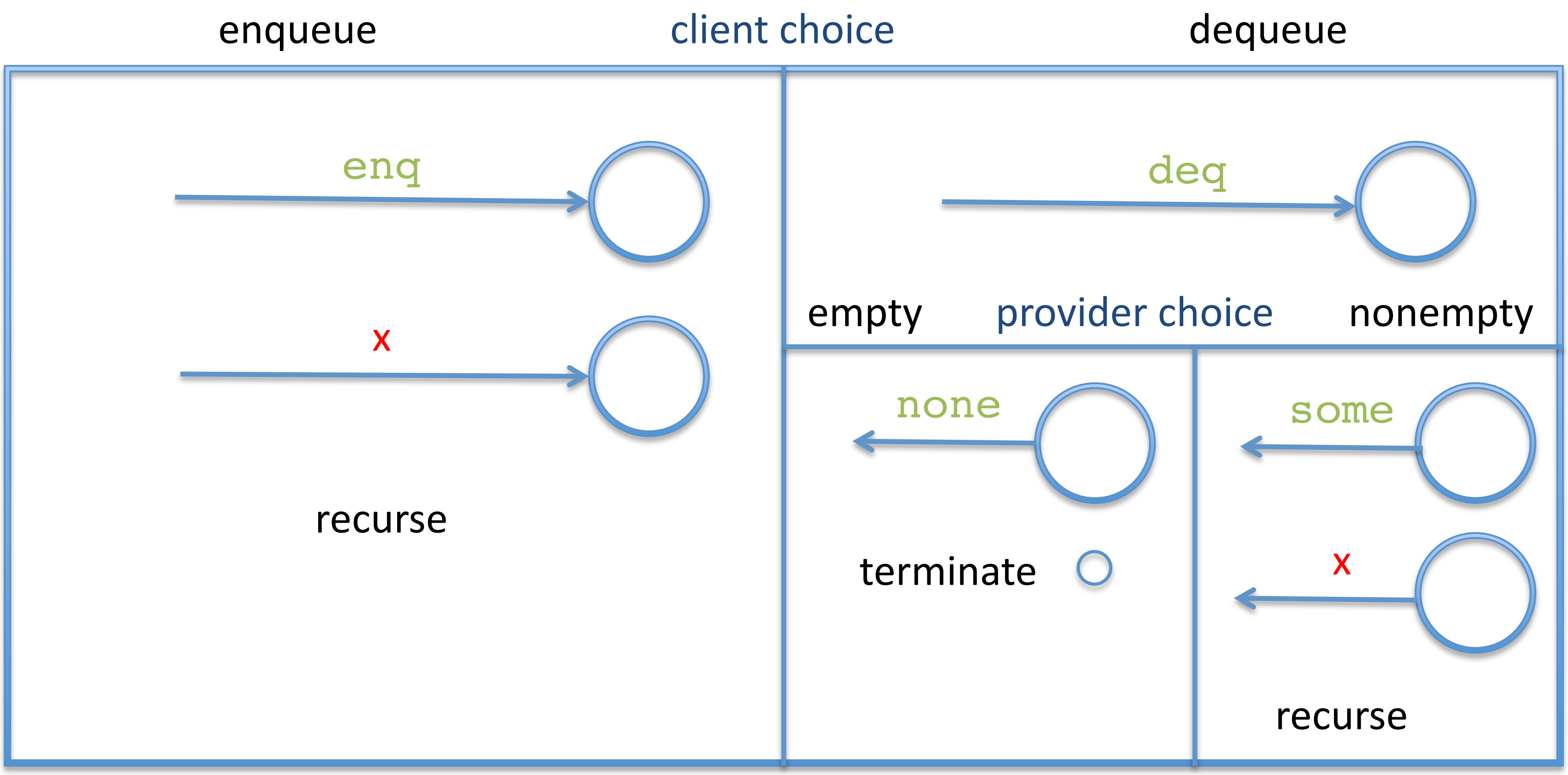


- How do we interact with a queue?



Queue Interface

- Interaction protocol



Linear Session Types

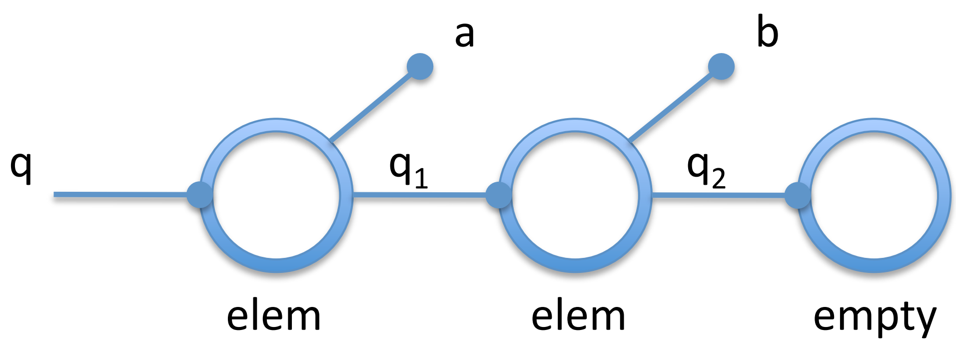
- Interface specification

```
queue A = &{enq: A —o queue A,
            deq: ⊕{none: 1, some: A ⊗ queue A}};
```

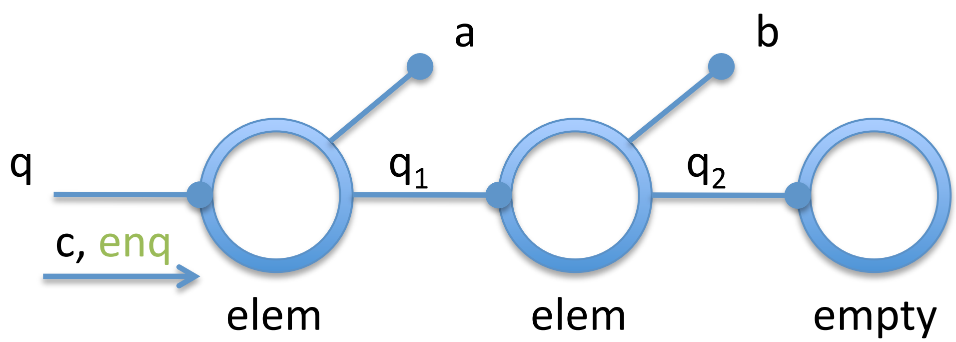
$\&\{lab_i : A_i\}_i$	External choice (receive) between lab_i, continue as A_i
A —o B	Receive channel of type A, continue as B
$\oplus\{lab_i : A_i\}_i$	Internal choice (send) between lab_i, continue as A_i
1	Terminate
$A \otimes B$	Send channel of type A, continue as B

Sample Run

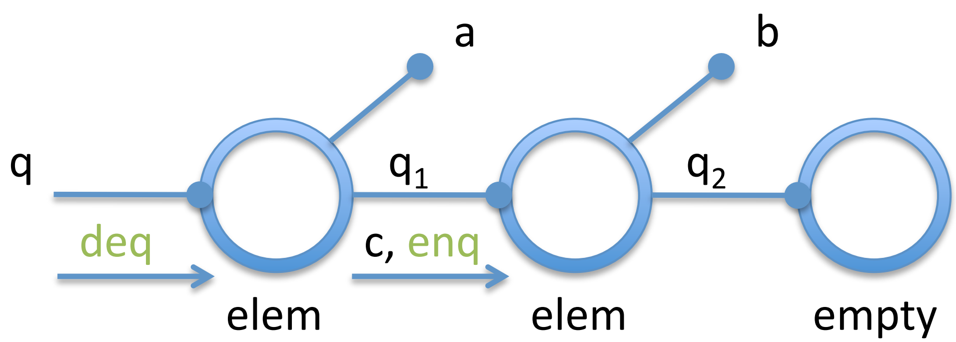
enqueue c, then dequeue



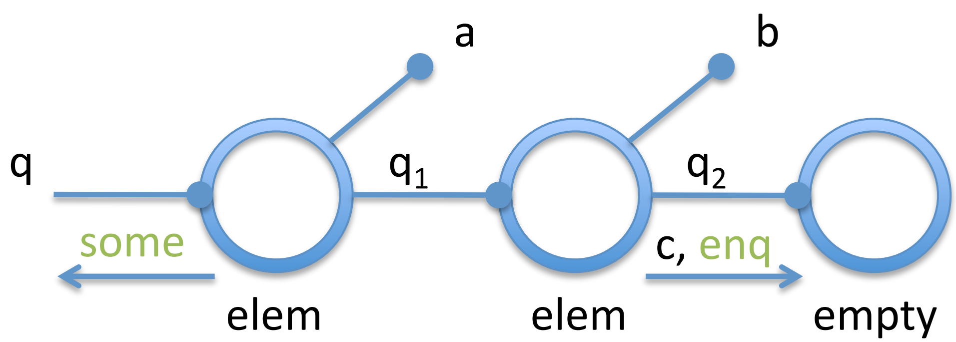
Sample Run



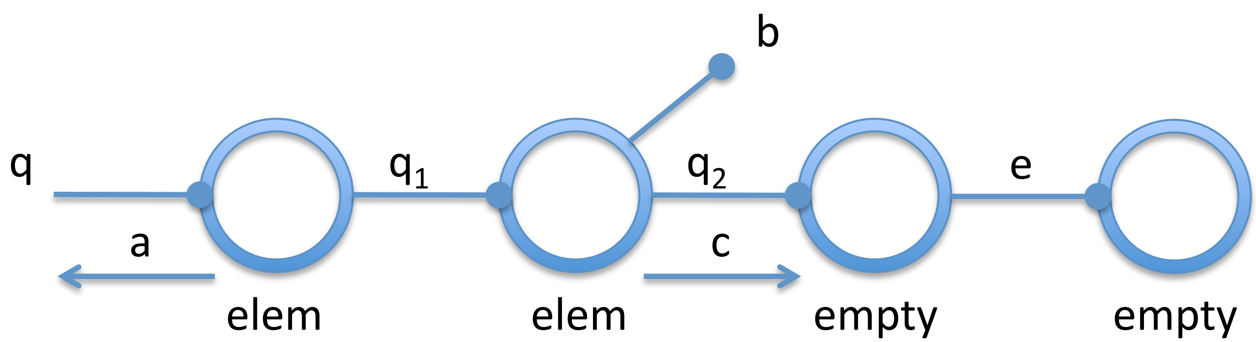
Sample Run



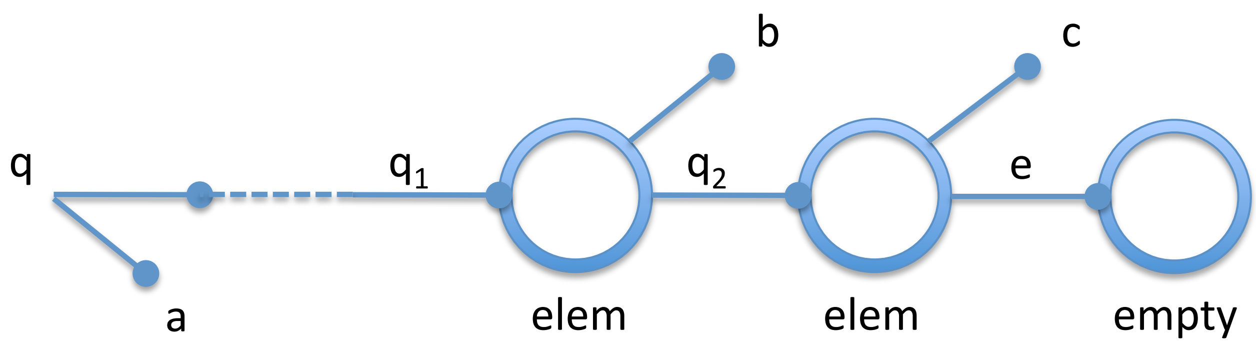
Sample Run



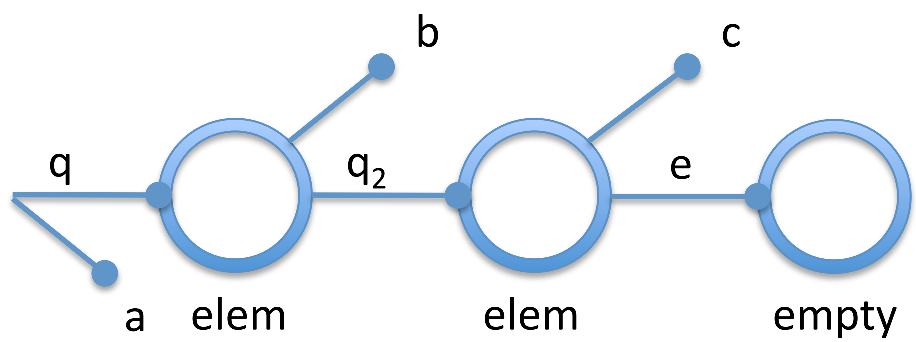
Sample Run



Sample Run



Sample Run



Outline

- Example: implementing queues
- Linear session types
  - A Curry-Howard correspondence
- Linear, affine, and shared channels
  - Substructural adjoint logic
- Synchronous & asynchronous communication
  - Polarization
- Synthesis in polarized adjoint logic
- Conclusion

Linear Session Types

- Typing, from the provider's perspective

$\texttt{c : }\&\{\texttt{lab}_i\texttt{:A}_i\}_i$	$\texttt{case c }\{\texttt{lab}_i \Rightarrow \texttt{P}_i\}_i$
$\texttt{c : A} \multimap \texttt{B}$	$\texttt{x} \leftarrow \texttt{recv c ; P}_x$
$\texttt{c : }\oplus\{\texttt{lab}_i\texttt{:A}_i\}_i$	$\texttt{c.lab}_j\texttt{ ; P}$
`c : 1`	`close c`
$\texttt{c : A} \otimes \texttt{B}$	`send c d ; P`

- Client's perspective is dual
- Process declarations $\texttt{p : }\{\texttt{A} \leftarrow \texttt{A}_1\texttt{, ..., A}_n\}$

  p provides A, uses A_1, …, A_n

  $\texttt{c} \leftarrow \texttt{p} \leftarrow \texttt{d}_1\texttt{, ..., d}_n \texttt{ = } \mathit{body}$

  where c:A and d_1:A_1, …, d_n:A_n

Implementation in SILL

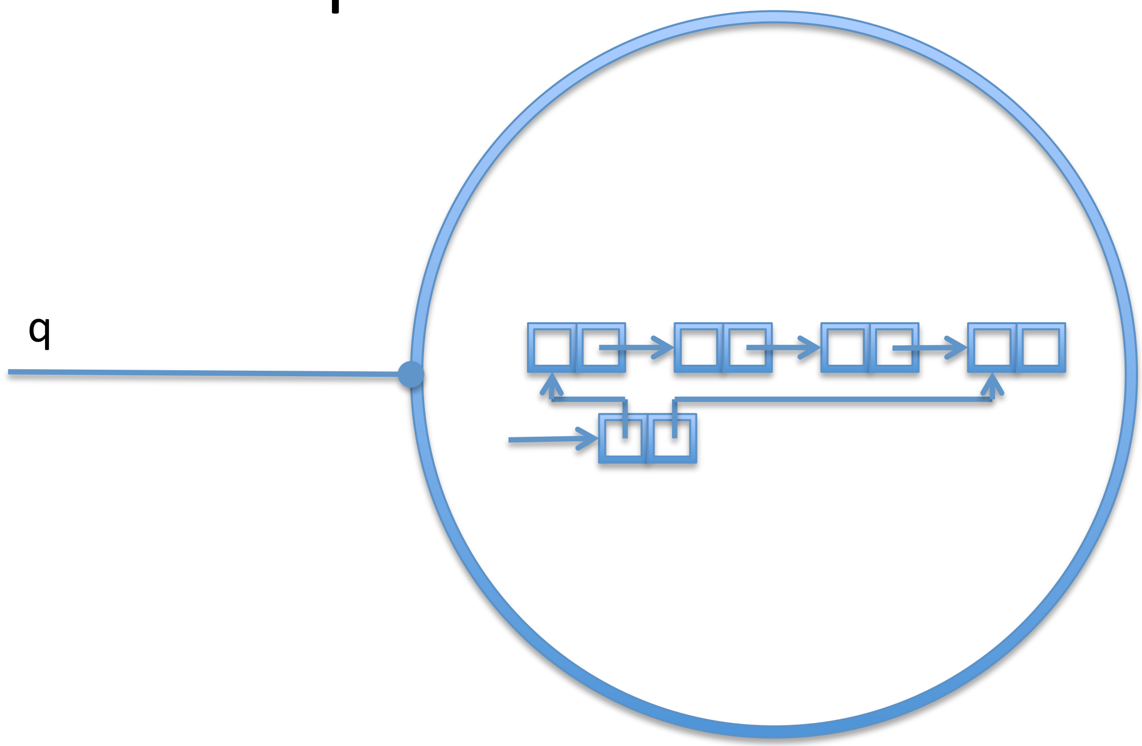
```
queue A = &{enq: A —o queue A,
            deq: ⊕{none: 1, some: A ⊗ queue A}};

elem : {queue A ← A, queue A};
q ← elem ← x, r =
  case q
  | enq ⇒ y ← recv q ;
          r.enq ; send r y ;
          q ← elem ← x, r
  | deq ⇒ q.some ; send q x ;
          q ← r
empty : {queue A};
q ← empty =
  case q
  | enq ⇒ x ← recv q ;
          e ← empty ;
          q ← elem ← x, e
  | deq ⇒ q.none ; close q
```

Some Observations

- Communication is bidirectional
- Enqueue has O(1) span, O(n) work
- Dequeue has O(1) span, O(1) work
- Everything is linear
  - Queue data structure must preserve elements
- Interface is abstract

Interface is Abstract

- Another implementation



The Curry-Howard Correspondence

- Curry [1934]
  - Propositions as simple types
  - Intuitionistic Hilbert proofs as combinators
  - Combinator reduction as computation
- Howard [1969]
  - Propositions as simple types
  - Intuitionistic natural deductions as programs
  - Proof reduction as computation

For Linear Logic

- Linear propositions as session types
- Sequent proofs as concurrent programs
- Cut reduction as communication

Intuitionistic Linear Logic

- Basic linear sequent calculus judgment

$$A_1, \ldots, A_n \vdash A$$

  - With resources A_1, …, A_n we can prove A
  - Each linear hypothesis must be used exactly once
- Classical linear logic also possible

  [Wadler 2012, Caires, Pf, Toninho 2012]

Proofs as Processes

- With processes:

$$c_1{:}A_1, \ldots, c_n{:}A_n \vdash P :: (c{:}A)$$

  – Labeled hypotheses / channels $c_i{:}A_i$ used by P
  – Labeled conclusion / channel c:A provided by P
  – Process P communicates along channels c_i and c
- Strong identification of process with channel along which it offers
  – Channel c as "process id"

Judgmental Rules of Sequent Calculus

- Judgmental rules generic over propositions
- Define the meaning of sequents themselves

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

- Silently re-order linear hypotheses
- They are inverses
  - Cut: if you can prove A, you may use A
  - Identity: if you may use A, you can prove A

Cut as Process Composition

$$\frac{\Delta \vdash P_a :: (a{:}A) \quad \Delta', a{:}A \vdash Q_a :: (c : C)}{\Delta, \Delta' \vdash (a \leftarrow P_a \,;\, Q_a) :: (c : C)}\ \text{cut}$$

- $(a \leftarrow P_a\ ;\ Q_a)$ spawns P_b, continues as Q_b
  - P_b and Q_b communicate along fresh private channel b
- In π-calculus:

$$(a \leftarrow P_a \,;\, Q_a) \equiv (\nu a)(P_a \mid Q_a)$$

Identity as Process Forwarding

$$\frac{}{a : A \vdash (c \leftarrow a) :: (c : A)} \text{ id}$$

- Operationally
  - Substitute channel a for c in client of (c : A)
  - Process (c ← a) terminates
- No direct equivalent in π-calculus
- Implementation
  - c tells its client to use a instead
  - c terminates

External Choice

- In sequent calculus, connectives have right and left rules
  - Right rules define how to prove a proposition
  - Left rules define how to use a proposition
- External choice A & B

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

External Choice

- External choice, with processes

$$\frac{\Delta \vdash P :: (c : A) \quad \Delta \vdash Q :: (c : B)}{\Delta \vdash \mathsf{case}\ c\ \{\mathsf{inl} \Rightarrow P \mid \mathsf{inr} \Rightarrow Q\} :: (c : A \mathbin{\&} B)}\ \&R$$

$$\frac{\Delta, c{:}A \vdash R :: (e : C)}{\Delta, c{:}A \mathbin{\&} B \vdash c.\mathsf{inl}\ ;\ R :: (e : C)}\ \&L_1 \qquad \frac{\Delta, c{:}B \vdash R :: (e : C)}{\Delta, c{:}A \mathbin{\&} B \vdash c.\mathsf{inr}\ ;\ R :: (e : C)}\ \&L_2$$

- For cut reduction (= communication), client will send either label inl or inr

External Choice

- For programming, we use generalized form

$$\frac{\{\Delta \vdash P_i :: (c : A_i)\}_i}{\Delta \vdash \mathsf{case}\ c\ \{lab_i \Rightarrow P_i\}_i :: (c : \&\{lab_i : A_i\}_i)}\ \&R$$

$$\frac{\Delta, c{:}A_k \vdash R :: (e : C)}{\Delta, c{:}\&\{lab_i : A_i\}_i \vdash c.lab_k\ ;\ R :: (e : C)}\ \&L_k$$

- Client sends one of the provided labels
- Provider branches based on the received label

Closing a Channel

- Closing a channel = terminating provider proc.
- Logically

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

- Process assignment

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

- close sends a token ‘end’, wait receives it

Outline

- Example: implementing queues
- Linear session types
  - A Curry-Howard correspondence
- Linear, affine, and shared channels
  - Substructural adjoint logic
- Synchronous & asynchronous communication
  - Polarization
- Synthesis in polarized adjoint logic
- Conclusion

The Price of Linearity

- How do we deallocate a queue?

```
queue A = &{enq: A —o queue A,
            deq: ⊕{none: 1, some: A ⊗ queue A}};
dealloc : {1 ← queue A};
```

- Not implementable: elements are linear!
- Need element consumer, A —o 1

```
dealloc : {1 ← queue A, A —o 1};
```

- Not implementable: consumer must be reusable!

Channel Modes

- c_U unrestricted, can be reused arbitrarily
  - Logically: permits weakening and contraction
- c_F affine, need not be used
  - Logically: permits weakening
- c_L linear, must be used
- Notation: U > F > L
  - Mode is greater if more structural properties hold

Shifting Between Modes

- $\uparrow_k^m$ A$_k$ converts from k to higher mode m
- $\downarrow_m^r$ A$_r$ converts from r to lower mode m
- Propositions are stratified

$$
\begin{array}{llcl}
\text{Mode} & m & ::= & \mathsf{U} \mid \mathsf{F} \mid \mathsf{L} \\
\text{Prop.} & A_m & ::= & A_m \mathbin{\&} B_m \mid A_m \multimap B_m \\
 & & \mid & A_m \oplus B_m \mid A_m \otimes B_m \mid \mathbf{1} \\
 & & \mid & \uparrow_k^m A_k \quad (m > k) \\
 & & \mid & \downarrow_m^r A_r \quad (r > m)
\end{array}
$$

```
dealloc : {1 ← queue A, ↑^U_L(A —o 1)};
```

Of Course!

- The exponential modality !A is decomposed

$$!A_{\mathsf{L}} = \downarrow^{\mathsf{U}}_{\mathsf{L}} \underbrace{\uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}}_{\mathsf{U}}$$

[Benton'94][Reed'09]

- Decomposition reduces "administrative" code

Deallocation, Shared Consumer

```
dealloc : {1 ← queue A, ↑U_L (A —o 1)};

u ← dealloc ← q, d_U =
  q.deq ;
  case q
  | none ⇒ wait q ; close u
  | some ⇒ x ← recv q ;
           f ← shift d_U ;
           send f x ; wait f ;
           u ← dealloc ← q, d_U
```

Nonlinear reuse of d_U

Deallocation, Affine Elements

- Deallocate queue with affine elements

```
dealloc : {1 ← queue (↓^F_L A_F)};

u ← dealloc ← q =
  q.deq ;
  case q
  | none ⇒ wait q ; close u
  | some ⇒ x ← recv q ;
             y_F ← shift x ;
             u ← dealloc ← q
```

Affine y_F not used

Multimodal Sequents

- Ψ is multimodal context (unordered)

$$\Psi \quad ::= \quad \cdot \mid \Psi, c_m : A_m$$

- Write $\Psi \geq m$ if $k \geq m$ for all $c_k : A_k$ in Ψ
- Critical invariant

$$\Psi \vdash C_m \quad \text{presupposes} \quad \Psi \geq m$$

  - Otherwise, cut elimination fails
  - Example: linear antecedent with affine succedent

Multimodal Sequent Calculus

- Cut and identity are generalized

$$\frac{\Psi \geq m \quad m \geq r \quad \Psi \vdash A_m \quad \Psi', A_m \vdash C_r}{\Psi, \Psi' \vdash C_r}\ \text{cut} \qquad \frac{}{A_m \vdash A_m}\ \text{id}$$

- Unrestricted and affine antecedents
  - Satisfy structural rules (implicitly or explicitly)
- Cut elimination, identity expansion hold

Shifting Rules

- ↑R: Ψ ≥ m > k implies Ψ ≥ k
- ↓L: r > m ≥ k implies r ≥ k

$$\frac{\Psi \vdash A_k}{\Psi \vdash {\uparrow}_k^m A_k} \uparrow R \qquad \frac{k \geq r \quad \Psi, A_k \vdash C_r}{\Psi, {\uparrow}_k^m A_k \vdash C_r} \uparrow L$$

$$\frac{\Psi \geq m \quad \Psi \vdash A_m}{\Psi \vdash {\downarrow}_k^m A_m} \downarrow R \qquad \frac{\Psi, A_m \vdash C_r}{\Psi, {\downarrow}_k^m A_m \vdash C_r} \downarrow L$$

Multimodal Session Types

- Works well for programming
  - Operate directly on linear and affine channels
- Every left/right rule corresponds to exactly one action
- Linear channels more expressive than affine ones
  - Ensures data elements will not be dropped
  - But sometimes, garbage collection is helpful
- Shared (unrestricted) channels
  - Important for persistent services
  - Currently only shifting connectives
  - Why and how to integrate unrestricted connectives?

Outline

- Example: implementing queues
- Linear session types
  - A Curry-Howard correspondence
- Linear, affine, and shared channels
  - Substructural adjoint logic
- Synchronous & asynchronous communication
  - Polarization
- Synthesis in polarized adjoint logic
- Conclusion

Message Buffers

```
queue A = &{enq: A —o queue A,
            deq: ⊕{none: 1, some: A ⊗ queue A}};
```

- Assume asynchronous communication
- What is the bound on the buffer size?
- With this type, unbounded!
  - Arbitrary sequence enq, x_1, enq, x_2, …
- Might want to enforce some synchronization

Send vs Receive, Logically

- Left and right rules match, by construction
  - Right sends and left receives, or vice versa
  - Cut reduction is communication
- If a right rule for a connective is invertible*
  - Rule application has no information content
  - Corresponds to receiving information
- If a right rule for a connective is noninvertible*
  - Rule application involves a choice
  - Corresponds to sending information about choice
- That's all there is [Andreoli'92]

Polarization

- Polarization [Girard'91,Laurent'99]
  - Makes direction of communication explicit
  - Negative = invertible = receive
  - Positive = noninvertible = send

$$\begin{array}{llll} \text{Neg.} & A^- & ::= & A^- \mathbin{\&} B^- \mid A^+ \multimap B^- \mid {\uparrow}A^+ \\ \text{Pos.} & A^+ & ::= & A^+ \oplus B^+ \mid A^+ \otimes B^+ \mid \mathbf{1} \mid {\downarrow}A^- \end{array}$$

- $\uparrow A^+$ receive shift, then send
- $\downarrow A^-$ send shift, then receive

Expression Synchronization

- Minimal shifts = maximal asynchrony

```
queue⁻ A⁺ = &{enq: A⁺ —o queue⁻ A⁺,
              deq: ↑⊕{none: 1, some: A⁺ ⊗ ↓queue⁻ A⁺}};
```

- Double shift = explicit synchronization

```
queue⁻ A⁺ = &{enq: A⁺ —o ↑↓queue⁻ A⁺,
              deq: ↑⊕{none: 1, some: A⁺ ⊗ ↓queue⁻ A⁺}};
```

Explicit Synchronization

```
queue⁻ A⁺ = &{enq: A⁺ ⊸ ↑↓queue⁻ A⁺,
              deq: ↑⊕{none: 1, some: A⁺ ⊗ ↓queue⁻ A⁺}};
```

$A^- = \uparrow\downarrow B^-$ receives shift, sends shift, then receives

$A^+ = \downarrow\uparrow B^+$ sends shift, receives shift, then sends

- Second shift acts as an acknowledgment
- Arises from purely logical principles
- More efficient than one ack for every send
- Buffer bound now 3, one of

  enq, x, shift | shift | deq, shift | none, end | some, x, shift

Proof Theory of Synchronization

- Intuitionistic natural deduction does not fix call-by-name or call-by-value
- Linear sequent calculus does not fix synchronization
  - No commuting conversions = synchronicity
  - Commuting past positives = asynchronous output
  - Commuting past negatives = nonblocking input? [Guenot'14]
- Polarization clarifies in both cases

Implementation in SILL

```
queue A = &{enq: A ⊸ ↑↓queue A,
             deq: ↑⊕{none: 1, some: A ⊗ ↓queue A}};

elem : {queue A ← A, queue A};
q ← elem ← x, r =
  case q
  | enq ⇒ y ← recv q ;
          r.enq ; send r y ; send r shift ;
          shift ← recv r ;
          q ← elem ← x, r
  | deq ⇒ shift ← recv q ;
          q.some ; send q x ; send q shift ;
          q ← r
```

- Shifts may be “implicit coercions”

Rules for Polarity Shifts

- (*) rules are invertible, others noninvertable

$$\frac{\Psi \vdash A^+}{\Psi \vdash \uparrow A^+} \uparrow R^* \quad \frac{\Psi, A^+ \vdash C}{\Psi, \uparrow A^+ \vdash C} \uparrow L$$

$$\frac{\Psi \vdash A^-}{\Psi \vdash \downarrow A^-} \downarrow R \quad \frac{\Psi, A^- \vdash C}{\Psi, \downarrow A^- \vdash C} \downarrow L^*$$

- These are exactly the same as for mode shifts!

Outline

- Example: implementing queues
- Linear session types
  - A Curry-Howard correspondence
- Linear, affine, and shared channels
  - Substructural adjoint logic
- Synchronous & asynchronous communication
  - Polarization
- Synthesis in polarized adjoint logic
- Conclusion

A Unified System

- Add polarity to multimodal system
- Allow m = k in $\uparrow^{\mathtt{m}}_{\mathtt{k}}$ and $\downarrow^{\mathtt{m}}_{\mathtt{k}}$ so $\uparrow = \uparrow^{\mathtt{m}}_{\mathtt{m}}$, $\downarrow = \downarrow^{\mathtt{m}}_{\mathtt{m}}$

$$\frac{\Psi \vdash A_k^+}{\Psi \vdash \uparrow_k^m A_k^+} \uparrow R \qquad \frac{k \geq r \quad \Psi, A_k^+ \vdash C_r}{\Psi, \uparrow_k^m A_k^+ \vdash C_r} \uparrow L$$

$$\frac{\Psi \geq m \quad \Psi \vdash A_m^-}{\Psi \vdash \downarrow_k^m A_m^-} \downarrow R \qquad \frac{\Psi, A_m^- \vdash C_r}{\Psi, \downarrow_k^m A_m^- \vdash C_r} \downarrow L$$

Polarized Substructural Session Types

- Polarized adjoint logic satisfies
  - Cut elimination
  - Identity expansion
- Polarized substructural session types
  - Admit arbitrary recursive types
  - Session fidelity (preservation) and progress
  - Determinism (confluence), modulo termination
  - Preliminary syntax (implicit shifts)
  - Populating unrestricted stratum with connectives?

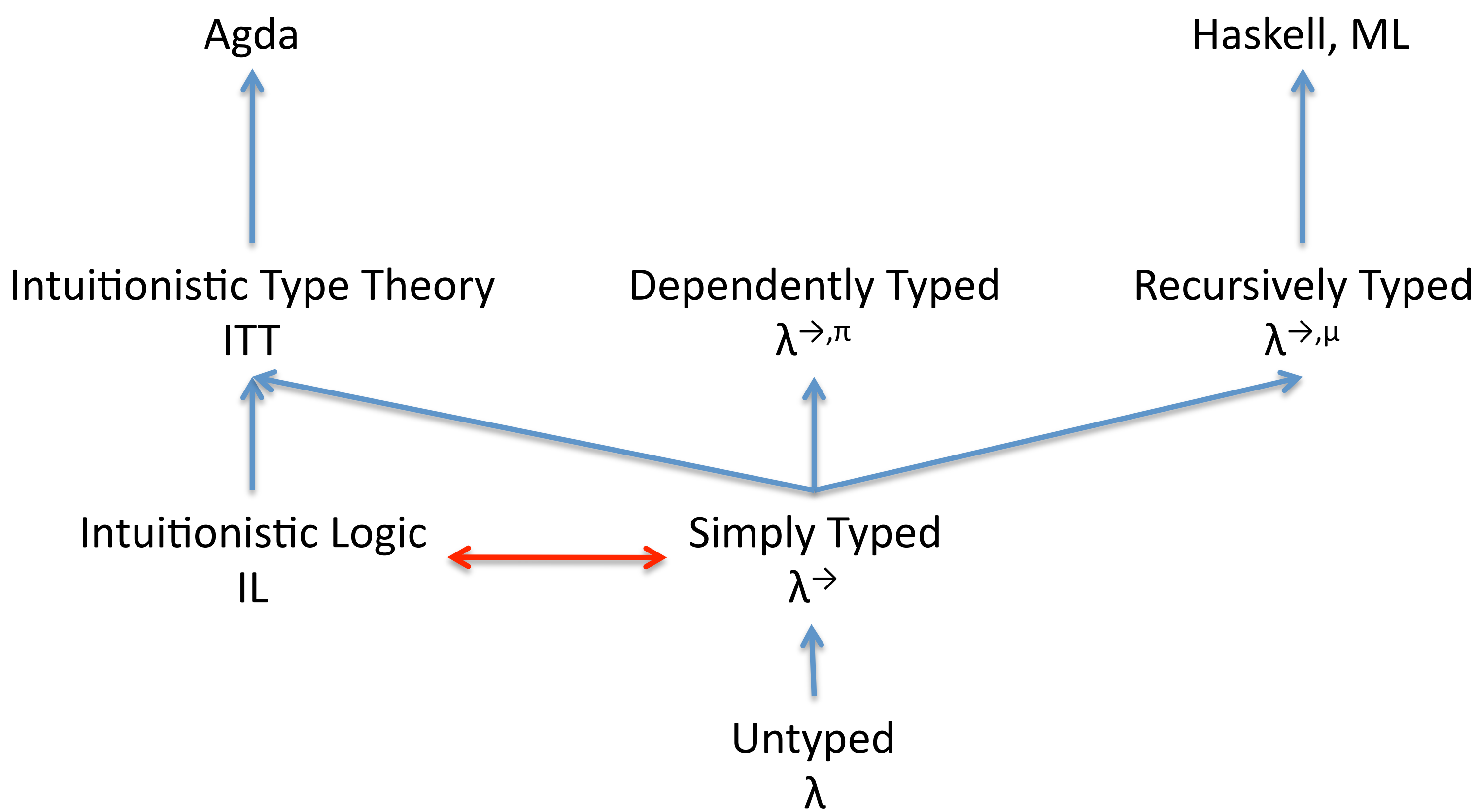
Outline

- Example: implementing queues
- Linear session types
  - A Curry-Howard correspondence
- Linear, affine, and shared channels
  - Substructural adjoint logic
- Synchronous & asynchronous communication
  - Polarization
- Synthesis in polarized adjoint logic
- Conclusion

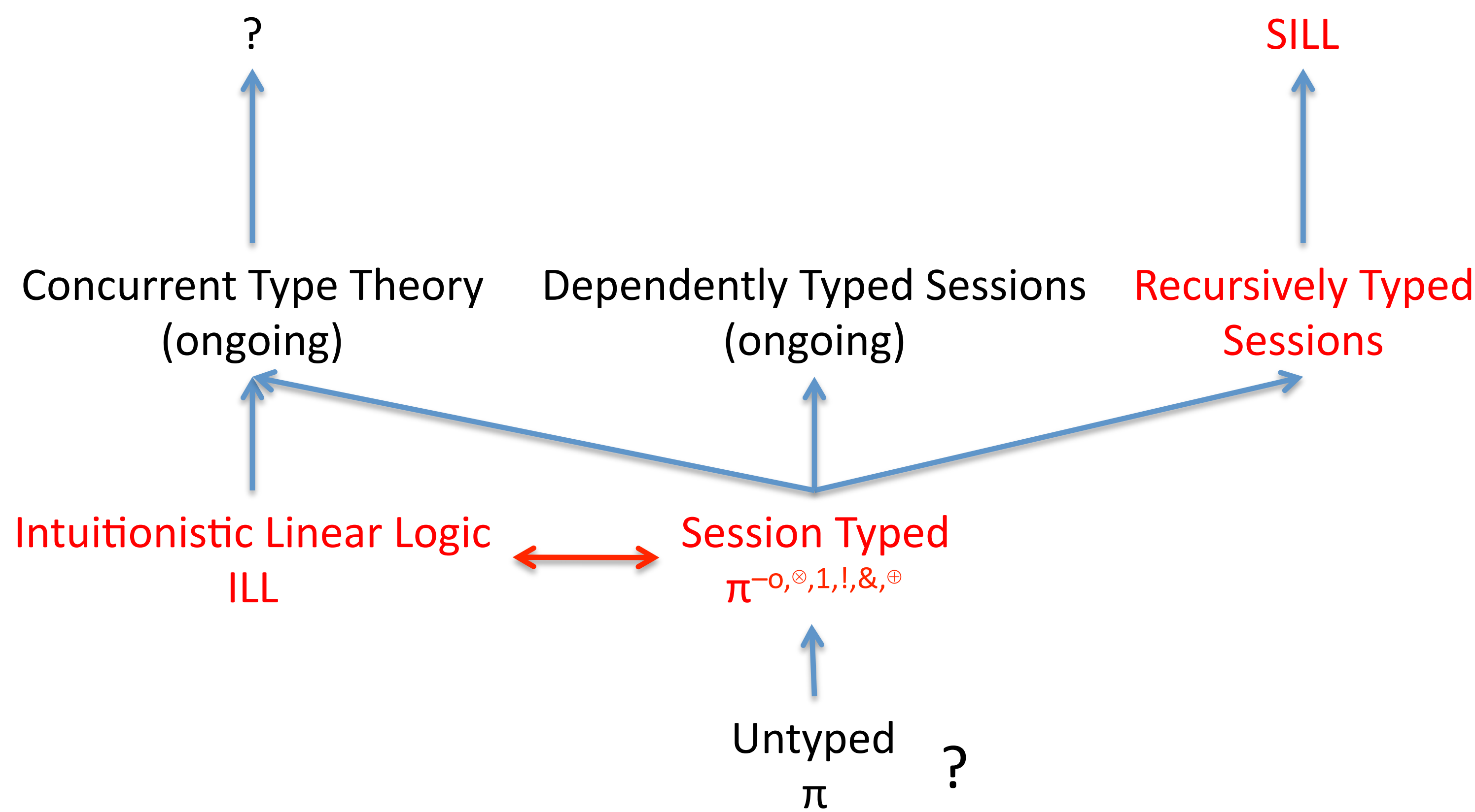
Limitations

- Linear channels with only two endpoints
  - Derives from linear cut and identity
- Shared channels have no shared state
  - Derives from copying semantics of A_U (~ !A)
- Restricted mobility for distributed case
- Challenges
  - Think parallel
  - What can we do without stateful sharing?
  - How can we integrate stateful sharing?

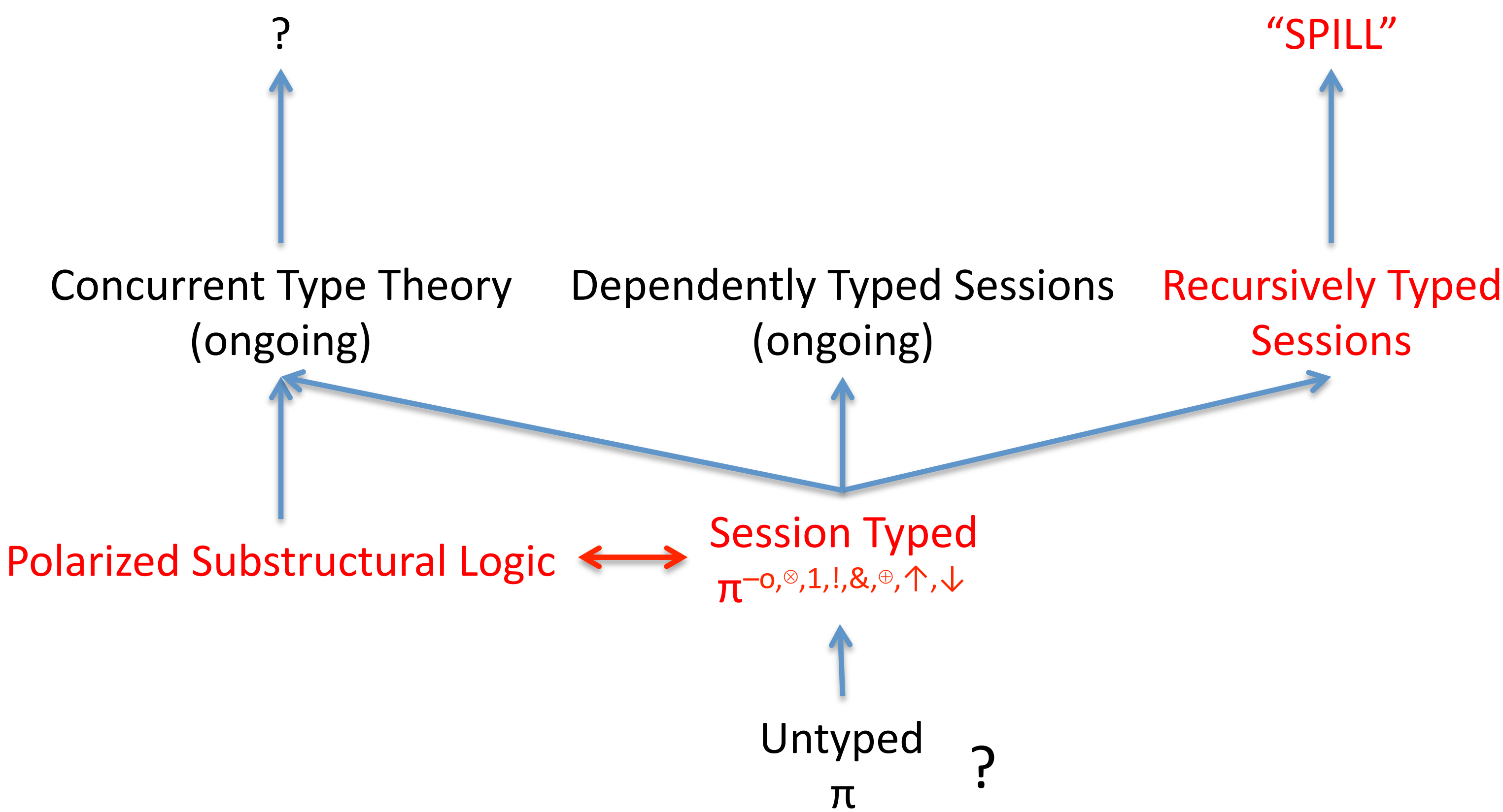
Foundations: Functions



Foundations: Processes



Foundations: Polarized Processes



Summary

- Linear session types &, –o, ⊕, ⊗, 1, (∀ , ∃ , μ)
  - Isomorphic to intuitionistic linear logic (MALL fragm.)
- Affine and unrestricted session types $\uparrow_{k}^{m}$, $\downarrow_{k}^{m}$
  - Modes m,k ::= U | F | L
  - Adjoint logic [Benton'94] [Reed'09]
- Directionality of communication ↑, ↓
  - Polarized linear logic [Andreoli'92] [Laurent'99]
  - Capture synchronization logically
- Synthesis: polarized substructural session types
  - Rules for mode and polarity shifts are identical!
- Paper with more detail in proceedings

Ongoing Work

- Dependent session types
- Dynamic checking of session types, contracts
- Integration with other paradigms
  - Functional, via contextual monad (SILL/SPILL)
  - Imperative (shared memory implementation)
  - Object-oriented (objects-as-processes)
- O'Caml prototype
  - `git clone https://github.com/ISANobody/sill.git`
  - `opam install sill`

Collaborators

- Luís Caires, Bernardo Toninho (Universidade Nova de Lisboa)
- Jorge Peréz (Groningen)
- Dennis Griffith, Elsa Gunter (UIUC)
- Anna Gommerstadt, Limin Jia (CMU) [Dyn. Monitors]
- Stephanie Balzer (CMU) [New foundation for OO]
- Rokhini Prabhu, Max Willsey, Josh Acay [Concurrent C0]
- Henry DeYoung (CMU) [From global to local types]
- Apologies for the lack of references to other related work

Thank you!