

# Session-Typed Concurrent Programming

Frank Pfenning

Carnegie Mellon University

Joint work with Luís Caires, Bernardo Toninho, Jorge Pérez, Dennis Griffith, Elsa Gunter, et al.

# Outline

- A new foundation for session types
- SILL by example
  - Prime sieve
  - List segments
- Language highlights
  - Types and programs
  - Implementation
  - Ongoing research

# Programming Language Design

- Designing effective programming abstractions for a class of computational models
- Measures of success
  - Easy to write correct programs
  - Hard to write incorrect programs
  - Simple, clean, uniform, elegant
  - Easy to reason about, operationally & logically
  - Translates well to lower-level abstractions
  - Interacts smoothly with other constructs

# Message-Passing Concurrency

- Distributed computation is ubiquitous
- Today's libraries too low-level
  - Tacked on to imperative/oo/functional languages
  - Do not yet provide the right abstractions
  - Hard to write correct programs
  - Difficult to reason about
- **New approach: start from logic!**

# Curry-Howard Isomorphisms

- Logical origins of computational phenomena
- Intuitionistic logic  $\Leftrightarrow$  functional programming
- S4 modal logic  $\Leftrightarrow$  quote/eval staging
- S5 modal logic  $\Leftrightarrow$  distributed programming
- Temporal logic  $\Leftrightarrow$  partial evaluation
- **Linear logic  $\Leftrightarrow$  session-typed concurrency**
- More than an analogy!

# Session Types

- Prescribe communication behavior between message-passing concurrent processes
- May be synchronous or asynchronous
- Linear channels with two endpoints
- Shared channels with multiple endpoints
- Messages exchanged can be
  - data values (including process expressions)
  - channels (as in the  $\pi$ -calculus)
  - labels (to indicate choice)

# Linear Logic: A New Foundation

- Linear propositions  $\Leftrightarrow$  session types
- Sequent proofs  $\Leftrightarrow$  process expressions
- Cut  $\Leftrightarrow$  process composition
- Identity  $\Leftrightarrow$  message forwarding
- Proof reduction  $\Leftrightarrow$  communication
- Linear type theory generalizes linear logic
  - Logic: propositions do not mention proofs
  - Type theory: proofs are internalized as terms

# Benefits of Curry-Howard Design

- Integrated development of programming constructs and reasoning principles
  - Correct programs via simple reasoning principles
  - Even if they are not formalized in the language!
- Elegant and expressive language primitives
- Orthogonality and compatibility of constructs
- Programming language theory as proof theory



# Some Choices for SILL

- SILL = Sessions in Intuitionistic Linear Logic
- Conservatively extend functional language
  - Process expressions form a (contextual) monad
  - Communication may be observable
- Manifest notion of process
  - Offer vs. use of a service
  - Process  $\leftrightarrow$  channel along which service is offered

# Properties of SILL

- Type preservation
  - Entails session fidelity on processes
- Progress
  - Absence of deadlock
  - Absence of race conditions on linear channels
- Termination and productivity
  - Some restrictions on recursive types required
- Obeys a general theory of logical relations!

# SILL by Example

- Syntax close to implementation in O'Cam1
- No inference rules, just intuition
- Examples
  - Endless streams of integers
  - Streams of integers
  - Stream filter
  - Prime sieve
  - Linked lists
  - List segments

# Stream of Numbers

- Data types

$\tau ::= \text{bool} \mid \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \dots \mid \{ A \}$

- $\{ A \}$  is type of process offering service  $A$

- Session types

$A ::= \dots$

- Data and session types may be recursive
- In type theory, should be inductive or coinductive (ongoing work)

# Endless Streams of Integers

```
ints = int ^ ints;  
  
from : int → {ints};  
c ← from n =  
  send c n ;  
  c ← from (n+1)
```

- $c : \tau \wedge A$  send value  $v : \tau$  along  $c$  and behave as  $A$
- Non-dependent version of  $\exists x : \tau. A$
- Tail call represents process continuation
- A single process will send stream of integers
- Channel variables and session types in red

# Streams of Integers

```
ints = &{next:int ^ ints, stop:1};  
  
from : int → {ints};  
c ← from n =  
  case (recv c)  
  | next ⇒ send c n ;  
           c ← from (n+1)  
  | stop ⇒ close c
```

- $c : \{l_i : A_i\}_i$  receive label  $l_i$  along  $c$  and continue as  $A_i$
- Labeled n-ary version of linear logic A & B
- External (client's) choice
- $c : 1$  terminate process; as linear logic 1
- Closing a channel  $c$  terminates offering process

# Filtering a Stream

```
ints = &{next:int ^ ints, stop:1};
filter : (int → bool) → {ints ← ints};
filterNext : (int → bool) → {int ^ ints ← ints};

c ← filter q ← d =
  case (recv c)
  | next ⇒ c ← filterNext q ← d
  | stop ⇒ send d stop ;
           wait d ;
           close c
```

- $\{A \leftarrow A_1, \dots, A_n\}$  process offering  $A$ , using  $A_i$ 's
- Type of channels changes based on process state!
- Type error, say, if we forget to stop  $d$

# Finding the Next Element

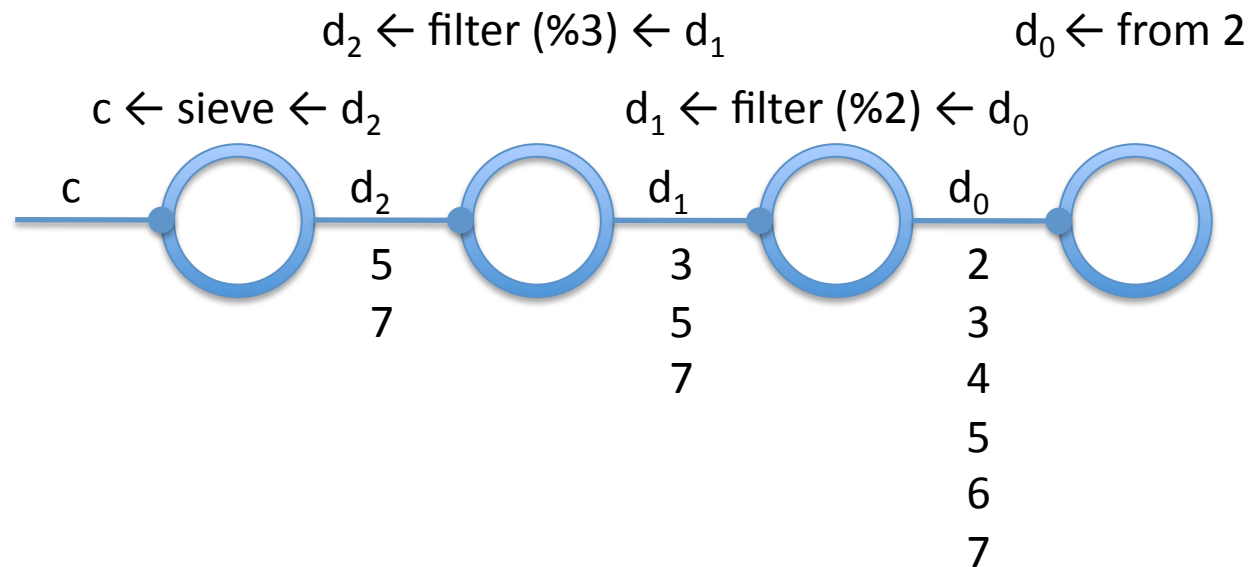
```
ints = &{next:int ^ ints, stop:1};
filter : (int → bool) → {ints ← ints};
filterNext : (int → bool) → {int ^ ints ← ints};

c ← filterNext q ← d =
  send d next ;
  n ← recv d ;
  case (q n)
  | true ⇒ send c n ;
           c ← filter q ← d
  | false ⇒ c ← filterNext q ← d
```

- filter/filterNext process identified with channel **c**

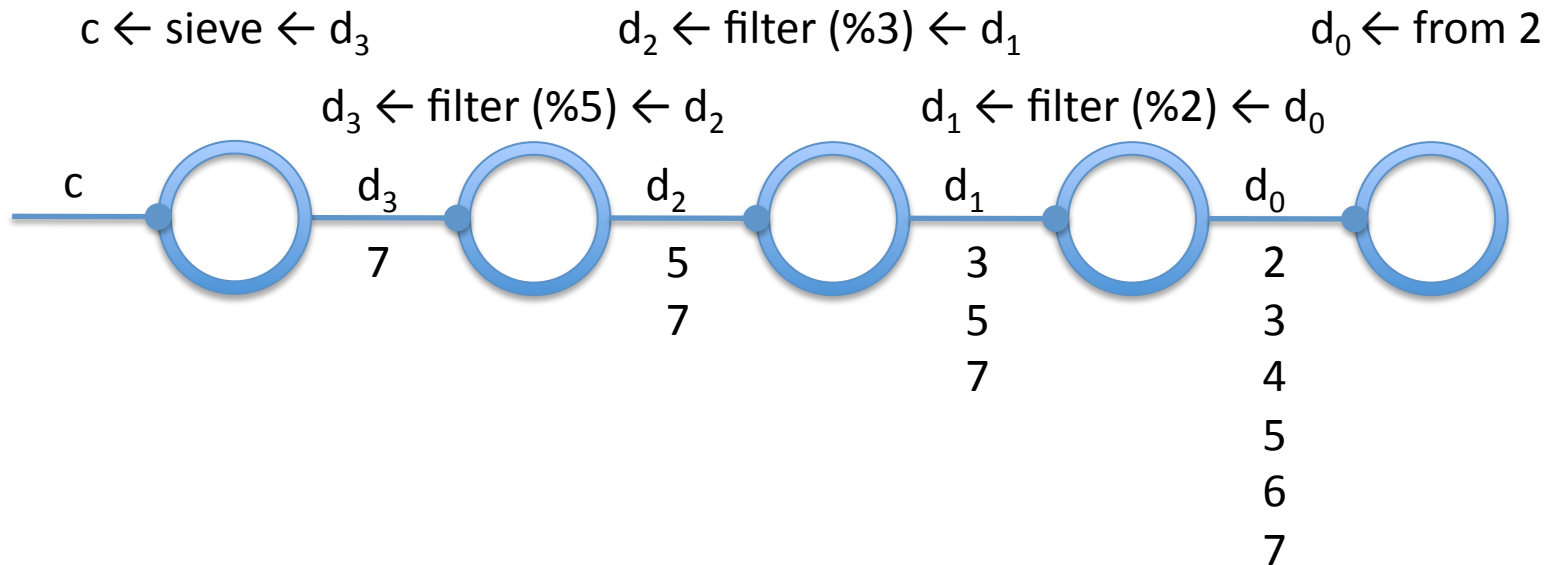


# Prime Sieve



- $c \leftarrow \text{sieve} \leftarrow d$  sends first value  $p$  on  $d$  along  $c$
- Then spawns new process to filter out  $\%p$

# Prime Sieve



- $c \leftarrow \text{sieve} \leftarrow d$  sends first value  $p$  on  $d$  along  $c$
- Then spawns new process to filter out  $\%p$

# Prime Sieve

```
ints = &{next:int ^ ints, stop:1};
sieve : {ints ← ints};

c ← sieve ← d =
  case (recv c)
  | next ⇒ send d next ;
           p ← recv d ;
           send c p ;
           e ← filter (mod p) ← d ;
           c ← sieve ← e
  | stop ⇒ send d stop ; wait d ; close c
```

- $e \leftarrow \text{filter}(\text{mod } p) \leftarrow d$  spawns new process
- Uses  $d$ , offers  $e$  (which is used by sieve)

# Primes

```
ints = &{next:int ^ ints, stop:1};  
primes : {ints};  
  
c ← primes =  
  d ← from 2 ;  
  c ← sieve ← d
```

- Primes correct with sync or async communication
- $n+2$  processes for  $n$  primes

# List Segments

- Segment is linked list, abstracted over its tail
- Have functional and imperative versions
- Concurrent version
  - No particular implementation prescribed
  - Elements are channels to access data

# Linked Lists

```
list =  $\oplus$ {nil:l, cons:A  $\otimes$  list};
```

```
nil : {list};
```

```
c  $\leftarrow$  nil =  
  send c nil ;  
  close c
```

```
cons : {list  $\leftarrow$  A, list}
```

```
c  $\leftarrow$  cons  $\leftarrow$  x, d =  
  send c cons ;  
  send c x ;  
  c  $\leftarrow$  d
```

- $c : \oplus\{l_i : A_i\}_i$  send label  $l_i$  along  $c$  and continue as  $A_i$
- Labeled n-ary version of linear logic  $A \oplus B$
- Internal (provider's) choice
- $c : A \otimes B$  output a channel of type  $A$ , continue as  $B$

# List Segments

```
seg = list -o list;
```

```
empty : {seg};
```

```
c ← empty =  
  t ← recv c ;  
  c ← t
```

```
tolist : {list ← seg}
```

```
c ← tolist ← d =  
  n ← nil ;  
  send d n ;  
  c ← d
```

- $c : A -o B$  input a channel of type  $A$ , continue as  $B$
- Can add to segment at both ends, concatenate segs
- Can convert to list

# Concatenating List Segments

```
seg = list -o list;  
  
concat : {seg ← seg, seg}  
c ← concat ← d, e =  
  t ← recv c ;  
  send e t ;  
  send d e ;  
  c ← d
```

- Implements more general composition



# Prepending, Appending Elements

```
seg = list -o list;
```

```
prepend : {seg ← A, seg}    append : {seg ← seg, A}  
c ← prepend ← x, d =      c ← append ← d, x =  
t ← recv c ;              t ← recv c ;  
send d t ;                e ← cons ← x, t ;  
e ← cons ← x, d ;         send d e ;  
c ← e                     c ← d
```

# Stateful Shared Processes

- Shared channels (type **!A**) do not share state
  - Cannot use list segments for usual imperative implementation of queues
  - Use functional technique in process setting
- Challenge: can we recover mutable sharing?
  - Type preservation (session fidelity)
  - Progress (freedom from deadlock)
  - Ongoing

# Other Examples

- Data structures
  - Stacks, queues, binary search trees
  - Syntax trees, evaluation, tree transformation
- Algorithms
  - Lazy and eager prime sieve
  - Merge sort, odd/even sort, insertion sort
- Protocols
  - Needham/Schroeder, safe and unsafe

# Outline

- A new foundation for session types
- SILL by example
  - Prime sieve
  - List segments
- **Language highlights**
  - Types and programs
  - Implementation
  - Ongoing research

# Session Type Summary

- From the point of view of session provider

$c : \tau \wedge A$	send value $v : \tau$ along $c$ , continue as $A$
$c : \tau \rightarrow A$	receive value $v : \tau$ along $c$ , continue as $A$
$c : A \otimes B$	send channel $d : A$ along $c$ , continue as $B$
$c : A \multimap B$	receive channel $d : A$ along $c$ , continue as $B$
$c : 1$	close channel $c$ and terminate
$c : \oplus\{l_i : A_i\}$	send label $l_i$ along $c$ , continue as $A_i$
$c : \&\{l_i : A_i\}$	receive label $l_i$ along $c$ , continue as $A_i$
$c : !A$	send persistent $!u : A$ along $c$ and terminate
$!u : A$	receive $c : A$ along $!u$ for fresh instance of $A$

# Contextual Monad

- $M : \{ A \leftarrow A_1, \dots, A_n \}$  process expressions offering service  $A$ , using services  $A_1, \dots, A_n$
- Composition  $c \leftarrow M \leftarrow d_1, \dots, d_n ; P$ 
  - $c$  fresh, used (linearly) in  $P$ , consuming  $d_1, \dots, d_n$
- Identity  $c \leftarrow d$ 
  - Notify client of  $c$  to talk to  $d$  instead and terminate
- Strong notion of process identity

# Static Type Checking

- Bidirectional
  - Precise location of type errors
  - Based on definition of normal proofs in logic
  - Fully compatible with linearity
- Natural notion of behavioral subtyping, e.g.
  - $\&\{l:A, k:B\} \leq \&\{l:A\}$  (we can offer unused alt's)
  - $\oplus\{l:A\} \leq \oplus\{l:A, k:B\}$  (we need not produce all alt's)
- Supports ML-style value polymorphism
- Explicit behavioral polymorphism for sessions

# Dynamic Semantics

- Three back ends
  - Synchronous threads
  - Asynchronous threads
  - Distributed processes
- Fourth back end (hypothetical):
  - Solos ?
- Curry-Howard lesson:
  - The syntax can remain stable (proofs!)
  - The semantics can vary: controlling reductions
  - Must be **consistent** with proof theory
- Not released (but multiple “friendly” users)



# Dynamic Type Checking

- May not trust all participating processes
- Type system compatible with
  - Value dependent types, e.g.  $\text{nat} = \{x:\text{int} \mid x \geq 0\}$
  - Full dependent types (ongoing):
    - “Right” equivalence on process expressions
    - Restrictions on recursive types
- Contracts are partial identity processes
  - Blame assignment (ongoing)
  - Causality (ongoing)

# Some Refinements

```
nat = {x:int | x ≥ 0};
nats = &{next:nat ∧ nats, stop:1};

eq n = {x:int | x = n};
succs n = &{next:eq n ∧ succs(n+1), stop:1};

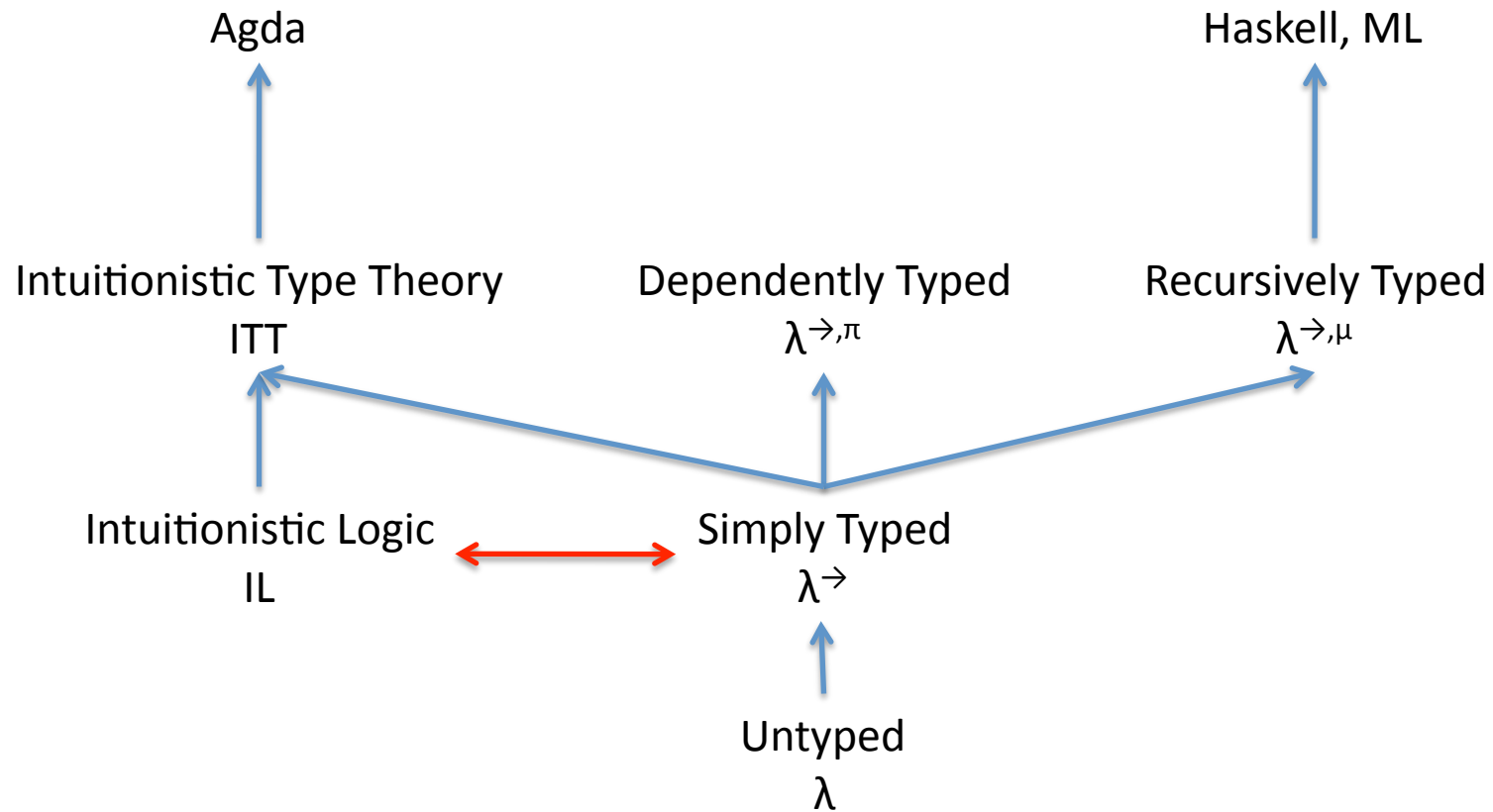
gt n = {x:int | x > n};
incrs n = &{next:∃k:gt n. incrs k, stop:1};
```

- eq and gt are value type families
- succs and incrs are session type families
- Last line illustrates  $\exists$  as dependent  $\wedge$
- Not yet implemented

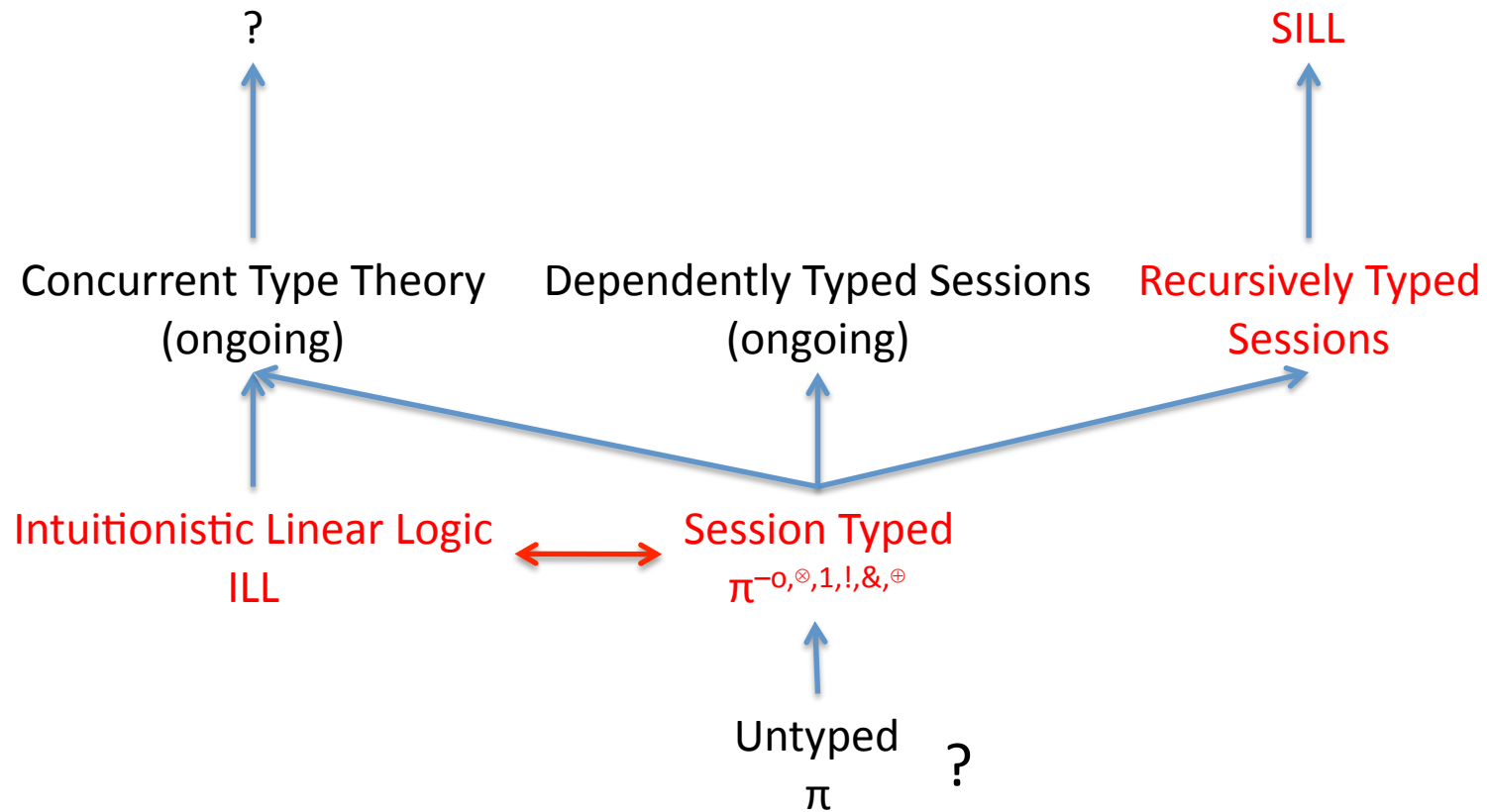
# Other Logical Thoughts

- Affine logic (= linear logic + weakening)
  - Static deallocations inserted
  - Shorter programs, but errors more likely
- Hybrid linear logic (= linear logic + worlds)
  - Worlds representing security domains
  - Accessibility relation between domains
  - Ongoing
- Affirmation modality for digital signatures

# Foundations: Functions



# Foundations: Processes



# $\pi$ -Calculus as Computational Model

- Not as robust or canonical as  $\lambda$ -calculus
- A session-typed, linear logic perspective
  - Replication (!) is too general ( $!P \equiv !P \mid P$ )
  - Lacks forwarding primitive
  - Lacks labeled choice
  - Input always blocking
- Alternative: substructural operational semantics
  - Forward chaining linear logic programming
  - Towards a synthesis of paradigms? (ongoing)

# Curry-Howard: How Far to Go?

- Computation vs. proof reduction
  - Computation imposes a strategy
  - Proof reduction could be anywhere
  - $\eta$ -expansion as equality, not computation
- Functional programming
  - Always stop at  $\lambda$ -abstraction (negative type)
  - Call-by-name vs. call-by-value vs. call-by-need vs...

# Curry-Howard: How Far to Go?

- Option 1: Synchronous  $\pi$ -calculus
  - Only judgmental rules (cut, id) commute
  - No propositional rules commute
- Option 2: Asynchronous  $\pi$ -calculus
  - Commute past outputs (pos. multiplicatives)
  - Don't commute past inputs (as in functional progs)
- Option 3: Solos
  - Commute past inputs (neg. multiplicatives)
  - Do not commute past neg. additives, exponentials



# Summary

- SILL, a functional language with a contextual monad for session-typed message-passing concurrency
  - Type preservation (session fidelity)
  - Progress (deadlock and race freedom)
  - Implementation with subtyping, polymorphism, recursive types
- Based on a Curry-Howard interpretation of intuitionistic linear logic
- Full dependent type theory in progress
- Dynamic check of types and contracts in progress

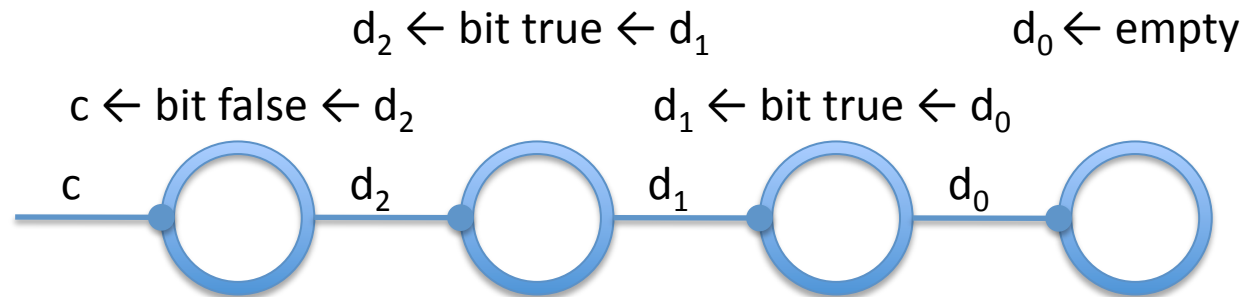
# Some References

- 2010
  - CONCUR: the basic idea, revised for MSCS, 2012
- 2011
  - PPDP: dependent types
  - CPP: digital signatures ( $\diamond A$ )
- 2012
  - CSL: asynchronous comm.
  - ESOP: logical relations
  - FOSSACS: functions as processes
- 2013
  - ESOP: behavioral polymorphism
  - ESOP: monadic integration (SILL)
- 2014 (in progress)
  - Security domains ( $A @ w$ ), spatial distribution
    - J. Pérez, 14:30 today!
  - Coinductive types
  - Blame assignment

# Thanks!

- Luís Caires, Bernardo Toninho, Jorge Pérez (Universidade Nova de Lisboa)
  - FCT and CMU|Portugal collaboration
- Dennis Griffith, Elsa Gunter (UIUC) [Implementation]
  - NSA
- Michael Arntzenius, Limin Jia (CMU) [Blame]
- Stephanie Balzer (CMU) [New foundation for OO]
- Henry DeYoung (CMU) [From global specs to local types]
- Much more to say; see <http://www.cs.cmu.edu/~fp>
- Apologies for the lack of references to related work

# Bit Strings



```
bits = ⊕{eps:1, bit:bool ∧ bits};
```

- Lowest bit on the left (above represents 6)
- $c : \oplus\{l_i:A_i\}_i$  send a label  $l_i$  along  $c$  and cont. as  $A_i$
- n-ary version of linear logic  $A \oplus B$
- Internal (provider's) choice

# Bit String Constructors

```
bits =  $\oplus$ {eps:1, bit:bool  $\wedge$  bits};  
  
empty : {bits};  
c  $\leftarrow$  empty =  
  send c eps ;  
  close c  
  
bit : bool  $\rightarrow$  {bits  $\leftarrow$  bits};  
c  $\leftarrow$  bit b  $\leftarrow$  d =  
  send c bit ;  
  send c b ;  
  c  $\leftarrow$  d;
```

- Forwarding  $c \leftarrow d$  represents logical identity
  - Process offering along  $c$  terminates
  - Client subsequently talks to process offering along  $d$

# Alternative Constructor

```
bits =  $\oplus$ {eps:1, bit:bool  $\wedge$  bits};  
  
num : int  $\rightarrow$  {bits};  
c  $\leftarrow$  num n =  
case n == 0  
| true  $\Rightarrow$  send c eps ; close c  
| false  $\Rightarrow$  send c bit ;  
             send c (odd n) ;  
             c  $\leftarrow$  num (n/2)
```

- num as a single process holding an int n
- Channel type is process interface, not representation

# Increment

```
bits = ⊕{eps:1, bit:bool ^ bits};
inc : {bits ← bits};

c ← inc ← d =
  case (recv d)
  | eps ⇒ wait d ;
          e ← eps ;
          c ← bit true ← e
  | bit ⇒ b ← recv d ;
          case b
          | true ⇒ e ← inc ← d ;
                  c ← bit false ← e
          | false ⇒ c ← bit true ← d
```

- inc process generates one bit string from another
- Spawns a new inc process in case of a carry

# Addition

```
bits =  $\oplus$ {eps:1, bit:bool  $\wedge$  bits};
add : {bits  $\leftarrow$  bits, bits};
c  $\leftarrow$  add  $\leftarrow$  d, e =
  case (recv d)
  | eps  $\Rightarrow$  wait d ;
                  c  $\leftarrow$  e
  | bit  $\Rightarrow$  b1  $\leftarrow$  recv d ;
                case (recv e)
                | eps  $\Rightarrow$  wait e ;
                      send c bit;
                      send c b1;
                      c  $\leftarrow$  d
                | bit  $\Rightarrow$  b2  $\leftarrow$  recv e ; ...
```

- add uses two channels, provides one
- Receives are sequential; additional parallelism could be justified by commuting conversions in proof theory



# Odd/Even Sort

```
cell = ⊕{someR:int ∧ cell', tail:cell};
cell' = &{someL:int → cell, head:cell};
elem : side → int → int → {cell ← cell};
c ← elem _ 0      n ← d = ... (sorted)
c ← elem L (i+1) m ← d =
  case (recv d)
  | someR ⇒ k ← recv d ;
            send d someL ; send d m ;
            case m > k
            | true  ⇒ c ← elem R i k ← d
            | false ⇒ c ← elem R i m ← d
  | tail ⇒ c ← elem R i m ← d
c ← elem R (i+1) k ← d =
  send c someR ; send c k ;
  case (recv c)
  | someL ⇒ m ← recv c ;
            case m > k
            | true  ⇒ c ← elem L i m ← d
            | false ⇒ c ← elem L i k ← d
  | head ⇒ c ← elem L i k ← d
```