From Linear Logic to Session-Typed Concurrent Programming

Frank Pfenning

Joint work with Luís Caires, Bernardo Toninho, Dennis Griffith

Outline

- The Curry-Howard correspondence
- A change of perspective: linear logic
- Constructing the language SILL
  - Propositions as types
  - Proofs as processes
  - Cut reduction as communication
- Examples
- O'Caml source

```
git clone https://github.com/ISANobody/sill.git
```

Language Design from Logic

- Integrate computation and reasoning
- Logic programming (not this talk)
  - Theories as programs
  - Proof construction as computation
- “Functional” programming (this talk)
  - Propositions as types
  - Proofs as programs
  - Proof reduction as computation

Functional Programming

- Curry [1934]
  - Propositions as simple types
  - Intuitionistic Hilbert proofs as combinator terms
  - Computation is combinator reduction
- Howard [1969]
  - Propositions as simple types
  - Intuitionistic natural deductions as programs
  - Proof reduction as computation

Deceptive Simplicity

- Generalizes to intuitionistic type theory
  - General recipe beyond the propositional fragment
- Matching computational phenomena and logic
  - Insufficient to just define proof terms
- Many properties come “for free”
  - Type preservation
  - Progress
  - Termination (on pure fragment)

Examples

- Modal logic S4 and staged computation
- Temporal logic and partial evaluation
- Lax logic and computational effects
- Modal logic T and proof irrelevance
- Modal logic S5 and distributed computation
- Classical logic and continuations
- Today: linear logic
  - Let's see what happens!

Intuitionistic Logic and Functions

- Basic natural deduction judgment

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

  – From hypotheses A_1, …, A_n derive conclusion A
- With proof terms:

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : A$$

  – Labeled hyps / variables x_i of type A_i
  – Proof / program M of type A
  – When given $N_i{:}A_i$, $[N_i/x_i]M \Rightarrow^* V$ (a value)

(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
- In full language:
  - Affine resources: use at most once
  - Unrestricted hypotheses: use arbitrarily often
- Classical linear logic also possible [Wadler 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 can 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
- Operational semantics
  - $\mathsf{proc}_c(P)$: process P provides along channel c
  - State is multiset of executing processes

$$\mathsf{proc}_c(a \leftarrow P_a \; ; Q_a) \Longrightarrow \mathsf{proc}_b(P_b), \mathsf{proc}_c(Q_b) \quad (b \text{ fresh})$$

- In π-calculus: $(a \leftarrow P_a \; ; Q_a) \equiv (\nu a)(P_a \mid Q_a)$

Identity as Process Forwarding

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

- Operationally

$$\mathsf{proc}_c(c \leftarrow a) \Longrightarrow c = a$$

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

Existential Quantification

- Connectives have right and left rules
- Right rule: how do we prove ∃x. A?
- Left rule: how do we use ∃x. A?
- The existential quantifier

$$\frac{\Delta \vdash [t/x]A}{\Delta \vdash \exists x.\, A}\ \exists R \qquad \frac{\Delta, [y/x]A \vdash C}{\Delta, \exists x.\, A \vdash C}\ \exists L^y$$

  – y is fresh in premise of left rule

Right Rule Sends the Witness

- Right rule contains witness t

$$\frac{\Delta \vdash Q :: (c : [t/x]A)}{\Delta \vdash (\mathsf{send}\ c\ t\ ;\ Q) :: (c : \exists x.\, A)}\ \exists R$$

- Send the witness along channel c
- Continuation Q will provide [t/x]A along c
- Left rule will have a matching action
- Channel c "changes type"

Left Rules Receives the Witness

- Parameter y stands for received witness

$$\frac{\Delta, c{:}[y/x]A \vdash P :: (d : C)}{\Delta, c{:}\exists x.\, A \vdash (y \leftarrow \mathsf{recv}\ c\ ;\ P) :: (d : C)}\ \exists L^y$$

- Operational semantics communicates value

$$\begin{array}{l} \mathsf{proc}_c(\mathsf{send}\ c\ t\ ;\ Q), \mathsf{proc}_d(y \leftarrow \mathsf{recv}\ c\ ;\ P) \\ \Longrightarrow \\ \mathsf{proc}_c(Q), \mathsf{proc}_d([t/y]P) \end{array}$$

The Pattern of Right and Left Rules

- Each connective will be defined by right and left rules (sequent calculus)
- Right rules define how to prove A
  - For the process, how to provide A
- Left rules define how to exploit A
  - For the process, how to use A
- Matching complementary process actions
  - The one with information sends (non-invertible)
  - The one without information receives (invertible)

Cut Reduction as Communication

- Logical cut reduction is a communication

$$\dfrac{\dfrac{\Delta \vdash [t/x]A}{\Delta \vdash \exists x.\, A}\ \exists R \qquad \dfrac{\Delta', [y/x]A \vdash C}{\Delta', \exists x.\, A \vdash C}\ \exists L^y}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_{\exists x.\, A}$$

$$\Longrightarrow \qquad \dfrac{\Delta \vdash [t/x]A \qquad \Delta', [t/x]A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_{[t/x]A}$$

- Term t from first premise is substituted in second premise

Recursive Types

- Let's accept recursively defined propositions
  - Formal treatment as (co)inductive types
- Classify terms by simple types: ∃ x:τ. A
- Example: $\mathsf{ints} = \exists x{:}\mathsf{int}.\, \mathsf{ints}$
  - Represents an infinite stream of integers
- Abbreviate $\tau \wedge A = \exists x{:}\tau.\, A \qquad (x \text{ not free in } A)$
- Let's also accept recursively defined processes

Endless Streams of Integers

```
ints = int ∧ ints;

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

- {A} is the type of a process P :: (c : A)
- c ← P means process P offers along channel c
- Tail call represents process continuation
  - A single process will send stream of integers
- Channel variables and session types in red

External Choice

- Client chooses between provided alternatives
- Provider offers both
- Logically: A & B

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

- Duplication of Δ okay in &R, since only one of A and B will be used

Communicating a Choice

- Provider offers choice (receive label inl or inr)

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

- Client makes choice (send label inl or inr)

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

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

Communicating Choice Labels

- Communication

$$\mathsf{proc}_c(\mathsf{case}(c, \mathsf{inl}.P, \mathsf{inr}.Q)), \mathsf{proc}_d(c.\mathsf{inl}\ ;\ R) \Longrightarrow \mathsf{proc}_c(P), \mathsf{proc}_d(R)$$

$$\mathsf{proc}_c(\mathsf{case}(c, \mathsf{inl}.P, \mathsf{inr}.Q)), \mathsf{proc}_d(c.\mathsf{inr}\ ;\ R) \Longrightarrow \mathsf{proc}_c(Q), \mathsf{proc}_d(R)$$

- Can again be derived from cut reduction
- In SILL we use labeled choice &{l_k : A_k}
- A & B = &{inl : A, inr : B}

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

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

- close sends a token, wait receives it

Streams of Integers

```
ints = &{next:int ∧ ints, stop:1};

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

- Provider must always be able to send more
- Client can choose to stop or get next int

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 c
  | next ⇒ c ← filterNext q ← d
  | stop ⇒ d.stop ;
           wait d ;
           close c
```

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

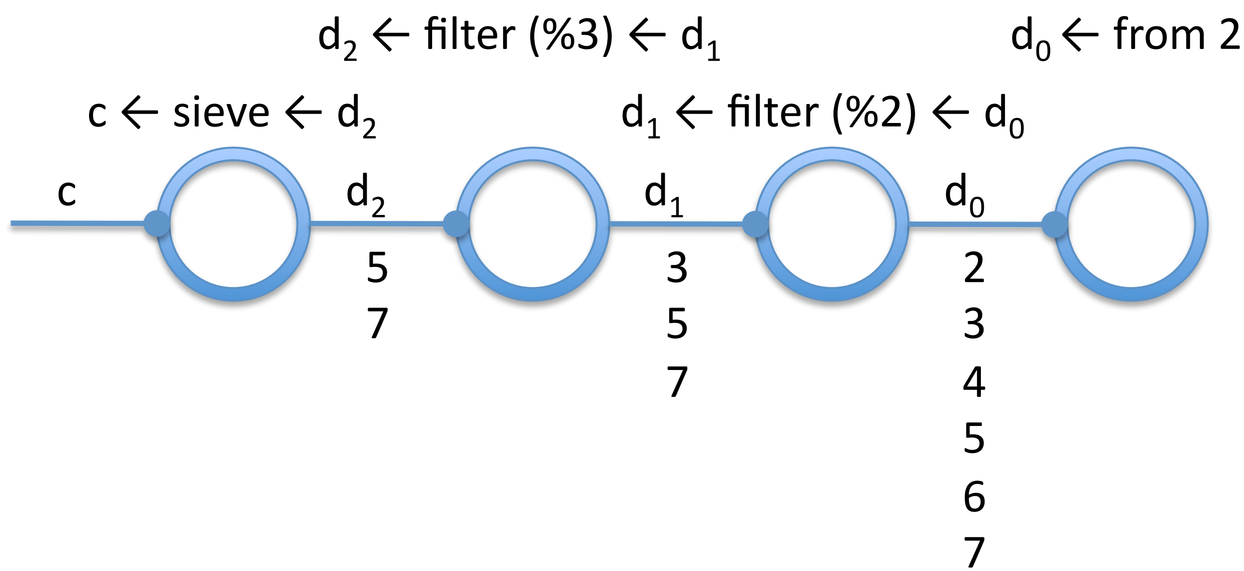
Finding the Next Element

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

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

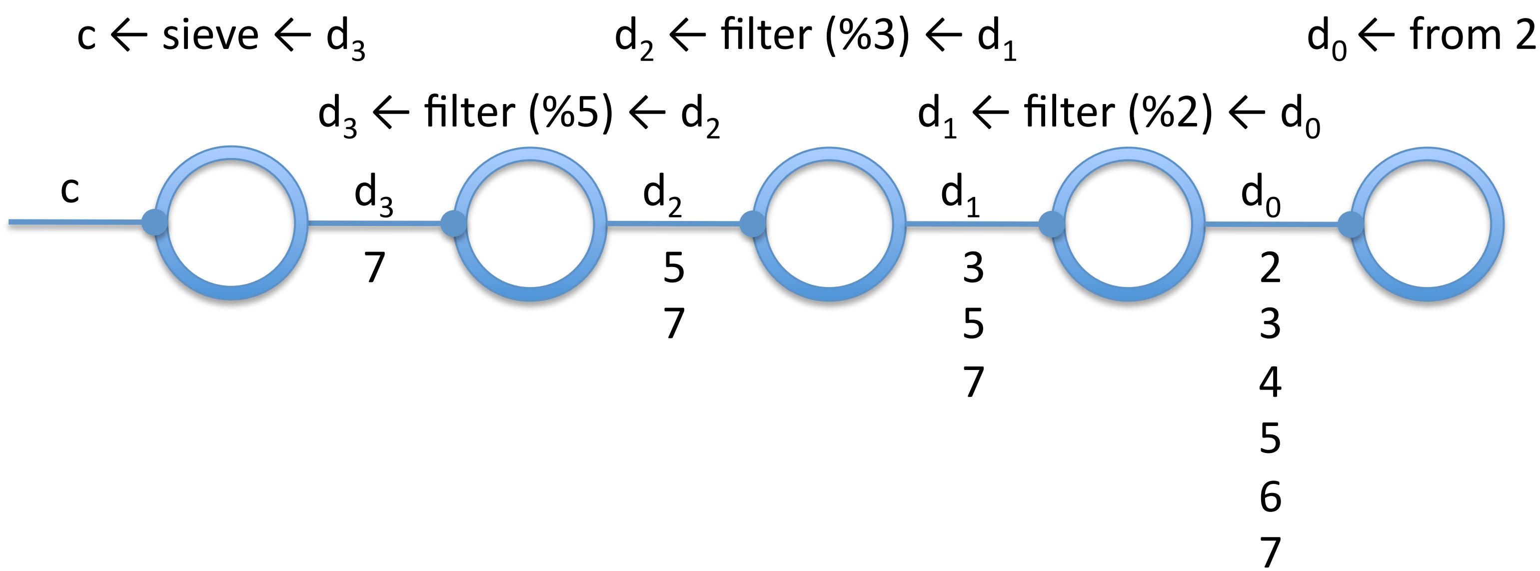
- filter/filterNext process identified with channel c

Prime Sieve



- c ← sieve ← d sends first value p on d along c
- Then spawns new process to filter out %p

Prime Sieve



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

- e ← filter (mod p) ← 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

Internal Choice

- External choice: client chooses
- Internal choice: the provider chooses
- Client has to account for both
- Logically: A ⊕ B

$$\frac{\Delta \vdash A}{\Delta \vdash A \oplus B} \oplus R_1 \quad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B} \oplus R_2 \quad \frac{\Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta, A \oplus B \vdash C} \oplus L$$

Internal Choice, Operationally

- Provider sends label, client branches on it

$$\frac{\Delta \vdash P :: (c : A)}{\Delta \vdash (c.\mathsf{inl}\ ;\ P) :: (c : A \oplus B)} \oplus R_1 \qquad \frac{\Delta \vdash P :: (c : B)}{\Delta \vdash (c.\mathsf{inr}\ ;\ P) :: (c : A \oplus B)} \oplus R_2$$

$$\frac{\Delta, c : A \vdash Q :: (d : C) \quad \Delta, c : B \vdash R :: (d : C)}{\Delta, c : A \oplus B \vdash \mathsf{case}(c, \mathsf{inl}.Q, \mathsf{inr}.R) :: (d : C)} \oplus L$$

- Nothing new in the operational semantics

Lists as Internal Choice

- Replace binary with n-ary labeled choice
  - $A \oplus B = \oplus\{inl: A, inr: B\}$
- Lists of ints
  - $list = \oplus\{nil: 1, cons: int \wedge list\};$
- Lists of channels
  - $list\ A = \oplus\{nil: 1, cons: A \otimes list\ A\};$
- Representation is unspecified!

Combining Resources

- Multiplicative conjunction A $\otimes$ B, logically

$$\frac{\Delta \vdash A \quad \Delta' \vdash B}{\Delta, \Delta' \vdash A \otimes B} \otimes R \qquad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C} \otimes L$$

- Operationally, $\otimes$R sends, $\otimes$L receives
  - $\otimes$R is non-invertible
  - $\otimes$L is invertible (carries no information)
- Designate B as continuation, send channel d:A
- Other choice also logically sound

Sending and Receiving Channels

- Other channels are 'split' between processes

$$\frac{\Delta \vdash P :: (d : A) \quad \Delta' \vdash Q :: (c : B)}{\Delta, \Delta' \vdash (\mathsf{send}\ c\ (d \leftarrow P_d)\ ;\ Q) :: (c : A \otimes B)} \otimes R$$

$$\frac{\Delta, d : A, c : B \vdash R :: (e : C)}{\Delta, c : A \otimes B \vdash (d \leftarrow \mathsf{recv}\ c\ ;\ R_d) :: (e : C)} \otimes L$$

- Operationally

$$\mathsf{proc}_c(\mathsf{send}\ c\ (d \leftarrow P_d)\ ;\ Q), \mathsf{proc}_e(d \leftarrow \mathsf{recv}\ c\ ;\ R_d)$$
$$\Longrightarrow$$
$$\mathsf{proc}_a(P_a), \mathsf{proc}_c(Q), \mathsf{proc}_e(R_a) \qquad (a \text{ fresh})$$

Sending Existing Channels

- Previous construct always sends fresh channel
- Frequently, channel we want to send not fresh
  - Employ forwarding
  - send c d = send c (d’ ← (d’ ← d))
- Derived rule

$$\frac{\Delta \vdash P :: (c : B)}{\Delta, d : A \vdash (\textsf{send}\ c\ d\ ;\ P) :: (c : A \otimes B)}$$

Lists of Channels

```
list A = ⊕{nil:1, cons:A ⊗ list A};

nil : {list A};
c ← nil =
  c.nil ;
  close c

cons : {list A ← A, list A}
c ← cons ← x, d =
  c.cons ;
  send c x ;
  c ← d
```

How to Implement a Queue?

- A header process with references to front and back is impossible
  - Race condition at last node
  - SILL is inherently free of race conditions
- Sharing only with persistent channels (!A)
  - Do not permit “mutation”
- Two stacks (as lists) is possible
- Alternative: exploit concurrency!

Behavioral Abstraction

- Interface to a process specifies interaction behavior, hides implementation
- Implement queue interface with constant time enqueue and dequeue operations
- One process for each element in queue
- Need: A –o B (with resource A, can prove B)
  - Receive a channel of type A
  - Proceed as B

Queues of Channels

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

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

SILL Properties

- Derived from logical origins
- Session fidelity (= type preservation)
- Deadlock freedom (= global progress)
- Absence of race conditions (= confluence)
- Termination & productivity
  - With restrictions on recursive types

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

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
- Restricted mobility for distributed case

Static Type Checking

- Bidirectional
  - Precise location of type errors (once it parses…)
  - Based on definition of normal proofs in logic
  - Fully compatible with linearity
- Natural notion of behavioral subtyping, e.g.
  - &{l:A, k:B} ≤ &{l:A} (we can offer unused alt's)
  - ⊕{l:A} ≤ ⊕{l:A, k:B} (we need not produce all alt's)
- Supports ML-style value and session polymorphism
- Explicit behavioral polymorphism for sessions
- Affine types @A, with distributed garbage collection

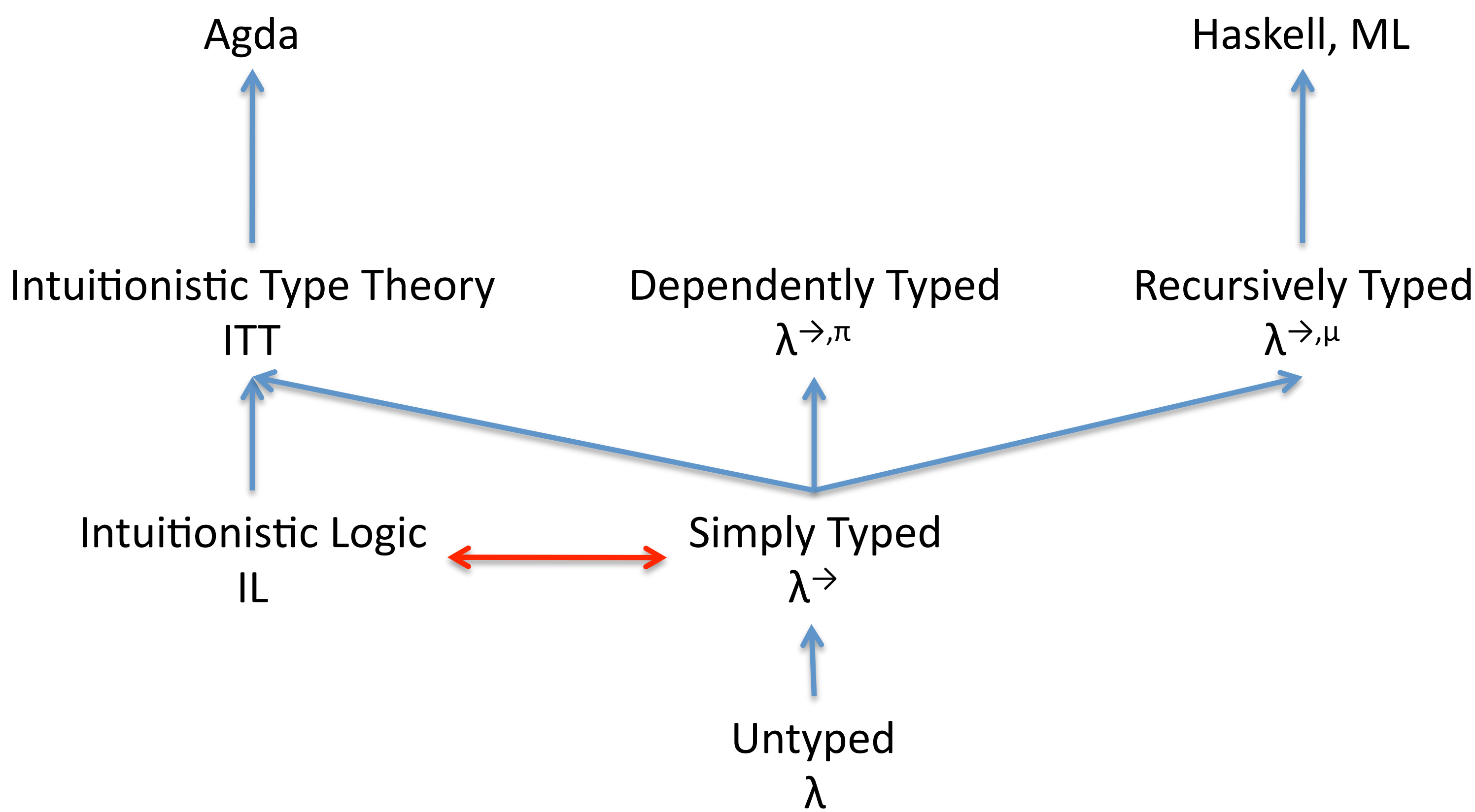
Dynamic Semantics

- Three back ends
  - Synchronous threads
  - Asynchronous threads
  - Distributed processes (incomplete)
- Curry-Howard lesson:
  - The syntax can remain stable (proofs!)
  - The semantics can vary: controling reductions
  - Must be consistent with proof theory
- O'Caml implementation at
  - `git clone https://github.com/ISANobody/sill.git`

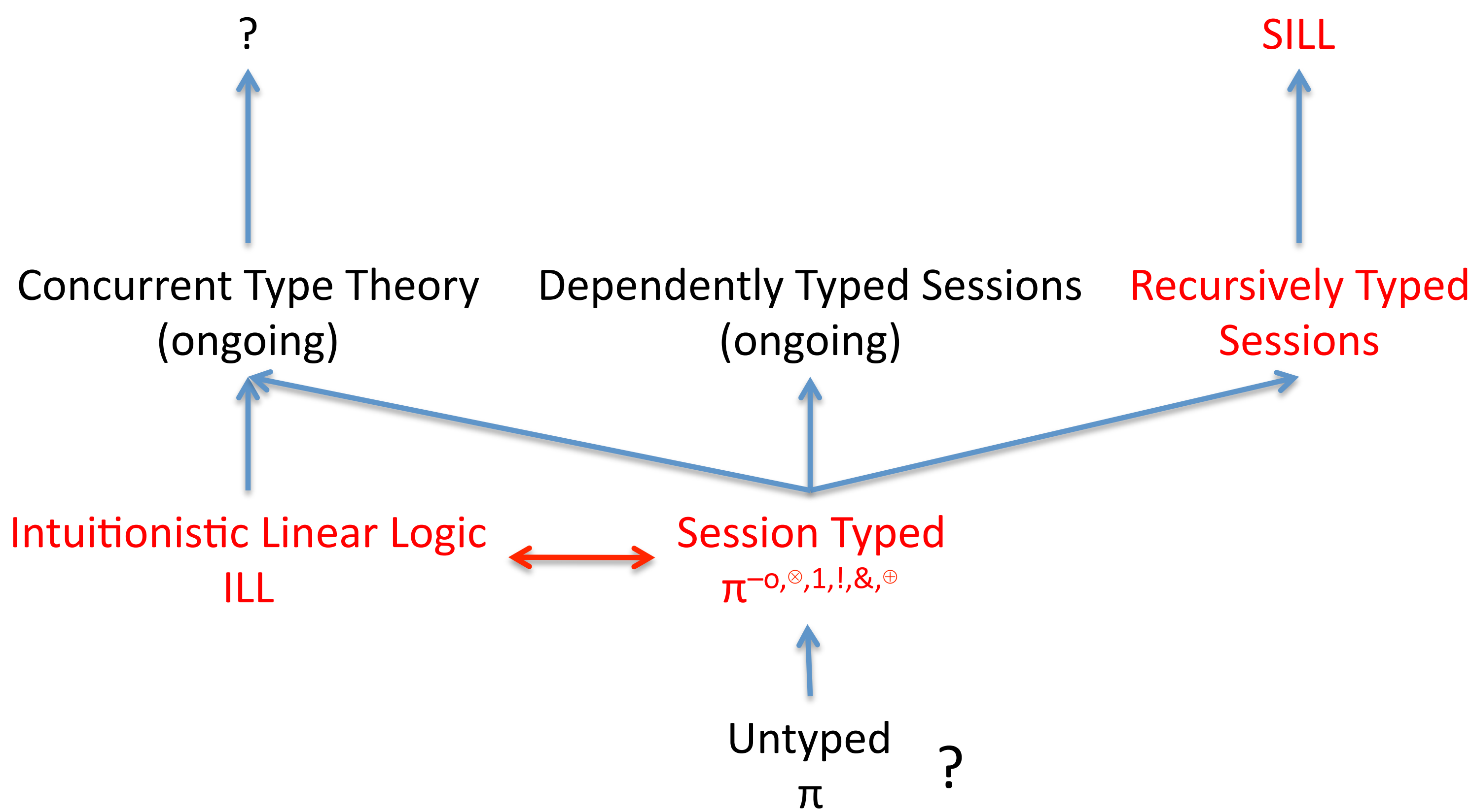
Much More to Say

- Theory of logical relations, observational equiv
- Hybrid linear logic with explicit worlds
- In progress
  - Dynamic monitoring and blame assignment
  - Refinement types / contracts
  - Full dependent types (= concurrent type theory)
  - Concurrent C0 (= imperative + threads)
  - New foundation of object-oriented programming

Foundations: Functions



Foundations: Processes



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 (♢A)
- 2012
  - CSL: asynchronous comm.
  - ESOP: logical relations
  - FOSSACS: functions as processes
- 2013
  - ESOP: behavioral polymorphism
  - ESOP: monadic integration (SILL)
- 2014
  - TGC: Coinductive types
  - Security domains (A @ w), spatial distribution

Collaborators

- Luís Caires, Bernardo Toninho, Jorge Peréz (Universidade Nova de Lisboa)
  - FCT and CMU|Portugal collaboration
- Dennis Griffith, Elsa Gunter (UIUC)
- Anna Gommerstadt, Limin Jia (CMU) [Dyn. Monitors]
- Stephanie Balzer (CMU) [New foundation for OO]
- Rokhini Prabhu, Max Willsey [Concurrent C0]
- Henry DeYoung (CMU) [From global to local types]
- Apologies for the lack of references to related work
- `git clone https://github.com/ISANobody/sill.git`