Concurrent Programming in Linear Type Theory

Frank Pfenning
Carnegie Mellon University
Joint work with Luís Caires, Bernardo Toninho, Jorge Peréz, Dennis Griffith, Elsa Gunter, et al.


ABCD
any body can dance

Outline

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

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 π-calculus)
  - labels (to indicate choice)

Curry-Howard Isomorphisms

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

Linear Logic: A New Foundation

- Linear propositions ⇔ session types
- Sequent proofs ⇔ process expressions
- Cut ⇔ process composition
- Identity ⇔ message forwarding
- Proof reduction ⇔ 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 ⇔ channel along which service is offered
- Later: CILL, sessions in a C-like language

Properties of SILL

- Type preservation
  - Entails session fidelity on processes
- Progress
  - Absence of deadlock
  - Absence of race conditions
- 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'Caml
- No inference rules, just intuition
- Examples
  - Endless streams of integers
  - Streams of integers
  - Stream filter
  - Prime sieve
  - Bit strings
  - Increment and addition

Stream of Numbers

- Data types

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

- { A } is type of process offering service A
- Session types

  $A ::= \ldots$

- 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
```

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

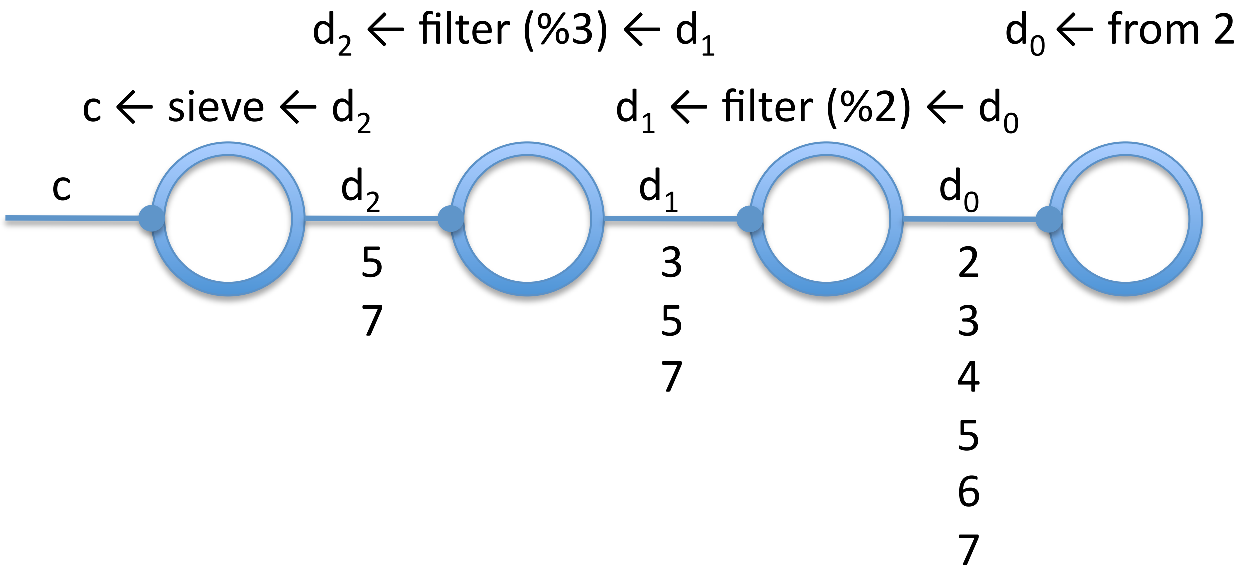
Finding the Next Element

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

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

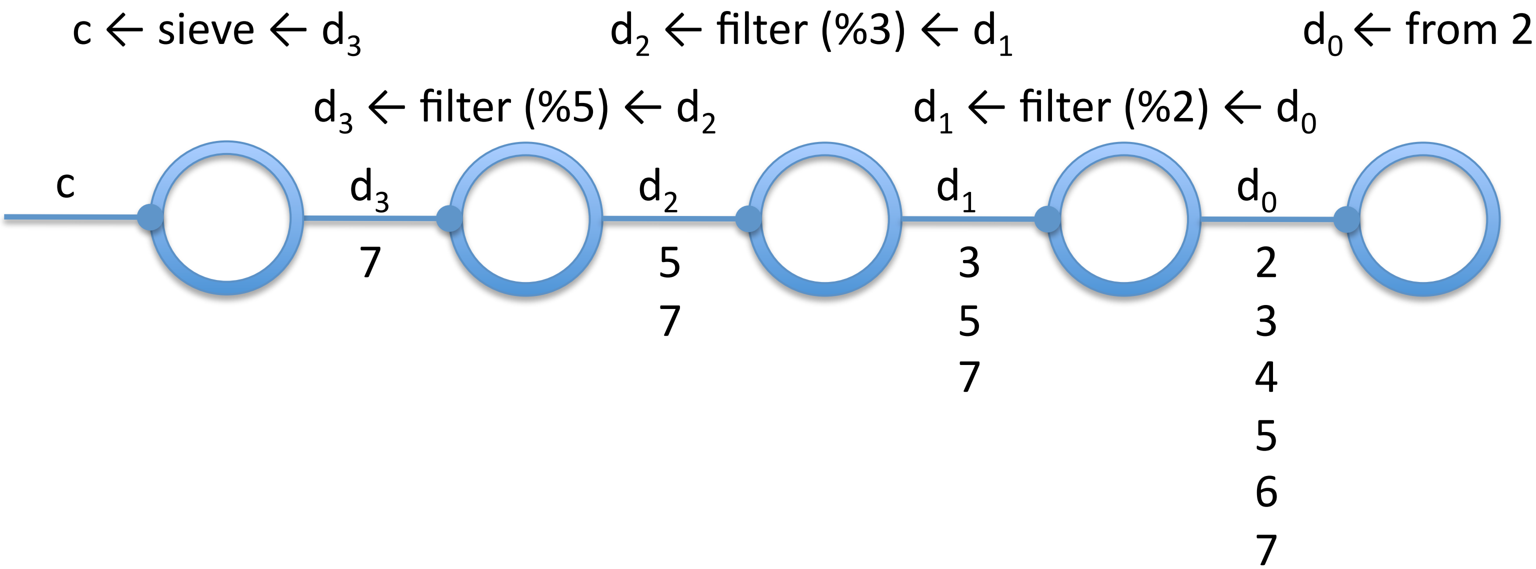
- filter/filterNext process identified with channel c

Prime Sieve



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

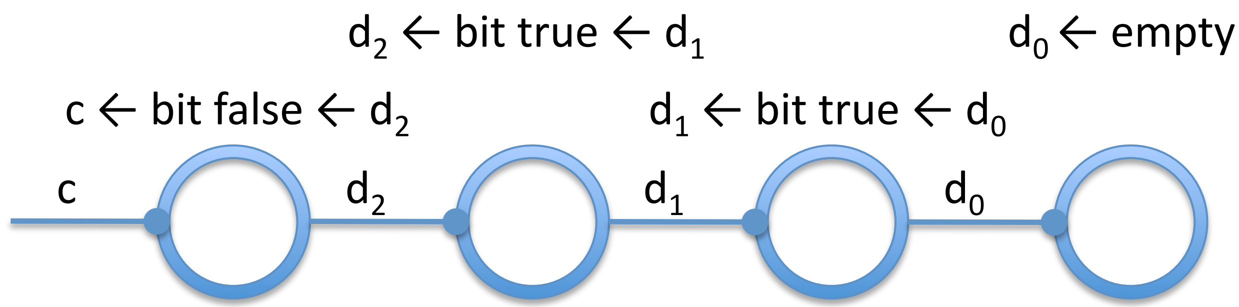
Primes

```
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

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 = ⊕{eps:1, bit:bool ∧ bits};

empty : {bits};
c ← empty =
  send c eps ;
  close c

bit : bool → {bits ← bits};
c ← bit b ← d =
  send c bit ;
  send c b ;
  c ← d;
```

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

Alternative Constructor

```
bits = ⊕{eps:1, bit:bool ∧ bits};

num : int → {bits};
c ← num n =
case n == 0
| true ⇒ send c eps ; close c
| false ⇒ send c bit ;
           send c (odd n) ;
           c ← 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 = ⊕{eps:1, bit:bool ∧ bits};
add : {bits ← bits, bits};
c ← add ← d, e =
  case (recv d)
   | eps ⇒ wait d ;
           c ← e
   | bit ⇒ b1 ← recv d ;
           case (recv e)
           | eps ⇒ wait e ;
                   send c bit;
                   send c b1;
                   c ← d
           | bit ⇒ b2 ← recv e ; …
```

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

Other Examples

- Data structures
  - Stacks, queues, hash tables, binary search trees
- 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
  - Bit strings
- 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

- $M : \{ A \leftarrow A_1, \ldots, A_n \}$ 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

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
- No behavioral polymorphism yet

Dynamic Semantics

- Three back ends
  - Synchronous threads
  - Asynchronous threads
  - Distributed processes
- Some cryptographic primitives
- Not released (but multiple “friendly” users)

Dynamic Type Checking

- May not trust all participating processes
- Type system compatible with
  - Value dependent types, e.g. nat = $\{x{:}int \mid x \geq 0\}$
  - Full dependent types, but still under investigation:
    - "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 ∃ as dependent ∧
- 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

Session Types in a C-like Language

- C0: a type-safe subset of C
  - Designed for teaching imperative programming, algorithms, and data structures to freshmen
  - Extended with contracts (pure boolean functions)
  - Contracts are crucial for design, proof, and testing
- C1: function pointers and polymorphism
- CILL: session-typed concurrency?

CILL

- Channels $c are linearly typed (as in SILL)
- Persistent channels $$c, variables x as usual
- Channel types must be loop invariants
  - lub at all join points in control-flow graph
- Possible with or without shared memory
  - No safety in the presence of shared memory
- Exploring robustness of SILL concepts in different setting

Integer Streams in CILL

```
choice intstream {
  int /\ choice intstream next;
  void stop;
};
typedef choice intstream ints;

ints $c from(int n) {
  while (true) {
    switch ($c) {
    case next:
      send($c, n);
      n = n+1;
    case stop:
      close($c);
    }
  }
}
```

Speculating on Contracts

```
ints $c from(int n)
//@requires n >= 0;
//@ensures $c = all_pos($c);
{
  while (true) {
    switch ($c) {
    case next:
      send($c, n);
      n = n+1;
    case stop:
      close($c);
    }
  }
}
```

- Value contracts must be pure boolean functions
- Channel contracts must be partial identity proc's

Partial Identity Process

```
ints $c all_pos(ints $d) {
  switch ($c) {
  case next:
    $d.next;
    int n = recv($d);
    if (n <= 0) abort;
    send($c, n);
    $c = all_pos($d);
  case stop:
    $d.stop; wait($d);
    close($c);
  }
}
```

- Synthesized in a type-directed way

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

Thanks!

- Luís Caires, Bernardo Toninho, Jorge Peréz (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

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
  - Coinductive types
  - Blame assignment