Distributed and Concurrent Objects Based on Linear Logic

Naoki Kobayashi **Akinori Yonezawa**

University of Tokyo

Distributed and Concurrent Computation Based on Linear Logic

Naoki Kobayashi **Akinori Yonezawa**

University of Tokyo

Outline

- Preliminary: Brief Introduction to Linear Logic
- Part I: Concurrent Linear Logic Programming
  - Review of Ideas
  - Limitations
- Part II: Recent Extensions of CLLP
  - Timed Concurrent Linear Logic Programming
  - Distributed Concurrent Linear Logic Programming
  - As a Whole
- Conclusion

Motivation

- **Develop uniform theoretical foundations for a wide range of concurrent/distributed programming languages based on Linear Logic**
  - **language design**

    What set of primitives is required for expressing a variety of mechanisms for concurrent/distributed programs?
  - **implementation techniques**
  - **type systems and program analyses**
  - **specification and verification**

Outline

- Preliminary: Brief Introduction to Linear Logic
- Part I: Concurrent Linear Logic Programming
  - Review of Ideas
  - Limitations
- Part II: Recent Extensions of CLLP
  - Timed Concurrent Linear Logic Programming
  - Distributed Concurrent Linear Logic Programming
  - As a Whole
- Conclusion

History of Concurrent Linear Logic Programming

- **Basic computation model based on linear logic**
  - **LO (Andreoli & Pareschi 1991)**
  - **Linear cc (Saraswat&Lincoln 1992?), Definitional Constraint Programming (Guo 1992?)**
  - **ACL, Higher-order ACL (Kobayashi&Yonezawa 1992, 1994)**
  - **temporal linear logic (Kanovich & Ito 1997)**
  - **distributed concurrent linear logic programming (Kobayashi, Shimizu & Yonezawa 1998)**

History of Concurrent Linear Logic Programming

- **Concurrent Objects [Andreoli & Pareschi 1991, 1992; Kobayashi&Yonezawa 1993,1994; Delzano&Martelli 1995, ...]**
- **Programming languages**
  - **LO (Andreoli&Pareschi 1992?)**
  - **Linear Janus (Tse & Saraswat 1993?)**
  - ***HACL* (Kobayashi, Shimizu & Yonezawa 1994)**

Preliminary:
Brief Introduction to Linear Logic

Introduction to Linear Logic

- **Consider the following propositions:**
  - ***A*: "You have one dollar."**
  - ***B*: "You can buy a chocolate bar."**
  - ***C*: "You can buy a candy."**
- **Suppose *A* "*implies*" *B* and *A* "*implies*" *C*: What does "*implies*" mean?**
  - **Implication of classical logic?**

    $A \supset B,\ A \supset C \Rightarrow\ A \supset (B \wedge C)$

    If you have one dollar, you can buy a chocolate bar and a candy??

Introduction to Linear Logic

- **What did we miss?**
  - **If you have one dollar, you can buy a chocolate and you lose one dollar.**

    $A \multimap B$

- **Is "if you have one dollar, you can buy a chocolate and you can buy a candy" true?**

  ✔ $A \multimap (B \,\&\, C)$

  ✗ $A \multimap (B \otimes C)$

  ✔ $A \otimes A \multimap (B \otimes C)$

$A1 \otimes \ldots \otimes An \multimap B$: each of $A1, \ldots, An$ can and must be used just once.

Introduction to Linear Logic: Summary

$A \multimap B$	if A holds, we can obtain B; after that, A no longer holds.
$A \,\&\, B$	We can assume whichever of A and B to be valid, but not both.
$A \otimes B$	Both A and B hold at the same time.
$!A$	Unbounded replication of A
0	zero element of $\otimes$ ($A \otimes 0 = 0 \otimes A = 0$)

e.g. ✔ $A \otimes (A \multimap B) \multimap B$

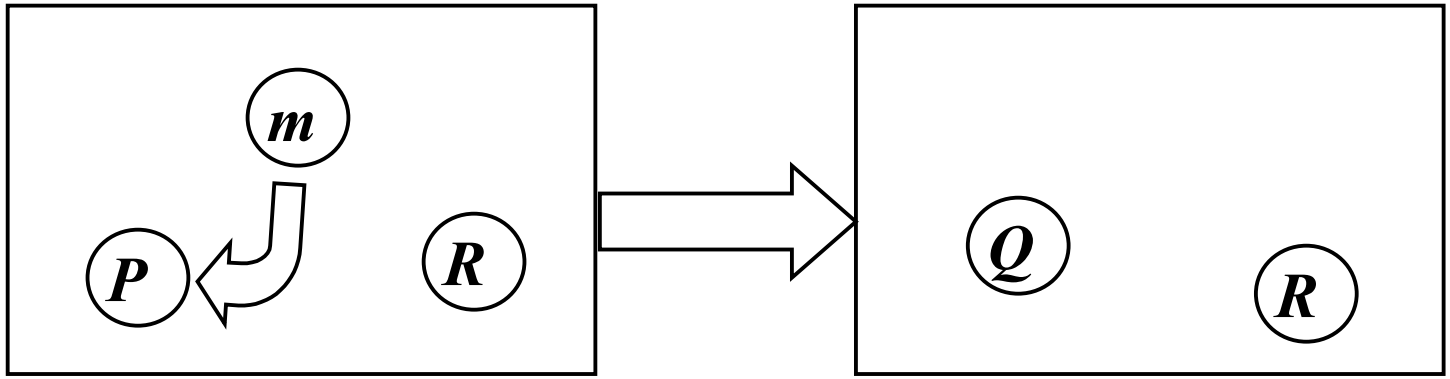
✗ $A \otimes (A \multimap B) \multimap A \otimes B$

✔ $!A \otimes (A \multimap B) \multimap !A \otimes B$

Part I:
Concurrent Linear Logic Programming

- **Basic Ideas**
- **Formal Correspondence between LL and Processes**
- **Advantages and Limitations**

Processes as formulas of linear logic



- A message is a consumable resource. (It disappears after being received by a process)
  - A message is represented by a linear logic formula.
- A process *P* receives a message *m* and then behaves like *Q*
  - *P* produces *Q* by consuming *m*
  - $P = m \multimap Q$
- Communication corresponds to deduction
  - $m \otimes (m \multimap Q) \multimap Q$ (c.f. $m \wedge (m \supset Q) \supset Q \wedge m \wedge (m \supset Q)$)

Processes as formulas of linear logic

- Run P and Q in parallel
  - P and Q are available at the same time
  - $P \otimes Q$
- Choice: behaves like P if a message m arrives and behaves like Q if n arrives
  - Can behave like $m \multimap P$ or $n \multimap Q$
  - $(m \multimap P) \mathbin{\&} (n \multimap Q)$

  $m \otimes ((m \multimap P) \mathbin{\&} (n \multimap Q)) \multimap P$

Processes as formulas of linear logic

- $\forall x.\ (c(x) \multimap P)$
  - For any v, if $c(v)$ holds, $[v/x]P$ can be obtained
  - A process that receives a value through channel c and behaves like P

    $\forall x.\ (c(x) \multimap P) \otimes c(v) \multimap [v/x]P$
- $\exists x.P$
  - For some x which we don't know, P holds
  - A process that creates a fresh name x and behaves like P

    $\exists c{:}\ int \to o.\ (c(1) \otimes \forall x{:}int.\ (c(x) \multimap P))$

    $\multimap \exists c{:}\ int \to o.[1/x]P$

Processes as LL formulas: Summary

0	**abort ($0 \otimes P = 0$)**
1	**inaction ($1 \otimes P = P$)**
$P \otimes Q$	**parallel composition**
$c(v)$	**message**
$\forall x.(c(x) \multimap P)$	**receiver process**
$P \,\&\, Q$	**choice**
$\exists x.P$	**name creation**
$!P$	**unbounded replication**

Concurrent Objects as LL formulas

!(counter (n, inc, read) –o

(inc (reply) –o (counter (n+1, inc, read) ⊗ reply())

&

read (reply) –o (counter (n, inc, read) ⊗ reply(n))))

See [Kobayashi&Yonezawa, TAPOS] for encoding of more complex mechanisms such as inheritance, intra-object concurrency, transaction-like method executions, etc.

Formal correspondence between LL formulas and processes

Theorem: $P \vdash_{LL}$ **if and only if** $P \rightarrow^* 0$

(for a subset of LL)

Def. $\mathrm{Succ}(P) = \{ Q \mid P \otimes Q \rightarrow^* 0 \}$

the set of testers P may pass

e.g. $m \multimap 0 \in \mathrm{Succ}(P)$

$\Rightarrow$ **A message m may eventually be sent by P**

Corollary: If P and Q are logically equivalent, then

$\mathrm{Succ}(P) = \mathrm{Succ}(Q)$

Proof: $\mathrm{Succ}(P) = \{R \mid P \otimes R \vdash_{LL}\} = \{R \mid Q \otimes R \vdash_{LL}\} = \mathrm{Succ}(Q)$

Advantages of LL-based Specification

- **Wide range of operational semantics can be captured**
  - **Can add a reduction rule $P \rightarrow Q$ as long as P –o Q**

    e.g. $n \otimes (m$ –o n –o $P) \rightarrow m$ –o P

    m –o $P \rightarrow m$ –o Q if $P \rightarrow Q$
- **More abstract semantics than process calculi**
  - **m –o n –o $P = n$ –o m –o $P = (m \otimes n)$ –o P**

    In order to specify a process that receives m and n, we need not specify the order of the messages

    (The order can be decided at a later stage so that deadlock cannot occur.)

c.f. $m.n.P \neq n.m.P$ in CCS

Advantages of LL-based Specification

- **Semantics can be understood by using "common sense"**
  - m #$"!)(n &%"!> P

    $\Rightarrow$ $m \otimes (n \multimap P)$

HACL: Concurrent Programming Language based on Linear Logic [Kobayashi, Shimizu&Yonezawa 1994]

- **core-ML + concurrency primitives based on (higher-order) linear logic**
- **ML-style type system**
- **High-level mechanisms such as concurrent objects are implemented by using higher-order processes**

Slogan = "ML in the world of concurrent programming languages"

Limitations of LL-based Specification

- **Cannot express precise operational semantics**
  - $m \multimap n \multimap P = n \multimap m \multimap P = (m \otimes n) \multimap P$
    - $\Rightarrow$ **Disadvantage in analyzing the behavior of existing systems or programs**
- **Not expressive enough to specify and reason about real programs**
  - **real-time systems/programs**
  - **distributed systems/programs**

Part II:
Extensions of CLLP

- Temporal/Timed Concurrent Linear Logic Programming
- Distributed Concurrent Linear Logic Programming
- As a Whole

Temporal Linear Logic [Kanovich&Ito, 1997]

- **Classical Logic**

  A: A **can be used an arbitrary number of times**

- **Linear Logic**

  A : A **can (and must) be used just once**

  $!A$: A **can be used an arbitrary number of times**

- **Temporal Linear Logic**

  A: A **can (and must) be used just once** ***now***

  $\bigcirc A$: A **can be used** ***at the next time***

  $\Box A = A \,\&\, \bigcirc A \,\&\, \bigcirc\bigcirc A \,\&\, \ldots$

  A **can be used** ***at some time from now***

Processes as TLL formulas

- $\Box(m \otimes \bigcirc \Box P)$
  - send a message m and after it is received, execute P (c.f. $\Box m \otimes \Box P$)
- $\Box(m \multimap \bigcirc \Box P)$
  - receive a message m and then execute P
- $\Box((m \otimes n) \multimap \bigcirc \Box P)$
  - receive m and n at the same time and then execute P

Example

$\Box(m \otimes \bigcirc \Box P) \otimes \Box(m \multimap \bigcirc \Box Q) \multimap \bigcirc (\Box P \otimes \Box Q)$

Real-time Processes

- $\Box_{S,\,T} A$: A must be used once between time S and T

- $!^1 \Box_{0,T} (m \multimap \Box P)$: waits for m until time T

  ($!^1 A$: A can be used *at most once*)

- $\exists d.(\Box_{0,T} ((m \multimap \Box P) \,\&\, (d \multimap \Box Q)) \otimes \Box_{T,T} d\,)$

  waits for m until time T and then behaves like P;

  behaves like Q if time-out occurs

Part II:
Extensions of CLLP

- Temporal/Timed Concurrent Linear Logic Programming
- Distributed Concurrent Linear Logic Programming
- As a Whole

Modal Linear Logic

- **[*L*]*A*** — ***A* holds at location *L***
- ***A*** — ***A* holds at the current location**

Example

!\[hotel\]($1 –o Coke)

⊗ !\[drug_store\]($1 –o (Coke ⊗ Coke ⊗ Coke))

Distributed Processes as MLL formulas

- **Migration**

  $[L]P$ execute P at location L

- **Providing the same functionality at different environments**

  $[L1]\ !\forall m,n,r.(mult(m, n, r) \multimap r\ (m \times n)) \otimes$

  $[L2]\ !\forall m,n,r.(mult(m, n, r) \multimap$

  if n=0 then $r(0)$

  else $\exists s.(mult(m,n\text{-}1,s)$

  $\otimes\forall x.(s(x) \multimap r(x+m))))$

  We can obtain $r(6)$ from $mult(2,3,r)$ at *L1* or *L2*.

Distributed Processes as MLL formulas

- **Specifying location-dependent information**

  ![*L1*]hostname(“L1”) ⊗ ![*L2*]hostname(“L2”) (= *P*)

  e.g. *P* ⊗ [*L1*] ∀*s*.(hostname(*s*) –o print(*s*))
  –o print(“L1”)

Distributed Processes as MLL formulas

- **Distinction between name passing and closure passing (distributed scope in Obliq)**

$c(\lambda x.P)$

$[L1]\ c(\lambda x.P) \otimes [L2]\ \forall z.(c(z) \multimap z(0))$

$\multimap [L2](c(\lambda x.P) \otimes \forall z.(c(z) \multimap z(0)))$

$\multimap [L2](\lambda x.P)\ 0 \equiv [L2][0/z]P$

$c(d) \otimes !\forall x.(d(x) \multimap P))\ (= Q)$

$[L1]\ Q \otimes [L2]\ \forall z.(c(z) \multimap z(0))$

$\multimap [L1]\ !\forall x.(d(x) \multimap P)) \otimes [L2]\ d(0)$

$\multimap [L1][0/z]P$

Mobile Ambient-like Computation

$(m[P])^* = [m]P^*$

ambient m that behaves like P

$(\text{open } m)^* = \Box\,(!\forall x.([m]x \multimap \bigcirc\, x) \otimes \bigcirc [m]0)$

open ambient m $(\text{open } m \mid m[P] \rightarrow P)$

$(\text{out } m\ n)^* = \Box\,(!\forall x.(x \multimap \bigcirc [..n][..m][n]x) \otimes \bigcirc 0)$

go outside ambient m

$(m[n[\text{out } m\ n \mid P] \mid Q] \rightarrow m[Q] \mid n[P])$

$(\text{in } m\ n)^* = \Box\,(!\forall x.(x \multimap \bigcirc\, [..n][m][n]x) \otimes \bigcirc\, 0)$

go inside ambient m

$(n[\text{in } m\ n \mid P] \mid m[Q] \rightarrow m[Q \mid n[P]])$

Part II:
Extensions of CLLP

- Temporal/Timed Concurrent Linear Logic Programming
- Distributed Concurrent Linear Logic Programming
- As a Whole

Formulas as Processes: As a Whole

- Pure LL formulas give abstract specification, but often imprecise
- Temporal/modal LL formulas give precise specification, but the cost of checking may be high

$\Rightarrow$ Choose an appropriate LL according to the required degree of preciseness of the specification

$\Rightarrow$ step-wise refinement of specification

  - First step: Use pure LL to give abstract specification and check correctness at logical level
  - Second step: Use temporal/modal LL to give precise specification and check the correctness in detail

Conclusion

- With linear logic, we can
  - express a wide range of concurrent/distributed computation model
  - give a more abstract specification than process calculi
- Precise specification can be obtained by extending LL with
  - temporal operators
  - location operators

Future Work

- Design of distributed, real-time programming languages based on LL
- Development of specification/verification tools based on LL
- Efficient proof systems for (a fragment of) temporal/modal LL
- Integration with other frameworks such as process algebras