Title: Concurrent Programming in Linear Logic

Author(s): Kobayashi, Naoki; Shimizu, Toshihiro; Yonezawa, Akinori

Citation: 数理解析研究所講究録 (1995), 902: 103-116

Issue Date: 1995-03

URL: http://hdl.handle.net/2433/59386

Type: Departmental Bulletin Paper

Textversion: publisher

Kyoto University
Concurrent Programming in Linear Logic

Naoki Kobayashi Toshihiro Shimizu
koba@is.s.u-tokyo.ac.jp shimizu@is.s.u-tokyo.ac.jp
Akinori Yonezawa
yonezawa@is.s.u-tokyo.ac.jp
Department of Information Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan

Abstract

HACL is a novel asynchronous concurrent programming language developed based on linear logic. It provides fruitful mechanisms for concurrent programming including higher-order concurrency and an elegant ML-style type system. Although HACL provides only a small set of primitive constructs, various constructs for communication and synchronization between processes are definable in terms of primitive asynchronous message passing and higher-order processes. In this paper, we demonstrate the power of HACL by showing several programming examples. For readers who are not familiar with linear logic and concurrent linear logic programming, we also give a brief introduction to the logical background of HACL.

1 Introduction

We developed a novel typed, higher-order, concurrent programming language called HACL based on a higher-order extension of ACL[3][2]. ACL is a variant of linear logic programming, whose operational semantics is described in terms of bottom-up search for a cut-free proof in linear logic. ACL naturally models concurrent computation based on asynchronous message passing, while traditional concurrent logic programming languages model stream-based communication. In the previous papers[3][2], we have shown most of computational models for asynchronous concurrent computation, including actor models, Linda, concurrent constraint programming, and asynchronous counterpart of CCS and π-calculus, can be naturally embedded in ACL. The power of ACL can be further strengthened by replacing the underlying logic with higher-order linear logic. The resulting framework, Higher-Order ACL[5] (in short, HACL), allows higher-order concurrent programming, where processes can be parameterized
by other processes, communication interfaces, and functions. HACL is also equipped
with an elegant ML-style type system, hence the type inference system infers the
most general types for untyped programs, and ensures that any well-typed program
never causes type mismatch error at run-time.

The purpose of this paper is to demonstrate the power of HACL by showing
several programming examples. Higher-order processes allow us to define a large
process by composing smaller subprocesses, by which improving the modularity and
code reuse of concurrent programs, and are also useful for making topologies between
multiple processes. Therefore, various constructs for concurrent programming, which
are introduced as primitives in an ad hoc manner in many concurrent programming
languages, can be defined using higher-order processes. HACL also allows object-
oriented style programming, as is shown in [4]. We believe that HACL can be used
not only as a specific concurrent programming language, but also as a common vehicle
to design novel, clean and powerful concurrent programming languages, to develop
program analysis techniques for concurrent programming languages, and to discuss
efficient implementation techniques.

Prototype interpreter of HACL and examples given in this paper are available via
anonymous ftp from camille.is.s.u-tokyo.ac.jp: pub/hacl.

2 Overview of Higher-Order ACL

This section informally overviews the syntax and operational semantics of higher-
order ACL. For the concrete definitions, please refer to [5][4]. For readers who are
not familiar with linear logic and concurrent linear logic programming, we attach a
brief introduction to both linear logic and its connection with concurrent computation
in Appendix.

As in the first-order ACL[3][2], computation in Higher-order ACL (in short, HACL)
is described in terms of bottom-up proof search in Girard's (higher-order) linear
logic[1]. Each formula of (a fragment of) higher-order linear logic is viewed as a
process, while each sequent, a multiset of formulas, is regarded as a certain state of
the entire processes. A message is represented by an atomic formula, whose predi-
cate is called a message predicate. The whole syntax of HACL process expressions is
summarized in Figure 2. The first column shows textual notations for process expres-
sions. The second column shows corresponding formulas of linear logic. In the figure,
x is ranged over variables, P over process expressions, and R over process expres-
sions of the form \( x(x_1, \ldots, x_n) \Rightarrow P \). A behavior of each process is naturally derived
from inference rules of linear logic. For example, from the inference shown in the
Figure 1, we can regard a formula \( \exists x_1 \cdots \exists x_n. (x(x_1, \ldots, x_n) \otimes P) \) as a process which
receives a message \( x(v_1, \ldots, v_n) \) and then behaves like \( P[v_1/x_1, \ldots, v_n/x_n] \). Message
predicates are created by the \( \$ (\forall, \text{in linear logic notation}) \) operator. \( \$ x.P \) creates a
message predicate and binds \( x \) to it in \( P \). A choice \(( m(x) \Rightarrow P) \& (n(x) \Rightarrow Q)\) either
receives a message \( m(v) \) and becomes \( P[v/x] \), or receives a message \( n(v) \) and becomes

\footnote{For other correspondence between process transitions and inference rules, please refer to [3][2].}
\[
\frac{x(v_1,\ldots,v_n), x(v_1,\ldots,v_n)^\perp \vdash P[v_1/x_1,\ldots,v_n/x_n], \Gamma}{x(v_1,\ldots,v_n)^\perp \otimes P[v_1/x_1,\ldots,v_n/x_n], \Gamma \vdash x_1 \exists x_n. (x(x_1,\ldots,x_n)^\perp \otimes P), x(v_1,\ldots,v_n), \Gamma}
\]

Figure 1: Inference corresponding to Message Reception

\(Q[v/x]\). A proc statement defines a recursive process just as a fun statement defines a recursive function in ML. For example,

```plaintext
proc forwarder m n = m x => n x;
val forwarder=proc: ('a->o)-> ('a->o)-> o
```

defines forwarder as a process which takes two message predicates \( m \) and \( n \) as arguments, and receives a value of \( x \) via a message \( m \) and sends it via a message \( n \). The line printed in slanted style is the system's output. It indicates that forwarder has the following type:

\[ \forall \alpha.((\alpha \rightarrow o) \rightarrow (\alpha \rightarrow o) \rightarrow o) \]

where \( o \) is the type for processes or messages. Therefore, the type of forwarder implies that it takes two message predicates, which should take an argument of the same type.\(^2\)

A process point is defined in HACL as follows:

```plaintext
proc point (x, y) (getx, gety, set) =
  getx(reply) =>
  (reply(x) | point(x,y)(getx, gety, set))
& gety(reply) =>
  (reply(y) | point(x,y)(getx, gety, set))
& set(newx,newy, ack) =>
  (ack() | point(newx,newy)(getx, gety, set));
val point=proc: 'a*'b->(('a->o)->o)*(('a->o)->o)*('a* 'b->o)->o
```

The process point has two internal state variables \( x \) and \( y \), and is parameterized by message predicates \( \text{getx}, \text{gety}, \text{set} \). If it receives a \( \text{getx} \) message, it sends the value of \( x \) to the reply destination \( \text{reply} \). If it receives a \( \text{set} \) message, it sends an acknowledgement message \( \text{ack}() \), and changes values of \( x \) and \( y \) to \( \text{newx} \) and \( \text{newy} \). The following expression creates a new point process, and sends to it a \( \text{getx} \) message.

```plaintext
$\text{getx.}$getx.$\text{set}.$
  (point(1.0, 2.0)(getx, gety, set) | $\text{reply.}$getx($\text{reply}$) | reply(x)==>... ))
```

\(^2\)More refined type system, where I/O information of messages is inferred, is given in [5].
3 Programming in HACL

This section demonstrates the power of HACL, by showing several examples of concurrent programming in HACL. In HACL, processes can be parameterized by other processes and interfaces (message predicates), just as functions can be parameterized by other functions in functional programming. This dramatically improves the modularity and code reuse of concurrent programs. We also show that HACL extended with records naturally allows concurrent object-oriented style programming.

3.1 Object-Oriented Style Programming

In the definition of point process in the previous section, readers might have noticed that it is very cumbersome to separately create getx, gety, and set message predicates, and to remember in which order they should be applied to the point process. In order to overcome this problem, it is very natural to introduce records, and put the three message predicates together in a record:

```
proc point (x, y) self =
  #getx self (reply) => (reply(x) | point(x,y) self)
& #gety self (reply) => (reply(y) | point(x,y) self)
& #set self (newx,newy) => (point(newx,newy) self);
val point = proc: 'a*'+b->'c::{getx:'a->o}->o, gety:'b->o}->o, set:'a*'+b->o}->o
```

*(field-name)* is an operator for field extraction. The resulting process definition just looks like the definition of concurrent objects[10]. *self*, which is a record consisting of at least three fields getx, gety, and set, can be considered as an *identifier* of a concurrent object point. *self* can be used for sending a message to itself, as in
ordinary object-oriented languages. The type inference is based on Ohori’s algorithm for polymorphic record calculus[6]. The inferred type indicates that point takes a pair of values of type $\alpha$ and $\beta$ as its first argument, and a record of type $\{\text{getx} : (\alpha \rightarrow o) \rightarrow o, \text{gety} : (\beta \rightarrow o) \rightarrow o, \text{set} : \alpha \times \beta \rightarrow o, \ldots\}$ as its second argument self.

The following is an expression for creating a new point object and sends to it a message getx.

\[
\$id.(\text{point}(1.0, 2.0) \text{ id} | \$\text{reply.}(\#\text{getx id reply | reply}(x)\Rightarrow \ldots))
\]

If a programmer wants to ensure that messages except for getx, gety, and set are never sent to the point object, he can explicitly attach the following type constraints on self:

\[
\begin{aligned}
\text{proc point}(x, y) \text{ (self): getx:}\ 'a, \text{ set:}\ 'c) = \\
\text{ & set self (newx,newy) => (point(newx,newy) self));}
\end{aligned}
\]

Then, sending a message other than getx, gety and set is statically detected as an invalid field extraction from the record self.

Based on these observations, we can easily construct a typed concurrent object-oriented programming language with inheritance and method overriding on top of HACL. By the type system of HACL, all message not understood errors can be statically detected at type-checking phase.[4]

### 3.2 Synchronization and Sequencing between Processes

It is often important to write synchronization or serialization of execution of processes. Constructs for such purposes can be defined using higher-order processes. For example, consider the following process seq:

\[
\begin{aligned}
\text{proc seq (p1, p2) = $token.(p1(token) | token()$=> p2());}
\end{aligned}
\]

\[
\begin{aligned}
\text{val seq = proc: ((unit->o)->o)*(unit->o)->o}
\end{aligned}
\]

\[
\begin{aligned}
\text{seq (soutput("hello "), fn ()=>output("world\n"));}
\end{aligned}
\]

seq(p1, p2) executes a process p1(token) at first, and p2() is invoked only when the process p1 sends a message token(). soutput is a built-in process which takes a string s and a message predicate m as an argument. It displays the string s, and then sends the message m().

It is also easy to write processes for realizing broadcast, barrier synchronization, etc. For example, the following process make_group takes a message predicate m and a list of processes procs as arguments. Sending a message m(x) from the externals causes a value of x is broadcasted to a process in procs.
local
proc broadcast x ns =
case ns of
  nil => _
  | n::ns' => (n(x) | broadcast x ns')
in
proc broadcaster m ns =
m(x) => (broadcast(x, ns) | broadcaster m ns)
end
local
proc parapp procs ms =
case procs of
  nil => _
  | pr::procs' => (case ms of m::ms' => (pr(m) | parapp procs' ms'))
in
proc make_group m procs =
  let
    val gsize = length(procs)
in
    make_mpred_list(gsize, fn x => (broadcaster m x | parapp procs x))
  end
end

3.3 Higher-Order Processes for Making Topologies between Processes

Higher-order processes in HACL are especially useful for making topologies between processes. For example, consider the following higher-order process `linear`:

```
proc linear procs left right =
case procs of
  nil => _
  | pr::nil => pr(left, right)
  | pr::procs2 => $link.(pr(left, link) | linear procs2 link right);
val linear = proc: (('a->o)*('a->o)->o list->'(a->o)->'(a->o)->o
```

We use the notation of ML for the case statement and list expressions. The higher-order process `linear` takes a list of processes `procs` as the first argument, and connects them in a linear topology. A ring structure can be constructed by just assigning the same message predicate to the left and right of `linear` process.

```
proc ring procs = $m.(linear procs m m);
val ring = proc: (('a->o)*('a->o)->o list - o
```

The dining philosopher problem can be described by defining a behavior of each philosopher, then connecting them by `ring`:
半分に相当する$P_m(y)$の部分を接続した経路である。

【事実3.1】
（1）$v \in V_{\top b}(P_m(x))$ならば，$\text{label}(\delta^+_P(x) \oplus P_m(y)(v)) = \text{label}(\delta^+_P(x)(v))$。
（2）$v \in V_{\bot b}(P_m(y))$ならば，$\text{label}(\delta^+_P(x) \oplus P_m(y)(v)) = \text{label}(\delta^+_P(y)(v))$。

（証明略）

【命題3.2の証明の続き】$z$を$x$の上半分と$y$の下半分を接続した2次元テープ，すなわち，

i) $\delta_1(z) = \delta_2(z) = 2m$，
ii) $z([(1,1),(m,2m)]) = x([(1,1),(m,2m)])$，
iii) $z([(m+1,1),(2m,2m)]) = y([(m+1,1),(2m,2m)])$

とする。ここで，初期計算状況の節点$v_0$を始点とする$P_m(x) \oplus P_m(y)$上の経路巡回を考える：

① $v := v_0$とおく。
② もし，$\delta^+(v) = \{v\}'$なる節点$v' \in V(P_m(x) \oplus P_m(y))$が存在するならば，$v := v'$とおいて②を繰り返す。そうでなければ，停止する。

事実3.1より，変数$v$の値の系列が（受理計算状況の節点$v_k$もしくは$u_j$のどれかを終点とする）$M$の$z$上のある$L(m)$領域限定受理計算経路$P_m(z)$を構成することは明らか。以上により，$z \in T(M)$が導かれたが，これは$T(M) = T_1$なる最初の仮定に反する（$z \notin T_1$に注意）。□

【例3.1】繁雑さを避けるため，ここでは節点$v$とそのラベル$\text{label}(v)$を区別しないことにする。2次元テープ$x$と$y$に対する受理計算経路$P_m(x), P_m(y)$がそれぞれ，

$\ldots, v_1, u_1, \ldots, v_2, u_2, \ldots, v_3, u_3, \ldots, v_4, u_4, v_4,$

$\ldots, v_5, u_5, \ldots, v_6, u_6, v_6, \ldots, v_7, u_7, \ldots, u_8, v_8 \ldots$

ならばに，

$\ldots, v'_1, u_1, \ldots, v'_2, u_2, \ldots, v'_3, u_3, \ldots, v'_4, u_4, v_4,$

$\ldots, v'_5, u_5, \ldots, v'_6, u_6, v_6, \ldots, v'_7, u_7, \ldots, u'_8, v_8 \ldots$

と表されるものとする（図2(a)ならびに(b)参照）。このとき，$x$の上半分と$y$の下半分を接続した2次元テープ$z$に対して，次のような受理計算経路$P_m(z)$が構成される。

$\ldots, v_1, u_1, \ldots, u'_6, v_4, \ldots, v_5, u_5, \ldots, u'_4, v_2, \ldots, v_3, u_3, \ldots, u'_2, v_8 \ldots$

（図2(c)参照）□

【補題3.2の証明の続き】明らかに，$|V(m)| = (2m)^e$である。一方，各$m \geq 1$に対して，

$C(m) = \{ \text{Cross_Pair}(P_m(x)) \mid \exists x \in V(m) \ [ P_m(x) \text{はMのX上の} \ L(m) \text{領域限定受理計算経路} \} \}$

とおけば，

$|C(m)| \leq \sum_{L=0}^{e[m]/2} \binom{e[m]}{L} \binom{e[m]}{L} \leq \sum_{L=0}^{e[m]} e[m] e[m] 2 2 \cdot e[m] \leq 2\cdot e[m]$

が成り立つ。ここに，$e[m] = s(2m+2)L(2m)\cdot t(2m)$, $s$は$M$の有限制御部の状態数，$t$

2. グラフ$G$の節点集合，枝集合が，それぞれ$V(G), E(G)$のとき，各$v \in V(G)$に対して，$
\delta^+_G(v) = \{v' \in V(G) \mid (v,v') \in E(G)\}$と定義する。
We can define a rectangle process by composing two point processes as follows.

```haskell
local
  proc rectangle'(p1, p2) self =
    \#getx self (reply) =>
      ($m. (#getx p1 m | m(x1) => (#getx p2 m | m(x2) =>
          (reply((x1+x2)/2.0) | rectangle'(p1, p2) self)))
    & \#gety self (reply) =>
      ($m. (#gety p1 m | m(y1) => (#gety p2 m | m(y2) =>
          (reply((y1+y2)/2.0) | rectangle'(p1, p2) self)))
    & \#move self (dx, dy, ack) =>
      ($ack2. (#move p1 (dx, dy, ack2)
            | ack2()) => (#move p2 (dx, dy, ack) | rectangle'(p1, p2) self)
    in
  proc rectangle (point1, point2) self =
    $id1.$id2. (point1 id1 | point2 id2 | rectangle'(id1, id2) self)
end;
```

A particular instance of rectangle process is created by instantiating p1 and p2 to point processes as follows:

```haskell
proc new_rectangle (x1, y1, x2, y2) self =
  rectangle(point1(x1, y1), point2(x2, y2)) self;
val new_rectangle = proc: real*real->'a::{getx:(real->o)->o, gety:(real->o)->o, move:real*real*(unit->o)->o}->o
```

Important point about the above rectangle process is that it is independent of a particular implementation of point processes. Let point2 be another implementation of the point process. Then, we can make a rectangle process by applying point2 instead of point:

```haskell
  proc new_rectangle2 (x1, y1, x2, y2) self =
    rectangle(point2(x1, y1), point2(x2, y2)) self;
```

### 3.5 Atomic Execution of Methods

Some of successive executions of methods should be made atomic. For example, consider the point process in the previous section. When implementing a move method using getx, gety, and set methods, one might want to invoke getx, gety, and set methods atomically, so that values of x and y are not changed by other objects between executions of the getx method and the set method. The following 1point process allows such an atomic execution of methods.
local
proc philosopher n (lfork, rfork) =
(* get left and right forks *)
lfork()=>rfork()=>
(* start eating *)
seq(soutput(makestring(n)"I am eating\n"),
(* finish eating *)
  fn ()=seq(soutput(makestring(n)" finished\n"),
(* release forks, and *)
  fn ()=1fork() | rfork()
(* repeat the same behavior *)
  l philosopher n (lfork, rfork)))

in
proc philosopher_and_fork n (lfork, rfork) =
  lfork() | philosopher n (lfork, rfork)
end;
fun nlist n f = if n=0 then nil else f(n)::(nlist (n-1) f);
proc philosophers () = ring (nlist 5 philosopher_and_fork);

philosophers();
5: I am eating
5: finished
3: I am eating
1: I am eating
3: finished
....

In the same manner, we can also define higher-order processes for making any topologies between processes, such as mesh topology and torus topology. Note that this kind of programming was cumbersome with traditional concurrent object-oriented languages such as ABCL[9].

3.4 Hierarchical Construction of Processes

By using higher-order processes, we can construct a large process by composing smaller subprocesses. Although it might look possible with only first-order processes (as is shown in [3]), higher-order processes are very important in the sense that it makes a process independent of a particular implementation of its subcomponent processes, by which enhancing the modularity.

For example, consider the following point process:

proc point (x, y) self =
  #getx self (reply) => (reply(x) | point(x, y) self)
& #gety self (reply) => (reply(y) | point(x, y) self)
& #move self (dx, dy, ack) =>
proc lpoint (x, y, oldself) self =
    #lock self(reply) => $key.(reply(key) | lpoint (x, y, self::oldself) key)
    & #unlock self() =>
        (case oldself of self'::oldself' => lpoint (x, y, oldself') self')
    & #getx self(reply) => ...
    & #gety self(reply) => ...
    & #set self(newx, newy, ack) => ...

proc new_lpoint (x, y) self = lpoint (x, y, []) self;

Then, a lpoint can be moved as follows: (1) send lock message to the lpoint process, and get key, (2) send getx and gety messages to key, and get values of x and y, (3) send a set message to key, and (4) send a unlock message. This kind of programming is possible because communication interfaces (message predicates, or identity) of each process can be directly manipulated in HACL.

3.6 Invocation of Processes from Functions

HACL allows functions to be defined using a fun statement just as in ML, and to be called from a process. Conversely, HACL also has the catch expression to invoke processes from a function. The catch expression takes the following form:

\[
\text{catch } m \text{ in } p \text{ end,}
\]

which means “create a new message predicate \( m \), and execute a process \( p \). If a message \( m(v) \) is eventually sent, the whole expression is evaluated to \( v \).” \( e[(\text{catch } m \text{ in } p \text{ end})/x] \) can be considered an abbreviated form of the following expression.

\[
\text{fun fib}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ else if } n = 1 \text{ then 1 else } fib(n-1) + fib(n-2); \end{cases}
\]

The following function pfib computes the fibonacci number in parallel.

\[
\text{fun pfib}(n) = \begin{cases} fib(n) & \text{if } n < 5 \text{ then fib(n) else catch result in } \begin{cases} \text{fun fib}(n) \end{cases} \end{cases}
\]

If \( n \) is less than 5, pfib\((n)\) computes the fibonacci number sequentially. Otherwise computes pfib\((n-2)\) and pfib\((n-1)\) in parallel, and add results. In this manner, processes can be easily invoked from functions, and vice versa.
4 Related Work

Pierce and Turner [7] are developing a concurrent language called PICT, based on Milner's $\pi$-calculus. We believe that concurrent linear logic programming potentially offers a more fruitful programming model than $\pi$-calculus; it can formalize many kinds of communication [2] including Linda's generative communication, and also allow a natural integration with traditional logic programming.

5 Conclusion

We proposed HACL, a higher-order extension of the concurrent linear logic programming language ACL. Outstanding features of HACL include higher-order processes, which dramatically enhance the modularity of concurrent programs, and the elegant ML-style type system. We demonstrated the power of HACL by showing several examples. We have already implemented a prototype system of HACL. We are currently developing an efficient compiler system for HACL. Our current work also includes the design of a high-level concurrent object-oriented language on top of HACL, and the development of static analysis techniques for concurrent programs based on HACL. It would be also interesting to integrate HACL with traditional logic programming.

References


A The Essence of Concurrent Linear Logic Programming

A.1 Brief Guide to Linear Logic

This section gives an intuitive idea for understanding Girard’s linear logic[1], using the well-known examples.[8] Let us consider the following three formulas $A, B$ and $C$:

$A$ : You have one dollar.
$B$ : You can buy a chocolate.
$C$ : You can buy a candy.

We assume here that each of a chocolate and a candy costs $1. Then, it is trivial that $A$ implies $B,$ and $A$ implies $C.$ But what’s the meaning of “implies”? Let’s consider it as an implication in classical logic, that is, interpret “$A$ implies $B$” as “$A \supset B$” and “$A$ implies $C$” as “$A \supset C.$” Then, we can deduce $A \supset (B \land C)$ from $A \supset B$ and $A \supset C.$ Therefore, we are led to the strange conclusion that “if you have one dollar, then you can buy a chocolate and a candy.”

What was wrong with the above reasoning? It was the interpretation of “implies.” If you have one dollar, then you can buy a chocolate, but at the same time, you lose one dollar, that is, you can deduce $B$ from $A$, but you do not have $A$ any more. In order to express the above “implies,” we need to introduce a new implication ‘$\multimap$’ of linear logic, which is called linear implication. $A \multimap B$ means that if we have $A,$ we can obtain $B$ by consuming $A.$ Therefore, each formula of linear logic should be considered as a kind of consumable resource.

Going back to the above example, we also need to reconsider the interpretation of “and.” If you have one dollar, it is both true that you can buy a chocolate and that you can buy a candy. But they cannot be true at the same time. In order to express it, we need to use ‘&’, one of linear logic conjunctions. $A \multimap B \& C$ means that “we can obtain any one of $B$ and $C$ from $A,$ but not both at the same time”. In contrast, the other linear logic conjunction ‘$A \otimes B$’ means that we have both $A \otimes B$ at the same time. Therefore, $A \otimes A \multimap B \otimes C$ means that “If you have one dollar and one dollar (that is, two dollars), you can buy both a chocolate and a candy at the same time (and you lose two dollars!).”


Linear logic disjunction $\oplus$ and $\otimes$ is respectively de Morgan dual of $\&$ and $\otimes$:

\[
(A \& B)'^\perp \equiv A'^\perp \oplus B'^\perp  \\
(A \otimes B)'^\perp \equiv A'^\perp \otimes B'^\perp
\]

where $(\_)'\perp$ is a linear negation. $A \oplus B$ means that "at least one of $A$ and $B$ holds." (Compare with $A \& B$.) $A \otimes B$ can be also defined as $A'^\perp \ominus B'^\perp \equiv B'^\perp \ominus A$. $1$ and $\perp$ is respectively unit of $\otimes$ and $\&$. The above Girard's choice of symbols for connectives seems to be motivated by the following distributive laws:

\[
A \otimes (B \oplus C) \equiv (A \otimes B) \oplus (A \otimes C)  \\
A \& (B \otimes C) \equiv (A \& B) \otimes (A \& C)
\]

Girard also introduced exponential '!' for expressing unbounded resource. If we have '!'$A'$, we can use $A$ any number of times. '?' is de Morgan dual of '!'..

## A.2 Connection between Linear Logic and Concurrent Computation

What is the connection between linear logic and concurrent computation?³

Consider the situation where multiple processes perform computation while communication with each other via asynchronous message passing. Each message disappears after read by a process, that is, a message is a consumable resource. Therefore, it is natural to interpret a message as a formula of linear logic. From now on, we represent a message by an atomic formula of linear logic. How about process? Consider a process $A$, which waits for a message $m$ and behaves like $B$ after reception. $A$ consumes $m$ and produces $B$, hence $A$ is interpreted by a linear logic implication $m \rightarrow \ominus B$. Therefore, a process is also represented by a formula of linear logic. Consumption of a message $m$ by a process $m \rightarrow \ominus B$ is represented by the following deduction:

\[
m \otimes (m \rightarrow \ominus B) \otimes C \rightarrow B \otimes C
\]

where $C$ can be considered as other processes and messages, or an environment. Note that we cannot interpret it by connectives of classical logic. With classical logic, we can deduce

\[
m \land (m \sqcup B) \supset B
\]

but we can also deduce

\[
m \land (m \sqcup B) \supset m \land (m \sqcup B) \land B
\]

which implies that the original message $m$ and process $m \sqcup B$ may still remain after message reception!

Let us try to interpret other connectives. $A \otimes B$ means that we have both a process $A$ and a process $B$ at the same time, i.e., we have two concurrent processes $A$ and

---

³This material gives a connection based on proof search paradigm. Regarding the alternative approach, proof reduction paradigm, please refer to other literatures.
B. Therefore, \( \otimes \) represents a concurrent composition. If \( A \) is a message, we can also interpret \( A \otimes B \) as a process which throws a message \( A \) and behaves like \( B \).

\((m_1 \rightarrow o A_1) \& (m_2 \rightarrow o A_2)\) means that we have any one of \( m_1 \rightarrow o A_1 \) and \( m_2 \rightarrow o A_2 \), but not both at the same time. So, if there is a message \( m_1 \), we can obtain a process \( A_1 \), and if there is \( m_2 \), we can obtain \( A_2 \). But even if we have both \( m_1 \) and \( m_2 \), we can only obtain either of \( A_1 \) and \( A_2 \) (compare with \((m_1 \rightarrow o A_1) \otimes (m_2 \rightarrow o A_2)\)):

\[
\begin{align*}
m_1 \otimes ((m_1 \rightarrow o A_1) \& (m_2 \rightarrow o A_2)) \otimes C - o A_1 \otimes C \\
m_2 \otimes ((m_1 \rightarrow o A_1) \& (m_2 \rightarrow o A_2)) \otimes C - o A_2 \otimes C
\end{align*}
\]

Therefore, \((m_1 \rightarrow o A_1) \& (m_2 \rightarrow o A_2)\) can be interpreted as a process which waits for any one of \( m_1 \) and \( m_2 \), and becomes \( A_1 \) or \( A_2 \) depending on the received message.

Let us consider predicate logic. An atomic formula \( m(a) \) can be interpreted as a message carrying a value \( a \). A predicate \( m \) can now be considered a communication channel, or a port name. Then, what does \( \forall x. (m(x) \rightarrow o A(x)) \) mean? It implies that for any \( x \), if \( m(x) \) holds, we can deduce \( A(x) \). Computationally, it can be interpreted as "for any \( x \), if there is a message \( m \) carrying \( x \), we obtain a process \( A(x) \)." Therefore, \( \forall x. (m(x) \rightarrow o A(x)) \) represents a process which receives a value of \( x \) via a message \( m \), and becomes \( A(x) \):

\[ m(a) \otimes \forall x. (m(x) \rightarrow o A(x)) \otimes C - o A(a) \otimes C \]

How about existential quantification? \( \exists x. A \) hides \( x \) from externals. Therefore, \( x \) can be used as a private name, which is used for identifying a message receiver.

There are several minor variants for expressing communication. For example, if we allow a formula of the form: \( \forall x. (m(a, x) \rightarrow o A(x)) \), it receives only a message \( m \) whose first argument matches to \( a \), hence we can realize Linda-like generative communication. \( \forall x. (m(x, x) \rightarrow o A(x)) \) receives only a message \( m \) whose first and second arguments match. A formula \( \forall x. \forall y. (m(x) \otimes n(y) \rightarrow o A(x, y)) \) receives a value \( x \) via \( m \), and \( y \) via \( n \), and behaves like \( A(x, y) \). Since the formula is equivalent to

\[ \forall x. (m(x) \rightarrow o \forall y. (n(y) \rightarrow o A(x, y))) \equiv \forall y. (n(y) \rightarrow o \forall x. (m(x) \rightarrow o A(x, y))) \]

it can receives a message \( m(a) \) and \( n(b) \) in any order. Whether to delay the message reception until both \( m(a) \) and \( n(b) \) are ready or not is up to the choice of language designer.

Figure 3 summarizes the connection between linear logic formula and process. Some of concurrent linear logic programming languages, including ACL and Higher-Order ACL, are formalized using dual connectives. We show this dual representation in the second column (In the second column, positive and negative atoms are also exchanged).

### A.3 Higher-Order ACL

Higher-Order ACL[5] is based on Girard's second-order linear logic[1] with \( \lambda \)-abstraction. Note that with higher-order linear logic, a predicate can take predicates as arguments, hence we can express processes which take processes as arguments. Quantifications
Linear Logic Formula | Dual Representation | Process Interpretation
---|---|---
1 | ⊥ | inaction
m (atomic formula) | m | message
A ⊗ B | A®B | concurrent composition
∀x.(m(x)¬o A(x)) | ∃x.(m(x)⊥ ⊗ A(x)) | message reception
R₁&R₂ | R₁ ⊕ R₂ | selective message reception
∃x.A | ∀x.A | name creation
!p | ?p | unbounded replication

Figure 3: Connection between formula and process

can be also over predicates, hence processes can communicate processes and communication channels via a message.

Higher-Order ACL is equipped with an ML-style type system. Note that types in Higher-Order ACL have nothing to do with Curry-Howard isomorphism, because we regard formulas as processes, not as types. Higher-Order ACL took the similar approach to λProlog in introducing types. We have a special type ‘o’ for propositions. Computationally, it corresponds to the type of processes and messages. int → o is a type of predicate on integers. Computationally, it is a type of processes or messages which take an integer as an argument. (int → o) → o is a type of processes or messages which take a process or a message that takes an integer as an argument. (int → o) → int is a type of functions which take a process that takes an integer as an argument, and returns an integer. Processes can be defined and used polymorphically, by using let proc p(x) = e₁ in e₂ end statement, which is analogous fun statement in ML. Process definitions could be expressed by a formula !(p(x)¬o e₁), (or !(p(x)¬o e₁) by the dual encoding), which corresponds to clause in traditional logic programming. However, we preferred to interpret it as let p = fix(λp.λx.e₁) in e₂ end where fix is a fixpoint operator, for the introduction of ML-style polymorphic type system. It makes sense, because unlike traditional logic programming, we can restrict so that each predicate has only one definition clause.