数理解析研究所講究録
902巻 1995年 103-116

Concurrent Programming in Linear Logic

Naoki Kobayashi
koba@is.s.u-tokyo.ac.jp

Toshihiro Shimizu
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 predicate 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 expressions. 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 expressions of the form $x(x_1,\ldots,x_n)$`=>`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 $\exists x_1 \cdots \exists x_n.(x(x_1,\ldots,x_n)^{\perp} \otimes P)$ as a process which receives a message $x(v_1,\ldots,v_n)$ and then behaves like $P[v_1/x_1,\ldots,v_n/x_n]$.[1] Message predicates are created by the `$`($\forall$, in linear logic notation) operator. `$x.`P creates a message predicate and binds x to it in P. A choice `(m(x)=>P)&(n(x)=>Q)` either receives a message `m(v)` and becomes `P[v/x]`, or receives a message `n(v)` and becomes

[1] For other correspondence between process transitions and inference rules, please refer to [3][2].

$$\dfrac{\dfrac{\vdash x(v_1,\ldots,v_n), x(v_1,\ldots,v_n)^{\perp} \quad \vdash P[v_1/x_1,\ldots,v_n/x_n], \Gamma}{\vdash x(v_1,\ldots,v_n)^{\perp} \otimes P[v_1/x_1,\ldots,v_n/x_n], x(v_1,\ldots,v_n), \Gamma}\otimes}{\vdash \exists x_1 \cdots \exists x_n.(x(x_1,\ldots,x_n)^{\perp} \otimes P), x(v_1,\ldots,v_n), \Gamma}\exists$$

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,

```
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:

$$\forall\alpha.((\alpha \to o) \to (\alpha \to o) \to o),$$

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:

```
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 `getx`, `gety`, and `set`. If it receives a `getx` message, it sends the value of `x` to the reply destination `reply`. If it receives a `set` message, it sends an acknowledgement message `ack()`, and changes values of `x` and `y` to `newx` and `newy`. The following expression creates a new `point` process, and sends to it a `getx` message.

```
$getx.$gety.$set.
    (point(1.0, 2.0)(getx,gety,set) | $reply.(getx(reply) | reply(x)=>... ))
```

[2]More refined type system, where I/O information of messages is inferred, is given in [5].

Textual Notation	Linear Logic Formula	Description
-	$\bot$	inaction
$x(e_1,\ldots,e_n)$ | P	$x(e_1,\ldots,e_n) \parr P$	*1
$x(x_1,\ldots x_n)$ => P	$\exists x_1 \cdots \exists x_n.(x(x_1,\ldots,x_n)^\bot \otimes P)$	*2
$R_1 \& R_2$	$R_1 \oplus R_2$	behaves like R_1 or R_2
$P_1 \mid P_2$	$P_1 \parr P_2$	parallel composition
$\$x.P$	$\forall x.P$	name creation
$?P$	$?P$	unbounded replication
let proc $Ax_1 \ldots x_n = P$ in $Ae_1 \ldots e_n$ end	$\mathbf{fix}(\lambda A.\lambda x_1.\cdots\lambda x_n.P)e_1 \cdots e_n$	process definition

*1 $\cdots$ sends a message $x(e_1,\ldots,e_n)$, and then behaves like P
*2 $\cdots$ receives a message $x(e_1,\ldots,e_n)$, and then behaves like $P[e_1/x_1,\ldots,e_n/x_n]$

Figure 2: Syntax of HACL process expressions

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 α and β as its first argument, and a record of type $\{getx : (\alpha \to o) \to o, gety : (\beta \to o) \to o, set : \alpha \times \beta \to o, \ldots\}$ as its second argument `self`.

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

```
$id.(point (1.0, 2.0) id  | $reply.(#getx id reply | reply(x)=> ...)
```

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`:

```
proc point (x, y) (self:{getx:'a, gety:'b, set:'c}) =
  #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->{getx:('a->o)->o, gety:('b->o)->o, set:'a*'b->o}->o*

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`:

```
proc seq (p1, p2) = $token.(p1(token) | token()=> p2());
```

val seq = proc: ((unit->o)->o)(unit->o)->o*

```
seq (soutput("hello "), fn ()=>output("world\n"));
```

hello world

`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】[2]

(1) $v\in V_{top}(P_M(x))$ならば，$label(\delta^{+}_{P_M(x)\ominus P_M(y)}(v))=label(\delta^{+}_{P_M(x)}(v))$．

(2) $v\in V_{bot}(P_M(y))$ならば，$label(\delta^{+}_{P_M(x)\ominus P_M(y)}(v))=label(\delta^{+}_{P_M(y)}(v))$．

（証明略）

【命題３．２の証明の続き】ｚをｘの上半分とｙの下半分を連接した２次元テープ，すなわち，

i) $\ell_1(z)=\ell_2(z)=2m$,

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 $P_M(x)\ominus P_M(y)$ 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$v'\in V(P_M(x)\ominus P_M(y))$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 の何れかを終点とする）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lavel(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，

$$\cdots,v_1,u_1,\cdots,u_2,v_2,\cdots,v_3,u_3,\cdots,u_4,v_4,$$
$$\cdots,v_5,u_5,\cdots,u_6,v_6,\cdots,v_7,u_7,\cdots,u_8,v_8\ \cdots$$

ならびに，

$$\cdots,v'_1,u_3,\cdots,u'_2,v_8,\cdots,v'_3,u_5,\cdots,u'_4,v_2,$$
$$\cdots,v'_5,u_1,\cdots,u'_6,v_6,\cdots,v'_7,u_7,\cdots,u'_8,v_6\ \cdots$$

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の下半分を連接した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．

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

（図２(c)参照）□

【補題３．２の証明の続き】 明らかに，$|V(m)|=(2m)^m$ である．一方，各$m\geq 1$に対して，

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

とおけば，

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

が成り立つ．ここに，$e[m]=s(2m+2)L(2m)t^{L(2m)}$，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．

```
                    (ack() | point(x+dx+0.0, y+dy+0.0) self);
```

*val point = proc: real*real->'a::{getx:(real->o)->o, gety:(real->o)->o, move:real*real*(unit->o)->o}->o*

We can define a `rectangle` process by composing two `point` processes as follows.

```
local
  proc rectangle'(p1, p2) self =
     #getcx self (reply) =>
          ($m.(#getx p1 m | m(x1)=> (#getx p2 m | m(x2)=>
               (reply((x1+x2)/2.0) | rectangle'(p1, p2) self))))
   & #getcy self (reply) =>
          ($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:

```
proc new_rectangle (x1, y1, x2, y2) self =
         rectangle(point(x1, y1), point(x2, y2)) self;
```

*val new_rectangle = proc: real*real*real*real->'a::{getcx:(real->o)->o, getcy:(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`:

```
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 `lpoint` 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 ()=> (lfork() | rfork()
    (* repeat the same behavior *)
                  | philosopher n (lfork, rfork)))));
in
  proc philosopher_and_fork n (lfork, rfork) =
                                lfork() | philospher 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)=> ...
& #getx 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:

`catch` m `in` p `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[(`catch` m `in` p `end`)/x] can be considered an abbreviated form of the following expression.

```
$m.(p | m(x)=>e)
```

The following function `pfib` computes the fibonacci number in parallel.

```
fun fib(n) =
    if n=0 then 1 else if n=1 then 1
    else fib(n-1)+fib(n-2);
fun pfib(n) =
    if n<5 then fib(n)
    else catch result
         in
           $m1.$m2.(m1(pfib(n-2)) | m2(pfib(n-1))
                    | m1(x) => m2(y) => result (x+y))
         end;
```

If `n` is less than 5, `pfib(n)` computes the fibonnaci 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 π-calculus. We believe that concurrent linear logic programming potentially offers a more fruitful programming model than π-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 enhances 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

[1] Girard, J.-Y., "Linear Logic," *Theoretical Computer Science*, vol. 50, pp. 1–102, 1987.

[2] Kobayashi, N., and A. Yonezawa, "Asynchronous Communication Model Based on Linear Logic." to appear in Journal of Formal Aspects of Computing, Springer-Verlag.

[3] Kobayashi, N., and A. Yonezawa, "ACL – A Concurrent Linear Logic Programming Paradigm," in *Logic Programming: Proceedings of the 1993 International Symposium*, pp. 279–294, MIT Press, 1993.

[4] Kobayashi, N., and A. Yonezawa, "Type-Theoretic Foundations for Concurrent Object-Oriented Programming," in *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94), to appear*, 1994.

[5] Kobayashi, N., and A. Yonezawa, "Typed Higher-Order Concurrent Linear Logic Programming," Tech. Rep. 94-12, Department of Information Science, University of Tokyo, 1994. to be presented at Theory and Practice of Parallel Programming (TPPP'94), Sendai, Japan.

[6] Ohori, A., "A Compilation Method for ML-Style Polymorphic Record Calculi," in *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 154–165, 1992.

[7] Pierce, B. C., "Programming in the Pi-Calculus: An Experiment in Programming Language Design." Lecture notes for a course at the LFCS, University of Edinburgh., 1993.

[8] Troelstra, A. S., "Tutorial on Linear Logic," 1992. Tutorial notes in JICSLP'92.

[9] Yonezawa, A., *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[10] Yonezawa, A., and M. Tokoro, *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

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 $A \supset (B \wedge C)$ 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 $\parr$ is respectively de Morgan dual of & and $\otimes$:

$$(A\&B)^{\perp} \equiv A^{\perp} \oplus B^{\perp}$$
$$(A \otimes B)^{\perp} \equiv A^{\perp} \parr B^{\perp}$$

where $(\bullet)^{\perp}$ is a linear negation. $A \oplus B$ means that "at least one of A and B holds." (Compare with $A\&B$.) $A \parr B$ can be also defined as $A^{\perp} \multimap B \equiv B^{\perp} \multimap A$. $\mathbf{1}$ 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:

$$A \otimes (B \oplus C) \equiv (A \otimes B) \oplus (A \otimes C)$$
$$A\&(B \parr C) \equiv (A\&B) \parr (A\&C)$$

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 $m \multimap B$. Therefore, a process is also represented by a formula of linear logic. Consumption of a message m by a process $m \multimap B$ is represented by the following deduction:

$$m \otimes (m \multimap B) \otimes C \multimap B \otimes C$$

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

$$m \wedge (m \supset B) \supset B$$

but we can also deduce

$$m \wedge (m \supset B) \supset m \wedge (m \supset B) \wedge B$$

which implies that the original message m and process $m \supset B$ 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

[3]This 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.

$(m_1 \multimap A_1)\&(m_2 \multimap A_2)$ means that we have any one of $m_1 \multimap A_1$ and $m_2 \multimap A_2$, 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 $(m_1 \multimap A_1) \otimes (m_2 \multimap A_2)$):

$$m_1 \otimes ((m_1 \multimap A_1)\&(m_2 \multimap A_2)) \otimes C \multimap A_1 \otimes C$$
$$m_2 \otimes ((m_1 \multimap A_1)\&(m_2 \multimap A_2)) \otimes C \multimap A_2 \otimes C$$

Therefore, $(m_1 \multimap A_1)\&(m_2 \multimap A_2)$ 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 $\forall x.(m(x) \multimap A(x))$ 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, $\forall x.(m(x) \multimap A(x))$ represents a process which receives a value of x via a message m, and becomes $A(x)$:

$$m(a) \otimes \forall x.(m(x) \multimap A(x)) \otimes C \multimap A(a) \otimes C$$

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: $\forall x.(m(a,x) \multimap A(x))$, it receives only a message m whose first argument matches to a, hence we can realize Linda-like generative communication. $\forall x.(m(x,x) \multimap A(x))$ receives only a message m whose first and second arguments match. A formula $\forall x.\forall y.(m(x) \otimes n(y) \multimap A(x,y))$ receives a value x via m, and y via n, and behaves like $A(x,y)$. Since the formula is equivalent to

$$\forall x.(m(x) \multimap \forall y.(n(y) \multimap A(x,y))) \equiv \forall y.(n(y) \multimap \forall x.(m(x) \multimap A(x,y))),$$

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 λ-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
$\mathbf{1}$	$\bot$	inaction
m (atomic formula)	m	message
$A \otimes B$	$A \parr B$	concurrent composition
$\forall x.(m(x) \multimap A(x))$	$\exists x.(m(x)^{\bot} \otimes A(x))$	message reception
$R_1 \& R_2$	$R_1 \oplus R_2$	selective message reception
$\exists x.A$	$\forall x.A$	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. $int \rightarrow o$ is a type of predicate on integers. Computatinally, it is a type of processes or messages which take an integer as an argument. $(int \rightarrow o) \rightarrow o$ is a type of processes or messages which take a process or a message that takes an integer as an argument. $(int \rightarrow o) \rightarrow int$ 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 $!(p(x) \multimap e1)$, (or $!(p(x) \circ\!\!- e1)$ by the dual encoding), which corresponds to clause in traditional logic programming. However, we prefered to interpret it as **let** $p = \mathbf{fix}(\lambda p.\lambda x.e1)$ **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.