TECHNICAL REPORT 94-12

Typed Higher-Order Concurrent Linear Logic Programming

Naoki Kobayashi and Akinori Yonezawa

July 1994

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TECHNICAL REPORT 94-12

TITLE
Typed Higher-Order Concurrent Linear Logic Programming

AUTHORS
Set the author name(s) with "`Author`" macro.

KEY WORDS AND PHRASES
concurrent linear logic programming, higher-order concurrent programming, types, asynchronous concurrent computation

ABSTRACT
We propose a typed, higher-order, concurrent linear logic programming called *higher-order ACL*, which uniformly integrates a variety of mechanisms for concurrent computation based on asynchronous message passing. Higher-order ACL is based on a proof search paradigm according to the principle, *proofs as computations*, *formulas as processes* in linear logic. In higher-order ACL, processes as well as functions, and other values can be communicated via messages, which provides high modularity of concurrent programs. Higher-order ACL can be viewed as an asynchronous counterpart of Milner's higher-order, polyadic π-calculus. Moreover, higher-order ACL is equipped with an elegant ML-style type system that ensures (1) well typed programs can never cause type mismatch errors, and (2) there is a type inference algorithm which computes a most general typing for an untyped term. We also demonstrate a power of higher-order ACL by showing several examples of "higher-order concurrent programming."

REPORT DATE	**WRITTEN LANGUAGE**
July 1994	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
20	21

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
To be presented at Theory and Practice of Parallel Programming (TPPP), Sendai, Japan

DISTRIBUTION STATEMENT
First issue copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

Typed Higher-Order Concurrent Linear Logic Programming

Naoki KOBAYASHI Akinori YONEZAWA
Department of Information Science
University of Tokyo
{koba, yonezawa }@is.s.u-tokyo.ac.jp

Abstract

We propose a typed, higher-order, concurrent linear logic programming called *higher-order ACL*, which uniformly integrates a variety of mechanisms for concurrent computation based on asynchronous message passing. Higher-order ACL is based on a proof search paradigm according to the principle, *proofs as computations*, *formulas as processes* in linear logic. In higher-order ACL, processes as well as functions, and other values can be communicated via messages, which provides high modularity of concurrent programs. Higher-order ACL can be viewed as an asynchronous counterpart of Milner's higher-order, polyadic π-calculus. Moreover, higher-order ACL is equipped with an elegant ML-style type system that ensures (1) well typed programs can never cause type mismatch errors, and (2) there is a type inference algorithm which computes a most general typing for an untyped term. We also demonstrate a power of higher-order ACL by showing several examples of "higher-order concurrent programming."

1 Introduction

In logic programming, computation is often regarded as a *controlled deduction*. By changing the underlying logic, we can obtain a variety of programming languages. Recently, linear logic programming, whose underlying logic is Girard's linear logic[5], has been drawing great attentions. Several linear logic languages[6] were proposed as refinements of traditional logic programming languages, while others were proposed as novel concurrent programming languages[1][9]. In the previous papers[9][8], we proposed an asynchronous concurrent linear logic language called ACL. While traditional concurrent logic programming models stream-based communication, concurrent linear logic programming naturally models message-passing style asynchronous communication. Further difference comes from the fact that in concurrent linear logic programming, messages as well as processes are represented in terms of propositions[9]. This gives rise to an elegant abstract interpretation technique for linear logic languages[2].

The purpose of this paper is to further extend a concurrent linear logic language ACL and give a more powerful and elegant framework called *Higher-Order ACL* (in short, HACL), for asynchronous concurrent programming languages. We extend ACL in the following two significant ways. First, we replace the first-order linear logic with *higher-order linear logic*. This enables processes to pass processes as well as functions and basic values as arguments of messages, just as functions can take functions as arguments in functional programming. More importantly, processes can be parameterized by other processes and

communication interfaces (i.e., messages), which provides a high modularity of concurrent programs and enhances code reuse. Higher-order processes are useful for defining a large process by composing smaller subprocesses. This mechanism contributes to *structuring* concurrent programs and exploiting *intra-process* concurrency, by which separating *abstraction* and *granularity* of concurrent programs. On the other hand, this separation does not exist in the first-order concurrent language including most of the existing concurrent object-oriented languages. Therefore, if a programmer wants to obtain more concurrency from his program, he must change abstraction of the program by dividing some processes into smaller processes, or completely restructuring the program. Another relevant advantage of higher-order processes in HACL is that we can easily make *topologies* between multiple processes by using higher-order processes. For example, we can define a higher-order process which takes a list of processes as arguments, and connects them in a ring structure or a mesh structure.[1]

Second, we introduce a natural and elegant ML-style static, polymorphic type system into Higher-Order ACL. Since computation in HACL can be viewed as a deduction in higher-order linear logic, the notion of *well-typed processes* in Higher-Order ACL corresponds to that of *well-formed formulas* of higher-order linear logic. For example, a process which take an integer as an argument is assigned a type $int \to o$, where o is a distinguished type of propositions. A *message predicate* (which is viewed as a *communication channel* or a *port name*) that takes an argument of type τ is assigned a type $\tau \to o$. Since each message predicate is consistently typed in its scope, no type mismatch error occurs upon message passing. An operator of parallel composition $\mathfrak{B}$ is assigned a type $o \to o \to o$. Such formulation of type systems is found in other higher-order logic programming languages such as λProlog[12]. The basic type system for HACL obtained in this manner is exactly the same as Damas-Milner type system[3] for ML. An important observation from this fact is that we can easily enrich the basic type system with extensions of ML-type system such as subtyping, recursive typing, polymorphic records, and bounded polymorphism, which have been well-investigated in the communities of functional/object-oriented/logic programming. As a specific application, we show in [10] that polymorphic record extensions of higher-order ACL can be a powerful basis for accounting for typed concurrent object-oriented programming with inheritance and method overriding. Therefore, our type system of HACL is highly useful not only in HACL itself, but also in other concurrent languages based on message-passing. Although type systems for concurrent calculi have been recently investigated by many researchers[4][7][14][19][20], we believe that our formulation is one of the most elegant and useful ones. We also give several simple refinements of the basic type system to get more precise information such as input/output modes of messages. The refined type system is useful for the detection of orphan messages, dead-code elimination, etc. Our type system ensures that no type mismatch error occurs, liberates programmers from complex type declarations, and also provides for useful compiling information. For example, type information need not be embedded in messages at runtime, which reduces both the size of messages and the cost of message handling.

In Section 2, we present the syntax and operational semantics of HACL. Section 3

[1] From our experience in programming using concurrent object-oriented language ABCL[21], we have long recognized that it is very cumbersome to make such topologies between concurrent objects using only *first-order* concurrent objects.

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

Figure 1: Inference corresponding to Message Reception

demonstrates the power of higher-order concurrent linear logic programming. In Section 4, we refine the type system given in Section 2. Section 5 compares related work, and Section 6 concludes this paper. In Appendix A, we give a brief introduction to concurrent linear logic programming.

2 Syntax and Operational Semantics of Higher-Order ACL

This section introduces the syntax, operational semantics and the basic type system of higher-order ACL.

2.1 Overview

As in the first-order ACL[9][8], computation in Higher-order ACL (in short, HACL) is described in terms of bottom-up *proof search* in Girard's (higher-order) linear logic[5]. 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 ranges 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 $P[v_1/x_1,\ldots,v_n/x_n]$.[2] For instance, `(m(1) | m(x) => n(x+1))` makes transitions as follows:

`(m(1) | m(x) => n(x+1))` $\longrightarrow$ `n(1+1)` $\longrightarrow$ `n(2)`

Message predicates (like `m` and `n` above), which construct messages, 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 `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;
```

defines a process *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`.

A process `counter`[9] is defined in HACL as follows:

[2] For other correspondence between process transitions and inference rules, please refer to [9][8].

Textual Notation	Linear Logic Formula	Description
`_`	$\bot$	inaction
$x(e_1, \ldots, e_n)$ `|` P	$x(e_1, \ldots, e_n) \parr P$	sends a message $x(e_1, \ldots, e_n)$, and then behaves like P
$x(x_1, \ldots x_n)$ `=>` P	$\exists x_1 \cdots \exists x_n.(x(x_1, \ldots, x_n)^\bot \otimes P)$	receives a message $x(e_1, \ldots, e_n)$, and then behaves like $P[e_1/x_1, \ldots, e_n/x_n]$
$R_1 \& R_2$	$R_1 \oplus R_2$	behaves like R_1 or R_2
$P_1 \| P_2$	$P_1 \parr P_2$	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 (A is a name of newly defined process)

Figure 2: Syntax of $HACL$ process expressions

```
proc counter n (inc, read, reset) =
    inc() => (counter (n+1) (inc, read, reset))
  & read(reply) => (reply n | counter n (inc, read, reset))
  & reset() => (counter 0 (inc, read, reset));
```

The process `counter` is parameterized by message predicates `inc`, `read`, and `reset`. Higher-Order ACL can be viewed as an asynchronous counterpart of Milner's higher-order, polyadic π-calculus. For those who are familiar with π-calculus, we give a rough correspondence between higher-order ACL and polyadic π-calculus in Appendix B.

2.2 Syntax and Operational Semantics

Now, we formally define the syntax and operational semantics of HACL.

First, we define *preterms* of HACL.

Definition 2.1 (preterms) A set of *preterms*, ranged over by e, is defined by the following syntax:

$$
\begin{array}{l}
e ::= x \mid c \mid e_1 e_2 \mid \lambda x.e \mid (e_1, e_2) \mid fst(e_1) \mid snd(e_2) \mid \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \\
\qquad \mid \textbf{fix}(\lambda x.e) \mid e_1 \parr e_2 \mid r \mid \forall x.e \mid ?e \\
r ::= \exists x_1 \cdots \exists x_n.(x(x_1, \ldots, x_n)^\bot \otimes e) \mid r_1 \oplus r_2
\end{array}
$$

where x stands for variables and c for constants including logical connectives $\top, \bot$ and basic values such as integers.

Not all preterms are valid terms in HACL. For example, a term `m(true) | m(x) => n(x+1)` (in the textual notation) transits to `n(true+1)`, which results in type error. In order to exclude such terms, we introduce a notion of types. Since each process in Higher-Order ACL corresponds to a formula of higher-order linear logic, the notion of *well-typed processes* corresponds to *well-formed formulas* of higher-order linear logic. In order to allow polymorphism, we introduce type schemes as follows.

Definition 2.2 (monotypes, type schemes) A set of *monotypes*, ranged over by τ, and a set of *type schemes*, ranged over by σ, are given by the following syntax:

$$
\begin{array}{l}
\tau ::= \alpha \mid b \mid o \mid \tau \to \tau \mid \tau \times \tau \\
\sigma ::= \tau \mid \forall \alpha.\sigma
\end{array}
$$

(const) $\mathcal{T} \vdash c : \tau$ if $\langle c, \sigma \rangle \in \mathbf{Const}$ and τ is a generic instance of σ

(var) $\mathcal{T} \vdash x : \tau$ if $x \in dom(\mathcal{T}), \mathcal{T}(x) = \tau$. **(abs)** $\dfrac{\mathcal{T}\{x \mapsto \tau_1\} \vdash e : \tau_2}{\mathcal{T} \vdash \lambda x.e : \tau_1 \to \tau_2}$

(app) $\dfrac{\mathcal{T} \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{T} \vdash e_1 e_2 : \tau_2}$ **(let)** $\dfrac{\mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{T} \vdash e_2[e_1/x] : \tau_2}{\mathcal{T} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}$

Figure 3: Typing Rules

(exists) $\dfrac{\mathcal{T}\{x \mapsto \tau\} \vdash e : o}{\mathcal{T} \vdash \exists x.e : o}$ **(forall)** $\dfrac{\mathcal{T}\{x \mapsto \tau \to o\} \vdash e : o}{\mathcal{T} \vdash \forall x.e : o}$

Figure 4: Derived Typing Rules

where α stands for type variables and b for base types, and o is a distinguished type for propositions (i.e., processes or messages).

For example, $int \to o$ is a type of predicates (i.e., processes or messages) which takes an integer as an argument. $\forall\alpha.\alpha \times \alpha \to o$ represents a set of all monotypes of the form $\tau \times \tau \to o$.

Typing rules for preterms are given in Figure 3. In the figure, $\mathcal{T}$ is a map from a finite set of variables to monotypes. **Const** is a set of pairs $\langle c, \sigma \rangle$, where c is a constant and σ is its type scheme. A monotype τ is called a generic instance of $\forall\alpha_1 \cdots \forall\alpha_n.\tau'$, if $\tau \equiv \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ for some $\tau_1, \ldots, \tau_n$. We assume **Const** contains at least the following constants:

$(\bullet, \bullet) : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \times \beta$
$fst : \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$, $snd : \forall\alpha.\forall\beta.\alpha \times \beta \to \beta$
$\parr, \otimes, \oplus : o \to o \to o$
$\top, \bot : o$
$(\bullet)^\bot, ? : o \to o$
$\Sigma : \forall\alpha.(\alpha \to o) \to o$
$\Pi : \forall\alpha.((\alpha \to o) \to o) \to o$

$\exists x.e$ and $\forall x.e$ is respectively regarded as an abbreviated form of $\Sigma(\lambda x.e)$ and $\Pi(\lambda x.e)$.[3] The resulting typing rules for $\exists$ and $\forall$ are given in Figure 4.

Please note that the typing rules are exactly the same as Damas-Milner type system. The following is well-known theorem on Damas-Milner type system.

Proposition 2.1 *There is an algorithm which, given a preterm e, computes a principal typing for e if it exists, or reports failure otherwise.*

Now, we consider only well-typed subset of preterms.

Definition 2.3 (terms) A preterm e is called a *term* (or expression) of type τ, if there is a derivation of $\mathcal{T} \vdash e : \tau$ for some $\mathcal{T}$. A term of type o is called a process expression.

[3] The reason we assign different types to Σ and Π is that we restrict that x in $\forall x.e$ must be a message predicate, hence must be of type $\tau \to o$ for some τ.

We often write p for a process expression.

We have implemented a type inference system for Higher-Order ACL. The followings are sample sessions of the HACL type inference system.

```
proc forwarder m n = m x => n x;
(* This type writer style is user's input *)
```

val forwarder = proc: ('a->o)->('a->o)->o
(This slanted style is system's output. *)*

```
proc buffer(in, out) =
    in(x, ack) => (ack() | $ack2.(out(x, ack2) | ack2() => buffer(in, out)));
```

val buffer = proc: ('a(unit->o)->o)*('a*(unit->o)->o)->o*

The inferred type for `forwarder` implies that it takes two message predicates (i.e., communication channels) as arguments, and that `m` and `n` takes an argument of the same type. A process `buffer(in, out)` receives a message `in(x, ack)`, and then sends an acknowledgement message `ack()` and a message `out(x, ack2)`. Then, the process waits for an acknowledgement message from a receiver of the `out` message, and then becomes the original process `buffer(in, out)`. The inferred type implies that `in` and `out` has the same type $\tau \times (unit \to o) \to o$ for some τ, where τ is a type of `x` and $unit \to o$ is a type of `ack`, `ack2`.

Now, we give an operational semantics of HACL in the style of structural operational semantics. The following *structural congruence* simplifies the presentation of transition rules.

Definition 2.4 (structural congruence: $\cong$) *Structural congruence* $\cong$ over expressions is the smallest reflexive and transitive congruence relation including the following relations. (The last relation corresponds to α-conversion rule in λ-calculus.)

1. $p \parr \bot \cong p$
2. $p_1 \parr p_2 \cong p_2 \parr p_1$
3. $(p_1 \parr p_2) \parr p_3 \cong p_1 \parr (p_2 \parr p_3)$
4. $p_1 \oplus p_2 \cong p_2 \oplus p_1$
5. $(p_1 \oplus p_2) \oplus p_3 \cong p_1 \oplus (p_2 \oplus p_3)$
6. $\exists x.(m^{\perp} \otimes p) \cong m^{\perp} \otimes \exists x.p$ if x is not free in m.
7. $\forall x.(p_1 \parr p_2) \cong p_1 \parr \forall x.p_2$ if x is not free in p_1.
8. $\forall x.p \cong p$ if x is not free in p.
9. $\forall x.p \cong \forall y.p[y/x]$ where y does not appear free in p.

Please notice that the above structural congruence is based on logical equivalence of formulas in linear logic.

Transition relation $\longrightarrow$ over expressions is defined in Figure 5. In the figure, v ranges over a set of *values*, which is a subset of terms and defined by the following syntax:

$\cong$: $\dfrac{p_1 \cong p_2 \quad p_1 \longrightarrow p_1' \quad p_1' \cong p_2'}{p_2 \longrightarrow p_2'}$ λ: $(\lambda x.e)v \longrightarrow e[v/x]$

App1: $\dfrac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}$ **App2:** $\dfrac{e \longrightarrow e'}{ve \longrightarrow ve'}$

Pair1: $\dfrac{e_1 \longrightarrow e_1'}{(e_1, e_2) \longrightarrow (e_1', e_2)}$ **Pair2:** $\dfrac{e \longrightarrow e'}{(v, e) \longrightarrow (v, e')}$

Let1: $\dfrac{e_1 \longrightarrow e_1'}{\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \longrightarrow \textbf{let } x = e_1' \textbf{ in } e_2 \textbf{ end}}$ **Let2:** $\textbf{let } x = v \textbf{ in } e \textbf{ end} \longrightarrow e[v/x]$

Fix: $\textbf{fix}\lambda x.e \longrightarrow e[(\textbf{fix}\lambda x.e)/x]$

Com: $(\exists x_1 \cdots \exists x_n.(x(x_1, \ldots, x_n)^{\perp} \otimes p) \parr x(v_1, \ldots, v_n) \longrightarrow p[v_1/x_1, \ldots, v_n/x_n]$

$\parr$: $\dfrac{p_1 \longrightarrow p_1'}{p_1 \parr p_2 \longrightarrow p_1' \parr p_2}$ $\oplus$: $\dfrac{p_1 \parr p_3 \longrightarrow p'}{(p_1 \oplus p_2) \parr p_3 \longrightarrow p'}$

$\forall$: $\dfrac{p \longrightarrow p'}{\forall x.p \longrightarrow \forall x.p'}$ **?:** $\dfrac{p_1 \parr p_2 \longrightarrow p'}{?p_1 \parr p_2 \longrightarrow ?p_1 \parr p'}$

$\top$: $\top \parr g \longrightarrow \top$

Figure 5: Transition rules for expressions

$$v ::= x \mid c \mid \lambda x.e \mid \forall x.e$$

where e ranges over terms. We write $\longrightarrow^*$ for a reflexive and transitive closure of $\longrightarrow$. Then, the following theorem can be proved in the same manner as in the first-order ACL[8].

Proposition 2.2 *Let p be a process expression. Then, $p \longrightarrow^* \top$ only if $\vdash p$ is provable in $HLL_{\lambda,let}$.*

Inference rules of higher-order linear logic $HLL_{\lambda,let}$ are given in Appendix C. The above proposition says that every process that can reach the terminal state ($\top$) is a valid formula of higher-order linear logic.[4]

3 Higher-Order Concurrent Programming in HACL

This section demonstrates the power of higher-order concurrent programming in HACL. In Higher-Order ACL, 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.

3.1 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 [9]), higher-order processes are very important in the sense that it makes a process

[4]The converse of the above proposition does not hold, partly because of call-by-value semantics of Higher-Order ACL, and partly because of the reason described in Section 4.

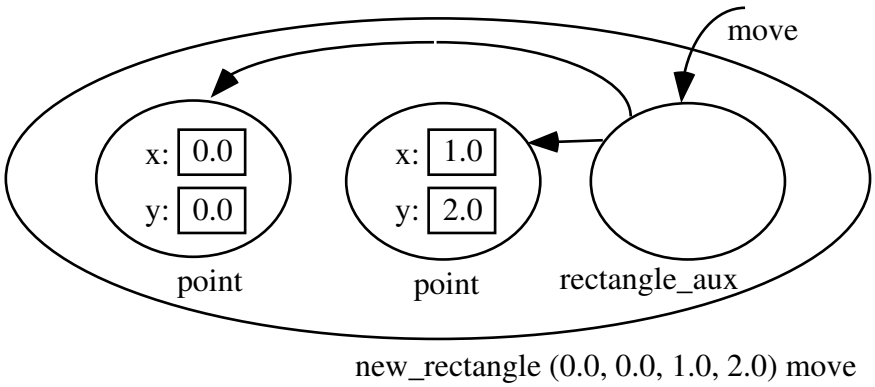

Figure 6: Rectangle Process

independent of a particular implementation of its subcomponent processes, by which enhancing modularity.

For example, consider the following `point` process:

```
proc point (x, y) move =
      move (dx, dy) => point (x+dx+0.0, y+dy+0.0) move;
```

*val point = proc: real*real->(real*real->o)->o*

For the simplicity, we assume that the `point` process handles only one message `move`. We can define a `rectangle` process by composing two `point` processes as follows.

```
let proc rectangle_aux move1 move2 move =
        move(dx, dy) => (move1(dx, dy) | move2(dx, dy)
                        | rectangle_aux move1 move2 move)
  in proc rectangle p1 p2 move =
           $m1.$m2.(p1 m1 | p2 m2 | rectangle_aux m1 m2 move);
```

val rectangle = proc: (('a'b->o)->o)->(('a*'b->o)->o)->('a*'b->o)->o*

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) move =
        rectangle (point(x1, y1)) (point(x2, y2)) move;
```

*val new_rectangle = proc: real*real*real*real->(real*real->o)->o*

A `rectangle` process is internally composed of two `point` processes and a new component `rectangle_aux` (see Figure 6). When a `move` message is sent to the `rectangle` process, the `rectangle_aux` process receives it, and forwards it to the two `point` processes. Important point about the `rectangle` process is that it is independent of a particular implementation of point processes. For example, let us define another point process:

```
proc point2 (r, theta) move =
        move dx dy => let x=r*cos(theta) and y=r*sin(theta) in
                      point2 (radius(x+dx, y+dy), angle(x+dx, y+dy)) move end;
```

*val point2 = proc: real*real->(real*real->o)->o*

Then, `rectangle` can take `point2` as an argument in place of `point`.

```
proc new_rectangle2 (x1, y1, x2, y2) move =
     rectangle (point2(radius(x1, y1), angle(x1, y1))) (point(x2, y2)) move;
```

*val new_rectangle = proc: real*real*real*real->(real*real->o)->o*

A `new_rectangle` process is composed of two point processes where one of them is `point2` and the other is `point`. Such independence of a particular implementation is very important in providing abstractions for concurrent programs. In the example of the rectangle process, we need not know how point processes are actually implemented; the point processes might be composed of multiple subprocesses or of a single process. This provides a separation between abstraction and granularity of processes. In the above example, if we require more parallelism, we can divide point processes into multiple processes. Such change of the component is transparent to the rectangle process. In contrast, in the first-order concurrent language, the implementation of rectangle process must be also modified, which violates the principle of *structured programming*.

3.2 Higher-Order Processes for Making Topologies between Processes

Higher-order processes are also 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. We can easily construct n-place buffer by applying a list of n buffer processes as an argument of `linear` process.

```
fun nlist(n, x) = if n=0 then nil else x::nlist(n-1, x);
```

val nlist = fn: int'a->'a list*

```
proc nbuffer n (in, out) = linear (nlist(n, buffer)) in out;
```

val nbuffer = proc: int->('a(unit->o)->o)*('a*(unit->o)->o)->o*

We can also construct a higher-order process for making a ring structure:

```
proc ring procs = $link.(linear procs link link);
```

val ring = proc: (('a->o)('a->o)->o) list->o*

The higher-order process `ring` can be constructed by instantiating `left` and `right` to the same message predicate (i.e. communication channel). In the same manner, we can also define higher-order processes for making any topologies between processes, such as mesh topology and torus topology.

4 Refinement of The Basic Type System

Since the basic type system of Higher-Order ACL presented in Section 2 is Damas-Milner's one, we can easily extend the basic type system with subtyping, recursive types, record types, etc., by which obtaining the more expressive power of Higher-Order ACL. In fact, we show in [10] that a typed concurrent object-oriented programming language with inheritance and method overriding can be easily constructed on top of the HACL extended with Ohori's polymorphic record calculus. This section introduces somewhat different extensions of the basic type system. We refine the type system in two ways, not for making the language more expressive, but for extracting computationally more useful information from programs.

4.1 Distinction between Messages and Processes

The first refinement is introduced for distinguishing between messages and processes. Since both messages and processes are represented by formulas, they are both assigned a type o in the basic type system. However, from the computational point of view, it is valuable to distinguish between them. For example, consider the expression `x(y) => y(z) => P` (`=>` is right associative), which receives a value of `y` via a message predicate `x`, then receives a value of `z` via `y`. Clearly, `y` should be a message predicate. However, the basic type system requires only that `y` be of type $\tau \to o$ for some τ. Therefore,

```
x(fn u => (p|q)) | x(y) => y(z) => r
```

(`fn x => e` stands for $\lambda x.e$) is well-typed (provided that `p`, `q`, and `r` are process expressions), although it transits to

```
(p|q) => r,
```

hence goes wrong![5]

In order to overcome this problem, we divide a type o into m, a type of messages, and p, a type of processes. m and p can be considered a kind of subtypes of o. To keep type inference tractable, instead of introducing a general form of subtyping, we consider the following refined type scheme.

τ (monotypes) $::= \alpha \mid b \mid m \mid p \mid \tau \times \tau \mid \tau \to \tau$
σ (type schemes) $::= \tau \mid \forall\alpha.\sigma \mid \forall\alpha :: o.\sigma$

"$\forall\alpha :: o.\sigma$" means that α in σ can be instantiated to either m or p. Types of constants are accordingly refined as follows:

$\parr, \otimes, \oplus : \forall\alpha :: o.\forall\beta :: o.\alpha \to \beta \to p$
$\top, \bot : p$
$(\bullet)^{\bot} : m \to p$
$? : \forall\alpha :: o.\alpha \to p$
$\Sigma : \forall\alpha.(\alpha \to p) \to p$
$\Pi : \forall\alpha.\forall\beta :: o\forall\gamma :: m.((\alpha \to \gamma) \to \beta) \to p$

Then, the existence of principal typing and type reconstruction algorithm is preserved. The following is a sample session in the refined type inference system.

[5] Although we can derive a type o for `(p|q)=>r` in the basic typing rules, it is not a preterm any more.

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

The refined type indicates that the first argument of a process `forwarder` must be a message predicate. The type of process `buffer` is refined as follows.

```
proc buffer(in, out) = ...;
```
val buffer = proc: ('a(unit->'b::o)->m)* ('a*(unit->m)->'c::o)->p*

4.2 I/O modes of messages

The basic type system does not convey any information on how message predicates are used in a particular process. For example, `m` and `n` of a process `forwarder m n` are assigned the same type, although `m` and `n` are used in a different way (i.e., `m` for input, and `n` for output). Such input/output information is useful for efficient implementation and also for statically detecting errors such as orphan messages (i.e., messages that have no receiver processes) and processes that wait for a never-returned message. For example, consider the following `worker` process:

```
proc worker job? = $reply.(job?(reply) | reply(job) => work(job));
```

A process `worker` asks for a job by `job?` message, and receives a job via a message `reply`. (We assume `work` is a previously defined process.) Obviously, `worker` process assumes that a receiver of the `job?` message should send a message `reply`. However, consider the following process `master`:

```
proc master job? = job?(reply) => _;
```

The `master` process receives a `job?` message, but just ignores it and produces no message. If we combine a `master` process and a `worker` process as follows:

```
$job?.(master job? | worker job?);
```

no type error is statically reported by the basic type system (and also by the extended type system in the previous subsection). However, it does go wrong in some sense at runtime, because the `worker` process never receives a `reply` message. It is better for a type system to statically detect such cases and produce some warning message. Although complete detections of such errors are difficult,[6] we can statically detect many of them by refining types for messages. We divide m, a type of messages, into the four types: m, m^-, m^+, $m^\pm$. A type m represents a type of messages that are used for neither input nor output. m^- (m^+, resp.) stands for a type of messages that are used only for input (output), and $m^\pm$ for both input and output. We accordingly refine the monotype and type scheme as follows:

$$\tau ::= \alpha \mid b \mid m \mid m^- \mid m^+ \mid m^\pm \mid p \mid \tau \times \tau \mid \tau \to \tau$$
$$\sigma ::= \tau \mid \forall\alpha.\sigma \mid \forall\alpha :: m.\sigma \mid \forall\alpha :: m^-.\sigma \mid \forall\alpha :: m^+.\sigma \mid \forall\alpha :: o.\sigma \mid \forall\alpha :: o^+.\sigma$$

A type scheme $\forall\alpha :: k.\sigma$ indicates that α can be instantiated to any τ such that $\tau \in S(k)$, where $S(k)$ is defined below.

[6] This is because flow of messages often depends not only on types but on values.

$S(m) = \{m, m^-, m^+, m^\pm\}, S(m^-) = \{m^-, m^\pm\}, S(m^+) = \{m^+, m^\pm\}$
$S(o) = \{p, m, m^-, m^+, m^\pm\}, S(o^+) = \{p, m^+, m^\pm\}$

The types of constants are refined as follows:

$\parr, \otimes, \oplus : \forall\alpha :: o^+.\forall\beta :: o^+.\alpha \to \beta \to p$
$\top, \bot : p$
$(\bullet)^\bot : \forall\alpha :: m^-.\alpha \to p$
$? : \forall\alpha :: o^+.\alpha \to p$
$\Sigma : \forall\alpha.(\alpha \to p) \to p$
$\Pi : \forall\alpha.\forall\beta :: o^+.((\alpha \to m) \to \beta) \to p$

The existence of principle typing and type reconstruction algorithm is again preserved. Subject reduction theorem is given as follows.

Proposition 4.1 (subject reduction) *If $\mathcal{T} \vdash e : \tau$ and $e \longrightarrow e'$, then $\mathcal{T} \vdash e' : \tau$.*

As a corollary of the subject reduction theorem, we obtain the following theorem.

Corollary 4.2 *If there is a derivation $x : \tau \to m^+ \vdash e : p$ for some τ, then x is never used for input in the evaluation of e.*

proof Suppose that x is used for input. Then, $e \longrightarrow^* (\exists x_1 \cdots \exists x_n.(x(x_1, \ldots, x_n)^\bot \otimes e_1)|e_2)$ for some e_1, e_2. However, from the type of $(\bullet)^\bot$, there is no derivation for

$$x : \tau \to m^+ \vdash \exists x_1 \cdots \exists x_n.(x(x_1, \ldots, x_n)^\bot \otimes e_1)|e_2) : o,$$

which contradicts the subject reduction theorem. □

Similarly, we obtain the following theorems.

Corollary 4.3 *If there is a derivation $x : \tau \to m^- \vdash e : p$ for some τ, then x is not used for output in the evaluation of e.*

Corollary 4.4 *If there is a derivation $x : \tau \to m \vdash e : p$ for some τ, then x is used for neither input nor output in the evaluation of e.*

Processes `worker` and `master` now have the following types:

`worker` : $\forall\alpha_1.\forall\alpha_2 :: m^-.\forall\alpha_3 :: o^+.((\alpha_1 \to \alpha_2) \to \alpha_3) \to p$
`master` : $\forall\alpha_1.\forall\alpha_2 :: m^-.(\alpha_1 \to \alpha_2) \to p$

Therefore, there is a derivation for

$$reply : \tau \to m^- \vdash e : p$$

where

$e \equiv$`$job?.(master job? | worker'(job?))`,
`worker'(job?)`$\equiv$`(job?(reply) | reply(job) => work(job))`.

Because `$reply.`$e \cong$ `$job?(master job? | worker job?)`, it is detected that no `reply` message is sent in the evaluation of `$job?(master job? | worker job?)`. Current type inference system incorporates this technique and produces warnings for orphan messages.

5 Discussion

Concurrent Logic Programming Although both concurrent logic programming[17] and HACL are both based on *computation as controlled deduction*, they are quite different concurrent programming paradigm, because of the difference of the underlying logic and representation of messages. Both paradigms have their advantage and disadvantages; concurrent logic programming better models stream-based communication, while HACL better models asynchronous message passing. However, type systems and inference for I/O modes on communication channels seem to be simpler and clearer in the context of HACL. We think it is because messages in HACL are represented at logic level.[7]

Type Inference v.s. Abstract Interpretation Andreoli and Pareschi[2] proposed an abstract interpretation technique for linear logic programming languages. Although the abstract interpretation technique has a potential power for obtaining much more information than our type inference, the abstract interpretation for a large program is likely to become intractable. On the other hand, type inference can be compositionally performed, hence it would be tractable in most cases. The refined type system in Section 4 allows static detections of orphan messages, dead-code eliminations, etc. Moreover, type information is also useful for abstract interpretation, because it gives information to decide which arguments can be dropped during abstract interpretation.

Typed Concurrent Calculi Recently, type systems for concurrent calculi such as polyadic π-calculus and object calculus, were proposed by several people[4][7][14][19][20]. Gay's sorting algorithm[4] can be considered a specific instance of the basic type system of HACL, where base types are restricted to only o and higher-order processes are dropped. Pierce and Sangiorgi[14] introduced I/O modes to π-calculus in a different manner from ours. They consider only explicitly-typed programs, and do not address the problem of type reconstruction. Thus, by viewing a process as a formula of linear logic, HACL clarifies the essence of those type systems. Since one of the purposes of type systems is to liberate programmers from the burden of writing complex type declarations and to aid them to understand concurrent programs, we believe that the simplicity is one of the most important criterion for type systems. Another relevant advantage is that we can naturally integrate other extensions of typed λ-calculus, such as subtyping, recursive types, polymorphic recursion, and polymorphic record calculus. The third advantage is that types of processes, functions and basic values are uniformly typed in our type system. Most of the previous type systems for concurrent calculi only deal with types of names. Although functions and basic values can be represented in name passing concurrent calculi, it is pragmatically important that functions and basic values can be directly written and typed.

λProlog Our approach to introducing type system is similar to that taken in λProlog[12]. The major difference is that in λProlog, types take an important role at runtime as well as at the type checking phase, while they do not in HACL. It is because the purpose of computation in HACL is not to find answer substutions, and HACL is designed so that higher-order unification is not required.

[7] The view "message as an atomic formula" is possible because of the *linearity* in linear logic.

Higher-order linear cc Saraswat and Lincoln[16] proposed higher-order linear cc, a kind of higher-order concurrent linear logic programming. In the untyped setting, the expressive power of their system and our system are almost the same; a message $x(v_1, \ldots, v_n)$ in higher-order ACL corresponds to $x : [v_1, \ldots, v_n]$ in higher-order linear cc. Their system, however, uses only a single binary predicate ":" to represent messages. Therefore, the difference between his system and our system is like that between π-calculus and polyadic π-calculus; In order to introduce a type system, our setting would be better.

Concurrent ML Concurrent ML[15] also allows higher-order concurrent programming. However, patterns of communication and process evolutions are more naturally expressed in HACL. Although effect analysis has recently been proposed to extract such patterns from Concurrent ML programs[13], it would be better for programmers to directly write such patterns. Moreover, communication in CML is synchronous, while it is asynchronous in HACL. Therefore, HACL can be directly used as a basis for concurrent object-oriented programming languages[10].

6 Conclusion

We proposed a higher-order concurrent linear logic programming called Higher-Order ACL (HACL). HACL naturally integrates typed functional programming and concurrent programming in terms of a proof search in higher-order linear logic. HACL is equipped with an elegant ML-style type system, which allows static detection of type errors and also polymorphic use of processes. We have also refined the basic type system to infer more useful information such as I/O modes of messages, which allows detection of errors such as orphan messages. Such type information is also useful for efficient implementation of message passing. As demonstrated in Section 3, higher-order processes in HACL provide a high modularity of concurrent programs.

We have already implemented a type inference system and interpreter of Higher-Order ACL with the refined type system in Section 4.[8] We have started to construct an efficient compiler system on multicomputers based on this work.

References

[1] Andreoli, J.-M., and R. Pareschi, "Communication as Fair Distribution of Knowledge," in *Proceedings of OOPSLA '91*, pp. 212–229, 1991.

[2] Andreoli, J.-M., R. Pareschi, and T. Castagnetti, "Abstract Interpretation of Linear Logic Programming," in *Proceedings of International Logic Programming Symposium*, pp. 315–334, 1993.

[3] Damas, L., and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 207–212, 1982.

[8] A prototype interpreter of HACL is available via anonymous ftp from `camille.is.s.u-tokyo.ac.jp: pub/hacl`

[4] Gay, S. J., "A Sort Inference Algorithm for the Polyadic π-calculus," in *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 429–438, 1993.

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

[6] Hodas, J., and D. Miller, "Logic programming in a fragment of intuitionistic linear logic," in *Proceedings of Sixth Annual Symposium on Logic in Computer Science*, pp. 32–42, 1991.

[7] Honda, K., "Types for Dydadic Interaction," in *Proceedings of CONCUR'93*, 1993.

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

[9] 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.

[10] 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.

[11] Milner, R., "The polyadic π-calculus: a tutorial," Tech. Rep. ECS-LFCS-91-180, University of Edinburgh, 1991.

[12] Nadathur, G., and F. Pfenning, "The Type System of a Higher-Order Logic Programming Language," in *Types in Logic Programming*, pp. 245–283, The MIT Press, 1992.

[13] Nielson, H. R., and F. Nielson, "Higher-Order Concurrent Programs with Finite Communicationn Topology," in *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 84–97, 1994.

[14] Pierce, B., and D. Sangiorgi, "Typing and Subtyping for Mobile Processes," in *Proceedings of IEEE Symposium on Logic in Computer Science*, pp. 376–385, 1993.

[15] Reppy, J. H., "CML: A Higher-order Concurrent Language," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 293–305, 1991.

[16] Saraswat, V., and P. Lincoln, "Higher-order, linear, concurrent constraint programming," July 1992. draft.

[17] Shapiro, E., "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 413–510, September 1989.

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

[19] Vasconcelos, V. T., and K. Honda, "Principal typing schemes in a polyadic π-calculus," in *Proceedings of CONCUR'93*, Lecture Notes in Computer Science, pp. 524–538, Springer Verlag, 1993.

[20] Vasconcelos, V. T., and M. Tokoro, "A Typing System for a Calculus of Objects," in *Proceedings of the International Symposium on Object Technologies for Advanced Software*, vol. 742 of *Lecture Notes in Computer Science*, pp. 460–474, Springer Verlag, 1993.

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

Appendix A 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[5] using the well-known examples.[18] 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 $\top$ is respectively unit of $\otimes$ and &. The above Girard's choice of symbols for connectives is 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?[9]

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 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

[9] This material gives a connection based on *proof search paradigm*. Regarding the alternative approach, *proof reduction paradigm*, please refer to other materials.

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 7: Connection between formula and process

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 7 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 is based on Girard's second-order linear logic[5] 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 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. $int \to o$ is a type of predicate on integers. Computatinally, it is a type of processes or messages which take an integer as an argument. $(int \to o) \to o$ is a type of processes or messages which take a process or a message that takes an integer as an argument. $(int \to o) \to int$ 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 = **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.

Appendix B Correspondence between Higher-Order ACL and Polyadic π-calculus

For those who are familiar with π-calculus, the following table summarizes a rough correspondence between higher-order ACL (in short, HACL) and Milner's polyadic π-calculus[11].

Polyadic π-calculus	Higher-Order ACL	HACL, Textual Representation
0	$\bot$	`_`
$P_1 + P_2$	$P_1 \oplus P_2$	`P1 & P2`
$\overline{y}\langle x_1, \ldots, x_n \rangle.P$[10]	$y(x_1, \ldots, x_n) \parr P$	`y(x1,...,xn) | P`
$y(x_1, \ldots, x_n).P$	$\exists x_1 \cdots \exists x_n.(y(x_1, \ldots, x_n)^\bot \otimes P)$	`y(x1,...,xn) => P`
$P_1 \mid P_2$	$P_1 \parr P_2$	`P1 | P2`
$(\nu x)P$	$\forall x.P$	`$x.P`
$!P$	$?P$	`?P`
$A(v_1, \ldots, v_n)$ where $A(x_1, ..., x_n) \stackrel{\text{def}}{=} P$	$(\mathbf{fix}\lambda A.\lambda x_1.\cdots\lambda x_n.P)v_1 \cdots v_n$	`let proc A x1 ... xn = P in A v1 ... vn end`

Appendix C Higher-Order Linear Logic

We give an inference rule of higher-order linear logic $HLL_{\lambda\mathbf{let}}$, following [16]. $\longrightarrow_{\lambda\mathbf{let}}$ represents a transition relation derived from rules λ, **App1**, **App2**, **Let1**, **Let2**, and **Fix** in Figure 1 in Section 2.

- Logical axioms

$$\vdash A, A^\bot$$

- $\longrightarrow_{\lambda\mathbf{let}}$

$$\frac{\vdash A, \Gamma \qquad B \longrightarrow_{\lambda\mathbf{let}} A}{\vdash B, \Gamma}$$

[10] Strictly speaking, this should be $\overline{y}\langle x_1, \ldots, x_n \rangle.0 \mid P$, since communication in π-calculus is synchronous.

- Cut rule

$$\frac{\vdash A, \Gamma \qquad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}$$

- Exchange rule

$$\frac{\vdash \Gamma}{\vdash \Delta}$$

  where Γ is a permutation of Δ.

- Additive rules

$$\vdash \top, A$$

$$\frac{\vdash A, \Gamma \qquad \vdash B, \Gamma}{\vdash A \,\&\, B, \Gamma}$$

$$\frac{\vdash A, \Gamma}{\vdash A \oplus B, \Gamma} \qquad \frac{\vdash B, \Gamma}{\vdash A \oplus B, \Gamma}$$

- Multiplicative rules

$$\vdash \mathbf{1} \qquad \frac{\vdash A}{\vdash \bot, A}$$

$$\frac{\vdash A, \Gamma \qquad \vdash B, \Delta}{\vdash A \otimes B, \Gamma, \Delta} \qquad \frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma}$$

- Exponential rules

$$\frac{\vdash A, \Gamma}{\vdash ?A, \Gamma} \text{ dereliction}$$

$$\frac{\vdash \Gamma}{\vdash ?A, \Gamma} \text{ weakening}$$

$$\frac{\vdash ?A, ?A, \Gamma}{\vdash ?A, \Gamma} \text{ contraction}$$

$$\frac{\vdash A, ?\Gamma}{\vdash !A, ?\Gamma}$$

- Quantifier rules

$$\frac{\vdash A, \Gamma}{\vdash \forall x.A, \Gamma} x \text{ not free in } \Gamma$$

$$\frac{\vdash A[B/x], \Gamma}{\vdash \exists x.A, \Gamma}$$