# Type-Theoretic Foundations for Concurrent Object-Oriented Programing

**Naoki Kobayashi and Akinori Yonezawa**

*Department of Information Science, University of Tokyo*
*7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan*

## Abstract

A number of attempts have been made to obtain type systems for object-oriented programming. The view that lies common is "*object-oriented programming = λ-calculus + record.*" Based on an analogous view "*concurrent object-oriented programming = concurrent calculus + record,*" we develop a static type system for concurrent object-oriented programming. We choose our own Higher-Order ACL as a basic concurrent calculus, and show that a concurrent object-oriented language can be easily encoded in the Higher-Order ACL extended with record operations. Since Higher-Order ACL has a strong type system with a polymorphic type inference mechanism, programs of the concurrent object-oriented language can be automatically type-checked by the encoding in Higher-Order ACL. Our approach can give clear accounts for complex mechanisms such as inheritance and method overriding within a simple framework.

## 1 Introduction

**Why Type System for Concurrent Object-Oriented Programming?** Significance of strong, static type systems has been widely recognized in the communities of functional programming and object-oriented programming. Static type systems increase reliability and readability of programs, enhance code reuse by polymorphic typing, and also provide useful compiling information. Moreover, static type systems with type inference mechanisms can liberate programmers from the burden of writing complex type declarations, hence programmers can benefit from such type systems without any additional efforts compared with dynamically typed programming languages. All the above arguments should also hold in *concurrent object-oriented programming.* We argue that the above mentioned benefits have much more significance in concurrent object-oriented programming than in functional programming, because runtime debugging of concurrent object-oriented programs seems difficult, especially on muticomputers (imagine the burden of debugging by tracing ten thousands of concurrent objects at the same time!).

**How Can We Obtain Typed Concurrent Object-Oriented Programming? – From a Philosophical Point of View** How can we systematically develop a flexible, powerful type system for *concurrent object-oriented programming*? Observation of the development of type systems[4][13] for sequential object-oriented programming is of great help. Many successful previous developments were based on the extension of *typed λ-calculus.* Although a number of powerful extensions of λ-calculus including $F_{\leq}$ and $F^{\omega}_{\leq}$ were investigated, the essence that lies common in those development is a view "typed object-oriented programming = typed λ-calculus + record." It is, therefore, natural to develop typed concurrent object-oriented programming based on a view "*typed concurrent object-oriented programming = typed concurrent calculus + record.*" This approach would provide a flexible and clear way for taking into account various mechanisms, including inheritance and method overriding, for typed concurrent object-oriented programming. Fortunately, work on typed concurrent calculi have recently made a great progress[?][9][11][16]; it is time to make a development in concurrent programming analogous to that[13][4] of typed object-oriented programming in sequential programming.

**Higher-Order ACL – A Typed Concurrent Calculus based on Linear Logic**[1] We use our Higher-Order ACL[9] (in short, HACL) as a basic concurrent calculus. HACL is a typed, higher-order extension of ACL[8][7] whose semantics is based on a fragment of linear logic[5]. HACL has an ML-style polymorphic type system. Thus there exists a simple unification-based type inference algorithm. In this paper, we extend HACL with Ohori's polymorphic record calculus[10] and show that a typed concurrent object-oriented programming with mechanisms for inheritance and method overriding can be easily encoded in the extended HACL. We can easily observe that a program is type error free if and only if the translated HACL program is type error free. In particular, there is a trivial embedding of Vasconcelos' calculus objects[17][15] into our HACL. Thus, HACL plays a significant role in developing typed concurrent object-oriented programming similar to those played by $\lambda$-calculus in developing typed sequential object-oriented programming.

**Inheritance and Method Overriding in Concurrent Object-Oriented Programming – From Type-Theoretic Point of View** Readers might fear that the introduction of inheritance and method overriding will require a more complex type system. In our approach, to incorporate inheritance and method overriding, *we need not modify the basic type system*; we have only to refine the *encoding* into HACL. Therefore, the introduction of inheritance does not impose any essential problems; inheritance is just a matter of programming and fully captured by our type system. This fact is analogous to Pierce's type system for object-oriented programming[13], where inheritance in functional object-oriented programming was treated as just a matter of programming and encoded in his basic object model. It is a stark contrast to Vasconcelos's object calculus[17]; his calculus does not provide such flexibility, hence the introduction of inheritance seems to require a major modification.

[1]In this paper, we do not assume readers' familiarity with linear logic.

**What Can We Benefit From Our Type System?** Our development of a type-theoretic foundation for concurrent object-oriented programming will provide the following benefits:

1. Since all type errors are detected statically, the burden of runtime debugging is highly reduced. This is a great advantage because it is very difficult to debug concurrent programs on massively parallel processors.

2. By a type inference mechanism, programmers are free from the burden of writing complex type declarations.

3. Code reuse is enhanced by polymorphism.

4. Some implementation techniques are naturally obtained by type inference. For example, in an efficient implementation[14] of a concurrent object-oriented language ABCL[19][18], a method lookup table plays an important role. We can naturally obtain entries of the table by type inference, even in the presence of inheritance. That enables method-lookup to be performed in a constant time.

5. Type information can be used for many kinds of optimizations. For example, we need not embed type information in messages, which reduces both the size of messages and the cost of message handling.

The rest of this paper is organized as follows. In section 2, we introduce an actor-based high-level language and a base language based on HACL. We show that the high-level language is easily encoded in the base language and type-checked. Section 3 extends the language to incorporate inheritance and accordingly refines the translation into Higher-Order ACL. Section 4 illustrates how method lookup table can be implemented using type information. Section 5 discusses related work and section 6 concludes this paper.

## 2 An Actor-based Concurrent Object-Oriented Language

In this section, we introduce two languages: a surface language $SL_1$ and a base language $BL_1$. $SL_1$ is an actor-based[1] simple concurrent object-oriented language. $BL_1$ is actually a subset of Higher-Order ACL extended with Ohori's polymorphic record calculus[10], into which the surface language is translated. We show a simple encoding of $SL_1$ in $BL_1$. Then, type inference is performed for $BL_1$, and a program written in $SL_1$ never causes type mismatch error if the translated program is well-typed. After that, we restrict the surface language so that an object may change its state but never changes its behavior. In the restricted language, we can ensure that well-typed programs never cause "message not understood" error.

### 2.1 The Surface Language $SL_1$

We first introduce an actor-based language $SL_1$. In the actor model, computations are performed by multiple agents called *actors*[1]. Each actor has its own mail address[2]. Upon receiving a message, an actor sends some messages to other actors or to itself, create new actors, and specify the replacement behavior.

For example, a class[3] *point* can be defined as follows:

```
(defclass (point x y)
  [getx (reply_adr)
     => (send_to reply_adr x)
        (become (point x y))
  [gety (reply_adr)
     => (send_to reply_adr y)
        (become (point x y))
  [set (newx, newy)
     => (become (point newx newy))])
```

An instance of class *point* has two state variables $x$ and $y$, and handles three messages $getx, gety$ and *set*. A message $getx$ ($gety$, resp.) takes an argument *reply_adr*, to which the *point* object sends the value of its state variable $x$ ($y$, resp.). A message *set* takes two arguments *newx* and *newy*. When the *point* object receives the message, it updates its instance variables to *newx* and *newy*.

An instance of class *point* can be created by the following expression:

**(let** id **:=new** point (1.0, 2.0) **in** e).

The above expression creates a new instance of class *point*, and binds *id* to its identifier in *e*.

The whole syntax of a class definition in $SL_1$ is given as follows:

```
⟨cl-def⟩ ::= (defclass ( ⟨cl-name⟩ ⟨state-var⟩* )
                ⟨script⟩*)
⟨script⟩ ::= [ ⟨m-name⟩ ( ⟨var⟩* ) => ⟨action⟩* ]
⟨action⟩::= (become ⟨cl-name⟩ ( ⟨arg⟩* ))
          | (send_to ⟨dest⟩ ⟨m-name⟩ ( ⟨arg⟩* ))
          | (send_to ⟨dest⟩ ⟨arg⟩ )
          | (let ⟨var⟩ := new ⟨cl-name⟩ ( ⟨arg⟩*)
                in ⟨action⟩*)
```

⟨script⟩ stands for a method definition. An expression `(send_to x m)` sends an asynchronous message `m` to `x`. A message `m` is either a pair consisting of a method name to be invoked and arguments, or an argument itself. ⟨argument⟩ can be a functional expression composed of bound variables, constants, built-in and user-defined functions, let-statements, and the following `synch_send_to` expression:

`(synch_send_to` ⟨dest⟩ ⟨m-name⟩ `(` ⟨arg⟩* `))`

This creates a new address $x$, and sends a message ⟨m-name⟩ ( ⟨arg⟩*, $x$). Then, the expression is evaluated to a value returned to the new address $x$. This expression is used together with let expressions in the following form:

[2]We often use a term *identifier* for *mail address*

[3]In this paper, we use a term *class* in place of *behavior*.

```
   (let ((y (synch_send_to x m))) ...).
```

For example,

```
(let id := new point(1.0, 2.0) in
 (let ((y (synch_send_to id getx())) ...)
```

is evaluated to:

```
(let ((y 1.0)) ...)
```

## 2.2 The Base Language $BL_1$

Now, we introduce the base language $BL_1$, into which programs in $SL_1$ are translated.

The syntax of $BL_1$ process expressions is summarized in Figure 1. Each process expression corresponds to some formula of linear logic. The second column shows this correspondence. Readers who are not familiar with linear logic can completely ignore this column. In the definition, $x$ ranges over variables, $P$ over process expressions, and $R$ over process expressions of the form $m(x_1, \ldots, x_n)$=>$P$. $m$ is a variable $x$ or a record field extraction $x.l$, where $l$ is a field name. In the terminology of HACL, we often call $m$ a *message predicate.*[4] Message passing can be performed if the message predicates of a sender and a receiver are the same. For example, `(m(1) | m(x) => n(x+1))` is evaluated as follows:

$$\texttt{(m(1) | m(x)=>n(x+1))} \longrightarrow \texttt{n(1+1)} \longrightarrow \texttt{n(2)}$$

Choice `(m1(x)=>n(x+1))&(m2(x)=>n(x))` can receive either a message `m1` or `m2`. Message predicates can be created by the `$` operator. `$`$x.P$ creates a message predicate or a record of message predicates and binds $x$ to it in $P$. Which of a message predicate or a record of message predicate is created is determined by the type inference system given later. For example, an expression `$id.(id.m(1) | id.m(x) => n(x+1))` creates a record `id` of message predicates which contains at least a field `m`, then performs message passing using a message predicate `id.m`. `Id` is hidden to the outside of the scope of `$id`. A `proc` statement defines a recursive process just as a `fun` statement defines a recursive function in ML.

[4]The term "predicate" comes from the fact that HACL is based on linear logic. Each message is represented by an atomic formula of linear logic, and $m$ corresponds to a predicate.

The whole syntax of expressions, ranged over by $e$, is defined by:

$$\begin{aligned} e ::= {} & x \mid c \mid e_1 e_2 \mid \lambda x.e \\ & \mid (e_1, e_2) \mid fst(e) \mid snd(e) \\ & \mid \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \mid \textbf{fix}\lambda x.e \\ & \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \mid e_1@\{l = e_2\} \\ & \mid (e_1|e_2) \mid \_ \mid \$x.e \mid ?e \\ & \mid (m_1(\vec{x_1})\texttt{=>}e_1)\&\cdots\&(m_n(\vec{x_n})\texttt{=>}e_n) \end{aligned}$$

where $\{l_1 = e_1, \ldots, l_n = e_n\}$ is a record whose value of $l_i$ field is $e_i$, $e.l$ is a record extraction, and $e_1@\{l = e_2\}$ is a non-destructive field update. On the sequential part, our language is consistent with Ohori's polymorphic record calculus[10].

## 2.3 Encoding $SL_1$ in $BL_1$

The surface language $SL_1$ can be easily encoded in $BL_1$. For example, the `point` class definition is encoded into the following process definition:

```
proc point (x, y) self =
  self.getx (reply_adr) =>
    (reply_adr(x) | point (x, y) self)
& self.gety (reply_adr) =>
    (reply_adr(y) | point (x, y) self)
& self.set (newx, newy) =>
    (point (newx, newy) self);
```

where we assume `=>` binds tighter than `&`. The translated program is almost the same as the original class definition except that an extra argument `self` to `point` is introduced. `self` can be considered an identifier of `point`, which was implicit in the surface language $SL_1$. In $BL_1$, it is implemented as a record whose field names are message names the object can handle.

An instance creation:

```
(let id := new point(1.0 2.0) in ...)
```

is encoded as

```
$id.(point (1.0, 2.0) id | ...).
```

Here, `$` is used as an operator for identifier creation.

The formal definition of the translation function from a class definition in $SL_1$ to a process definition in $BL_1$ is given in Figure 2. In the figure, $\epsilon$ stands

| Process Expression | Linear Logic Formula | Description |
|---|---|---|
| _ | $\bot$ | inaction |
| $m(e_1,\ldots,e_n)$ \| $P$ | $m(e_1,\ldots,e_n) \parr P$ | sends a message $m(e_1,\ldots,e_n)$, and then behaves like $P$ |
| $m(x_1,\ldots x_n)$ => $P$ | $\exists x_1 \cdots \exists x_n.(m(x_1,\ldots,x_n)^{\bot} \otimes P)$ | receives a message $m(e_1,\ldots,e_n)$, 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 1: Syntax of $BL_1$ process expressions

$\mathcal{F}_b$(**(defclass** (⟨cl-name⟩ ⟨state-var⟩*) ⟨script⟩*)) =**proc** ⟨cl-name⟩ (⟨state-var⟩*) **self** = $\mathcal{F}_s$(⟨scripts⟩);
$\mathcal{F}_s(\epsilon)$ =_
$\mathcal{F}_s$([⟨m-name⟩ (⟨var⟩* ) => ⟨action⟩*] ⟨script⟩*)=(**self**.⟨m-name⟩ ⟨var⟩* => $\mathcal{F}_a$(⟨actions⟩)) & $\mathcal{F}_s$(⟨script⟩*)
$\mathcal{F}_a(\epsilon)$ = _
$\mathcal{F}_a$(**(become** (⟨cl-name⟩ ⟨arg⟩*)) ⟨action⟩*) = ((⟨cl-name⟩ ⟨arg⟩* **self**) | $\mathcal{F}_a$(⟨action⟩*)
$\mathcal{F}_a$(**(let** $x$ := **new** ⟨cl-name⟩ (⟨arg⟩) **in** ⟨action⟩* )) =$\$x$.((⟨cl-name⟩ ⟨arg⟩* $x$) | $\mathcal{F}_a$(⟨action⟩*)
$\mathcal{F}_a$(**(send_to** ⟨dest⟩ ⟨m-name⟩ (⟨arg⟩*)) ⟨action⟩*) = ⟨dest⟩.⟨m-name⟩ (⟨arg⟩*) | $\mathcal{F}_a$(⟨action⟩*)
$\mathcal{F}_a$(**(send_to** ⟨dest⟩ ⟨arg⟩)) =⟨dest⟩ (⟨arg⟩) | $\mathcal{F}_a$(⟨action⟩*)

Figure 2: Translation from $SL_1$ into $BL_1$

for an empty sequence. We regard `P&_` $\equiv$ `P` in the translation. An expression using `synch_send_to`:

```
(let ((y (synch_send_to x m(e1,...,en))))
     e)
```

can be translated to:

```
$adr.(x.m(e1,...,en, adr) | adr(y) => e).
```

## 2.4 Type Inference

Type inference is performed on $BL_1$ programs. We say that a program of $SL_1$ is *well-typed* if the translated $BL_1$ program is *well-typed.*

We introduce a set of *kinds*[10], ranged over by $k$, as follows:

$$k ::= U \mid \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$$

where $\tau$ ranges over monotypes defined below. Intuitively, $U$ represents a set of all monotypes, while $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$ represents a set of record types of the form: $\{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\}$. Then, *monotypes* and *polytypes* are defined as follows:

**Definition 2.1 (monotypes, polytypes)** A set of *monotypes*, ranged over by $\tau$, and a set of *polytypes*, ranged over by $\sigma$, are given by the following syntax:

$$\begin{aligned} \tau &::= \alpha \mid b \mid o \mid \tau \to \tau \mid \tau \times \tau \\ &\quad\mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ \sigma &::= \tau \mid \forall\alpha :: k.\sigma \end{aligned}$$

where $\alpha$ stands for type variables, $b$ for base types, and $o$ for the type of messages and processes.

Intuitively, $\forall\alpha :: k$ implies that $\alpha$ can be instantiated to any monotype in $k$. For example, a polytype

$$\forall\alpha :: \langle l : \beta \rangle.\forall\beta :: U.\alpha$$

represents a set of all monotypes of the form $\{l : \tau, \ldots\}$, i.e., any record that contains a field $l$ of type $\tau$. We often just write "$\forall\alpha$" for "$\forall\alpha :: U$."

Rules for type judgement, which consist of *kinding rules* and *typing rules*, are given in Figure 3. A kinding statement is of the form $\mathcal{K} \vdash \tau :: k$ where a kind assignment $\mathcal{K}$ is a map from a finite set of type variables to kinds. It should be read as: "$\tau$ has a kind $k$ under a kind assignment $\mathcal{K}$." A typing statement is of the form $\mathcal{K}, \mathcal{T} \vdash e : \tau$ where a type assignment $\mathcal{T}$ is a map from a finite set of variables to types. It should be read: "$e$ has a type $\tau$ under a kind assignment $\mathcal{K}$ and a type assignment $\mathcal{T}$."

In the typing rules, **Const** is a set of pairs $\langle c, \sigma \rangle$ where $c$ is a constant and $\sigma$ is its polytype. We assume **Const** contains at least the following operators:

$$|, \& : o \rightarrow o \rightarrow o$$
$$? : o \rightarrow o$$
$$\textbf{fix}: \forall \alpha :: U.(\alpha \rightarrow \alpha) \rightarrow \alpha$$

A monotype $\tau$ is called a *generic instance*[10] of a polytype $\forall \alpha_1 :: k_1 \cdots \forall \alpha_n :: k_n.\tau'$ if $\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ for some monotypes $\tau_1, \ldots, \tau_n$ such that

$$\phi \vdash \tau_i :: k_i[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \text{ for all } i(1 \leq i \leq n)$$

*PropType*, ranged over by $\tau_p$, is a subset of monotypes defined by the following syntax:

$$\tau_p ::= o \mid \tau \rightarrow \tau_p \mid \{l_1 : \tau_{p_1}, \ldots, l_n : \tau_{p_n}\}$$

where $\tau$ ranges over the set of monotypes.

We say that an expression $e$ is *well-typed* if $\phi, \phi \vdash e : \tau$ for some $\tau$, where $\phi$ is a map whose domain is an empty set. The following theorem ensures that well-typed programs never cause type mismatch error.

**Proposition 2.1 (subject reduction)** *If $\mathcal{K}, \mathcal{T} \vdash e : \tau$ and $e \longrightarrow e'$, then $\mathcal{K}, \mathcal{T} \vdash e' : \tau$, where $\longrightarrow$ is a transition relation.*

Moreover, there exists a type inference algorithm which recovers the most general type for any well-typed term.

**Proposition 2.2** *There is an algorithm which, given any untyped term $e$, outputs its principal typing if it is well-typed, or reports failure otherwise.*

## Kinding Rules

$\mathcal{K} \vdash \tau :: U$ for all $\tau$

$\mathcal{K} \vdash t :: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$
if $t \in dom(\mathcal{K}), \mathcal{K}(t) = \langle l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots \rangle$

$\mathcal{K} \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} :: \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$

## Typing Rules

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

**(var)** $\mathcal{K}, \mathcal{T} \vdash x : \tau$ if $x \in dom(\mathcal{T}), \mathcal{T}(x) = \tau$.

**(abs)** $\dfrac{\mathcal{K}, \mathcal{T}\{x \mapsto \tau_1\} \vdash e : \tau_2}{\mathcal{K}, \mathcal{T} \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$

**(app)** $\dfrac{\mathcal{K}, \mathcal{T} \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \mathcal{K}, \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{K}, \mathcal{T} \vdash e_1 e_2 : \tau_2}$

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

**(=>)**

$$\dfrac{\begin{array}{c}\mathcal{K}, \mathcal{T} \vdash m : \tau_1 \times \cdots \times \tau_n \rightarrow o, \\ \mathcal{K}, \mathcal{T}\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : o\end{array}}{\mathcal{K}, \mathcal{T} \vdash m(x_1, \ldots, x_n) => e : o}$$

**(\$)** $\dfrac{\mathcal{K}, \mathcal{T}\{x \mapsto \tau\} \vdash e : o}{\mathcal{K}, \mathcal{T} \vdash \$x.e : o} \; \tau \in PropType$

**(record)**

$$\dfrac{\mathcal{K}, \mathcal{T} \vdash e_i : \tau_i (1 \leq i \leq n)}{\mathcal{K}, \mathcal{T} \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

**(field)** $\dfrac{\mathcal{K}, \mathcal{T} \vdash e : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \langle l : \tau_2 \rangle}{\mathcal{K}, \mathcal{T} \vdash e.l : \tau_2}$

**(modify)**

$$\dfrac{\mathcal{K}, \mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{K}, \mathcal{T} \vdash e_2 : \tau_2 \quad \mathcal{K} \vdash \tau_1 :: \langle l : \tau_2 \rangle}{\mathcal{K}, \mathcal{T} \vdash e_1 @\{l = e_2\} : \tau_1}$$

Figure 3: Typing Rules for $BL_1$

The proof is immediate from the result on Ohori's polymorphic record calculus[10].

We have already implemented a type inference system. The algorithm for type inference is a simple extension of Ohori's type inference algorithm for his polymorphic record calculus. For example, a type of the process *point* given in the previous subsection can be inferred as follows:

```
proc point (x, y) self =  ...    ;
(* Type writer style is user's input. *)
```

*val point = proc: 'a*'b->'c::{getx:('a->o)->o, gety:('b->o)->o, set:'a*'b->o}->o*
*(* Slanted style is system's output. *)*

The system's output says that `point` has a type:

$\forall \alpha.\forall \beta.\forall \gamma :: \langle getx : (\alpha \rightarrow o) \rightarrow o, gety : (\beta \rightarrow o) \rightarrow o, set : \alpha \times \beta \rightarrow o \rangle.\alpha \times \beta \rightarrow \gamma \rightarrow o.$

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 $\{getx : (\alpha \rightarrow o) \rightarrow o, gety : (\beta \rightarrow o) \rightarrow o, set : \alpha \times \beta \rightarrow o, \ldots\}$ as its second argument self. The type of self implies that a message set takes a pair of values of type $\alpha$ and $\beta$, while a message getx takes a message predicate which takes a value of type $\alpha$ as an argument. By combining with the translation of $SL_1$ into $BL_1$, $SL_1$ programs are statically type-checked.

*Well-typed* programs so far do NOT ensure that no "message not understood" error occurs. For example, the following expression, which creates an instance of class point and sends to it a message move,

```
(let id:= new point(1.0, 2.0) in
          (send_to id move(1.0, 1.0)))
```

is well-typed, because it is translated to:

```
$id.(point (1.0,2.0) id | id.move(1.0,1.0))
```
*val it = proc : o*

During type inference, a type of the form:
$\{getx : (real \rightarrow o) \rightarrow o, gety : (real \rightarrow o) \rightarrow o, set : real \times real \rightarrow o, move : real \times real \rightarrow o, \ldots\}$
is associated to id.

The fact that "message not understood" error cannot be detected is less problematic in the case of *asynchronous* message passing. We can implement concurrent objects so that undefined messages never raise runtime error; they are just enqueued forever and never processed. For practical purposes of ensuring program correctness, we would like to detect such errors. We will present a solution in the next subsection.

## 2.5 Detection of "Message Not Understood" Errors

Although "*message not understood*" error causes no problem in the implementation, one might want to ensure that no "*message not understood*" error occurs. It is better to statically detect it as an error, for the safety of concurrent programs. We can assure it by restricting the language so that each object may change its states but never changes its behavior; the replacement behavior of "become"[1] operation must be always what it was. An important point about this restriction is that all message names each object handles are completely known syntactically. Under this restriction, for example, we can translate point in the previous section into the following $BL_1$ program:

```
proc point(x,y) self:{getx:'a,gety:'b,set:'c}=
   self.getx(reply)=>
          (reply(x) | point(x, y) self)
 & self.gety(reply)=>
          (reply(y) | point(x, y) self)
 & self.set(newx, newy) =>
          (point(newx,newy) self);
```

*val point = proc: 'a*'b->{getx:('a->o)->o, gety:('b->o)->o, set:'a*'b->o}->o*

The only difference with the translation in the previous subsection is that a type specification for an argument self is explicitly attached. This specification says that self is a record consisting of just three fields getx, gety and set. Please notice that this specification can be *automatically* generated by the translation function; the system can syntactically find that point handles three methods getx, gety and set. Under this refined translation, no "*message not understood*" error occurs if the translated program is well-typed, because an identifier of each object is a record whose field names are just the message names it can receive. In order to send a message, a sender must extract a corresponding field from the identifier of the receiver. Therefore, the sender can send only messages the receiver can handle. For example, suppose that some object sends a message move to an instance of class point:

```
(let id := new point(1.0, 2.0) in
      (send_to id move(1.0, 1.0)))
```

This expression is translated into the following $BL_1$ program:

```
$id.(point(1.0,2.0) id | id.move(1.0,1.0));
```
*Type Mismatch Error on variable id*

Since point has a type

$$\forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \langle getx : (\alpha \rightarrow o) \rightarrow o, gety : (\beta \rightarrow o) \rightarrow o, set : \alpha \times \beta \rightarrow o\rangle \rightarrow o,$$

the expression point (1.0, 2.0) id in the above program requires id to be of type $\{getx$ :

$(real \rightarrow o) \rightarrow o, gety : (real \rightarrow o) \rightarrow o, set : real \times real \rightarrow o\}$. Therefore, the field selection `id.move` violates the type of `id`. Thus, all "*message not understood*" errors are statically detected by the type inference system; that is, well-typed programs cannot cause "*message not understood*" error.

Readers might think it is too restrictive that concurrent objects must not change its behavior. It is very easy to relax this condition so that objects can change its behavior as far as it is capable of handling the same set of messages, because the detection of "message not understood" errors is realized by only attaching the explicit type declaration to **self**. For example, consider the following one-place buffer object:

```
(defclass (ebuffer)
  [put (x reply) => (send_to reply ())
                    (become (fbuffer x))])
(defclass (fbuffer x)
  [get (reply) => (send_to reply x)
                  (become (ebuffer))])
```

It can be defined in $BL_1$ by:

```
proc ebuffer self:{get:'a, put:'b} =
   self.put(x, reply) =>
         (reply() | fbuffer (x) self)
and fbuffer x self =
   self.get(reply) =>
         (reply(x) | ebuffer self)
```

By attaching the type constraints
`{get:'a, put:'b}` to **self**, we can ensure that messages except for **get** and **put** are never sent to the one-place buffer object.

## 3 Inheritance

In this section, we extend the language $SL_1$ in the previous section and accordingly refine the translation step by step, to incorporate an inheritance mechanism.

### 3.1 Simple Inheritance

In this subsection, we introduce a simple inheritance mechanism. Other mechanisms associated with inheritance, such as *self* and *super* variables and method overriding, will be treated in the subsequent subsections.

First, we extend the surface language $SL_1$ so that superclass, from which its subclass inherit methods and state variables (or, instance variables in the terminology of Smalltalk), can be specified in the class definition. We assume that the superclass must be previously defined.

Before showing the encoding of inheritance, we also extend the base language $BL_1$ by introducing the following record coercion operator:

$$coerce\{l_1, \ldots, l_n\}\{l_1, \ldots, l_m\} : \forall\alpha_1 \cdots \forall\alpha_n.(\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \rightarrow \{l_1 : \alpha_1, \ldots, l_m : \alpha_m\}) \text{ where } m \leq n$$

By inheritance, we assume that a process (i.e., instance) of some subclass handles newly defined methods and those its superclass handles. We also assume that instance variables in a superclass is hidden to its subclasses; the instance of a subclass can access instance variables of its superclass only via message passing. We demonstrate the encoding of this kind of inheritance by using the class `point` introduced in the previous section.

Let us consider a colored point object, which has an instance variable c holding a color number as well as $x$- and $y$-coordinates **x** and **y**. A class `cpoint` of colored point objects can be defined as a subclass of `point`:

```
(defclass (cpoint c :inherit point)
  [setc(newc) => (become (cpoint newc))]
  [getc(reply) => (send_to reply c)
                  (become (cpoint c))])
```

`point` is declared as a superclass by using a keyword `:inherit`. An instance of class `cpoint` has five methods; three methods **set**, **getx**, and **gety** are inherited from the superclass, and two methods **setc** and **getc** are newly defined. An instance of `cpoint` whose color number is 1 and coordinate is (1.0, 2.0), is created by the following expression:

```
(let id := new cpoint(1,point(1.0,2.0))
 in ...)
```

The above definition is translated into the following HACL process definitions:

```
proc cpoint' c self =
  self.setc newc => cpoint' newc self
& self.getc reply => (reply c | cpoint' c self);
```

*val cpoint' = proc: 'a->'b::{setc:'a->o, getc:('a->o)->o}->o*

```
proc cpoint c p self:{set:'a,getx:'b,
             gety:'c,setc:'d,getc:'e}=
  let super=
     coerce{set, getx, gety, setc, getc}
           {set, getx, gety}(self)
  in (p super | cpoint' c self) end;
```

*val cpoint = proc: 'a->({set:'b, getx:'c, gety:'d}->o)->{set:'b, getx:'c, gety:'d, setc:'a->o, getc:('a->o)->o}->o*

An instance creation is encoded as:

```
$id.(new_cpoint (1, 1.0, 2.0) id | ...)
```

where `new_cpoint` is defined by:

```
proc new_cpoint (c, x, y) self =
      cpoint c (point (x, y)) self;
```

*val new_cpoint = proc: 'a*'b*'c -> {set:'b*'c->o, getx:('b->o)->o, gety:('c->o)->o, setc:'a->o, getc:('a->o)->o}->o*

An instance of subclass is encoded as a concurrent composition of an instance of its superclass and newly added component. Figure 4 illustrates the structure of a `cpoint` object. The object is internally composed of two subobjects, namely, an object of superclass `point` and a newly added component `cpoint'`. Please notice that the operator `coerce` can be automatically inserted from the types of `point` and `self`.

Let us look at another example of inheritance. We add a method `move` to the `point` class.

```
(defclass (point2 :inherit point)
  [move(dx, dy) =>
   (let ((x (synch_send_to self getx()))
         (y (synch_send_to self gety())))
        (send_to self set(x+dx, y+dy))
        (become (point2)))])
```

In the method body of `move`, messages `getx` and `gety` are sent to `self`. `self` is a pseudo variable used as an address of the called object itself, as in other object-oriented languages (e.g. Smalltalk). In this subsection, since we do not consider method overriding, the above definition is translated as follows:

```
proc point2' self =
  self.move (dx, dy) =>
    $adr1.(self.getx adr1 |
      adr1 x => $adr2.(self.gety adr2 |
        adr2 y => (self.set(x+dx+0.0,y+dy+0.0)
                            | point2' self)));
```

*val point2' = proc: 'a::{move:real*real->o, getx:(real->o)->o, gety:(real->o)->o, set:real*real->o}->o*

```
proc point2 p self:{set:'a,getx:'b,
                     gety:'c,move:'d}=
  let super=
    coerce{set, getx, gety, move}
          {set, getx, gety}(self)
  in (point2' self | p super) end;
```

*val point2 = proc: ({getx:(real->o)->o, gety:(real->o)->o,
set:real*real->o}->o)->{move:real*real->o, getx:(real->o)->o, gety:(real->o)->o, set:real*real->o}->o*

```
proc new_point2 (x, y) self =
          point2 (point (x, y)) self;
```

*val point2 = proc: 'a*'b -> {move:real*real->o, getx:(real->o)->o, gety:(real->o)->o, set:real*real->o}->o*

An instance of `point2` is a concurrent composition of `point` and `point2'`. Please note that in the presence of method overriding, the above encoding of `self` would be incorrect.

Although we do not give a formal definition of the translation function, it is obvious from the above examples that we can encode the language with an inheritance mechanism into the base language $BL_1$.

## 3.2 Self and Method Overriding

Up to now, method overriding has been forbidden, that is, we could not redefine a method of its superclass. In order to allow method overriding, we must refine the encoding of a pseudo variable `self`. In usual object-oriented languages like Smalltalk, variable `self` in the class definition does not necessarily refer to the identifier (which is a record of message predicates in HACL) of that class itself, but may refer to an identifier of its subclass that has not yet been defined. Therefore, we need to

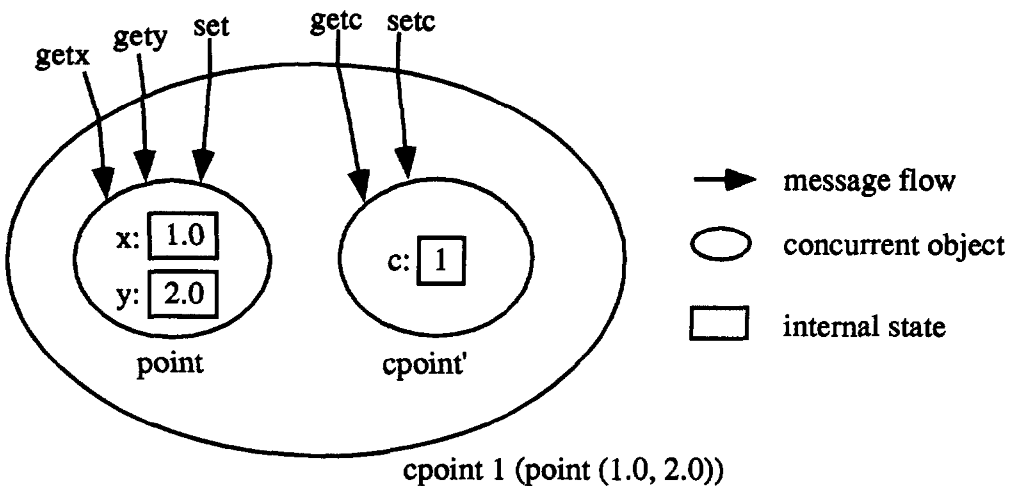

Figure 4: Conceptual structure of a cpoint object

distinguish `self` and the identifier of the defined class. We use a variable `methods` for the latter.

The idea of the refined encoding is to pass both `methods` and `self` to a class as a parameter. `methods` is used to refer to the method bodies of that class, while `self` is used in the method bodies to refer to the called object, which may be a subclass of the class currently executing the method. The encoding in the previous subsection is refined as follows:

```
proc point (x,y) methods:{set:'a,
               getx:'b, gety:'c} self=
  methods.set (newx, newy) =>
      (point (newx, newy) methods self)
& methods.getx reply =>
    (reply x | point (x, y) methods self)
& methods.gety reply =>
    (reply y | point (x, y) methods self);
```

*val point = proc: 'a*'b->{set:'a*'b->o, getx:('a->o)->o, gety:('a->o)->o}->'c->o*

```
proc point2' methods self =
  methods.move(dx, dy) =>
   $adr1.(self.getx adr1 |
     adr1(x)=> $adr2.(self.gety adr2 |
      adr2(y)=>(self.set (x+dx+0.0, y+dy+0.0)
                          | point2' self)));
```

*val point2' = proc: 'a::{move:real*real->o}->'b::{getx:(real->o)->o, gety:(real->o)->o, set:real*real->o}->o*

```
proc point2 point methods:{set:'a,
     getx:'b, gety:'c, move:'d} self =
  let super=
    coerce{set, getx, gety, move}
         {set, getx, gety}(methods)
  in (point super self | point2' methods self);
```

*val point2 = proc: ({getx:'a gety:'b, set:'c}->'d::{getx:(real->o)->o, gety:(real->o)->o, set:real*real->o} ->o)->{move:real*real->o, getx:'a, gety:'b, set:'c}->'d->o*

In the above, an additional argument `self` is attached to each process. Since `point` can handle only messages `set`, `getx` and `gety`, `methods` is coerced when applied to `point`. An instance is created by binding both `methods` and `self` to a new identifier:

```
proc new_point2 (x, y) self =
    point2 (point (x, y)) self self;
```

*val new_point2 = proc: real*real -> {set:real*real->o, getx:(real->o)->o, gety:(real->o)->o, move:real*real->o}->o*

```
$id.(new_point2 (1.0, 2.0) id | ...);
```

Now, let us consider method overriding. We define a new class `point3` by redefining methods `getx` and `gety` of `point2`:

```
(defclass (point3 :inherit point2)
  [getx(reply) => (send_to super gety(reply))
                  (become (point3))]
  [gety(reply) => (send_to super getx(reply))
                  (become (point3))])
```

A variable super is used to invoke a method of the superclass. This is encoded in HACL as follows:

```
proc point3' methods super =
  methods.getx reply =>
    (super.gety reply | point3' methods super)
& methods.gety reply =>
    (super.getx reply | point3' methods super);
```

*val point3' = proc: 'a::{getx:'b->o, gety:'c->o}-> 'd::{getx:'c->o, gety:'b->o}->o*

```
proc point3 p2 methods:{set:'a, getx:'b,
                        gety:'c, move:'d} self =
 $super_getx.$super_gety.(
  let super={getx=sup_getx,gety=sup_gety}
            \/coerce{set, getx, gety, move}
                    {set, move}(methods)
  in (point3' methods super
      | p2 super self) end;
```

*val point3 = proc: ({set:'a, getx:'b->o, gety:'c->o, move:'d}->'e->o) ->{set:'a, getx:'c->o, gety:'b->o, move:'d}->'e->o*

```
proc new_point3 (x, y) self =
  point3 (point2 (point (x, y))) self self;
```

*val new_point3 = proc: real*real-> {set:real*real->o, getx:(real->o)->o, gety:(real->o)->o, move:real*real->o}->o*

An operator \/ represents a record concatenation. Since methods getx and gety are overridden, different message predicates are created and assigned to fields getx and gety of a record super. If a message move is sent to an instance of point3 for example, a method dispatch goes as follows: First, the method defined in point2 is invoked. Then, during an execution of the method body, a message getx is sent to self. Since self refers to an instance of point3, the method getx defined in point3 is invoked, which in turn causes the method gety defined in point to be invoked.

## 4 Implementation

This section discusses an implementation technique for the language derived from type inference. We concentrate on an implementation of message passing mechanism. In particular, we can make a method dispatch table so that method lookup is performed in a constant time. ABCL/onAP1000[14][19] incorporated a similar mechanism and achieved high-performance message passing on multicomputers. Although our language is more complicated because of inheritance and method overriding, our type system allows a satisfactorily efficient implementation of message passing.

First, a message predicate $m$ can be represented as a pointer to the message handler specialized for $m$. Then, sending a message $m(v_1, \ldots, v_n)$ just causes the corresponding message handler to be invoked with arguments $v_1, \ldots, v_n$. (Notice that within the same processor node, this is almost the same as a local function invocation.) In the previous sections, an *identifier* of a process was encoded in the base language as a record of message predicates. It is, therefore, implemented as a reference to a method dispatch table, that is, a record of message handlers. By type inference, each record is compiled to just an array.

Implementation of message passing is slightly different depending on whether we allow inheritance or not. Without inheritance, the sender of a message first computes a reference to the message handler by using a reference to the receiver's method dispatch table and the index for the message handler to be invoked, and then sends it with the message arguments. The index for the message handler can be directly computed using Ohori's compilation method for a record calculus[10].

In order to incorporate inheritance, we must efficiently implement record coercion operations. We refine a method dispatch table as a pair consisting of an array of message handlers and an indirect index table to access it. Then, a coercion operator has only to create a new index table.

Figure 5 illustrates how method dispatch tables are created. The figure shows what happens when we create a point3 process by

```
$self.(new_point3 (1.0, 2.0) self | ...).
```

Creation of method dispatch tables goes as follows:

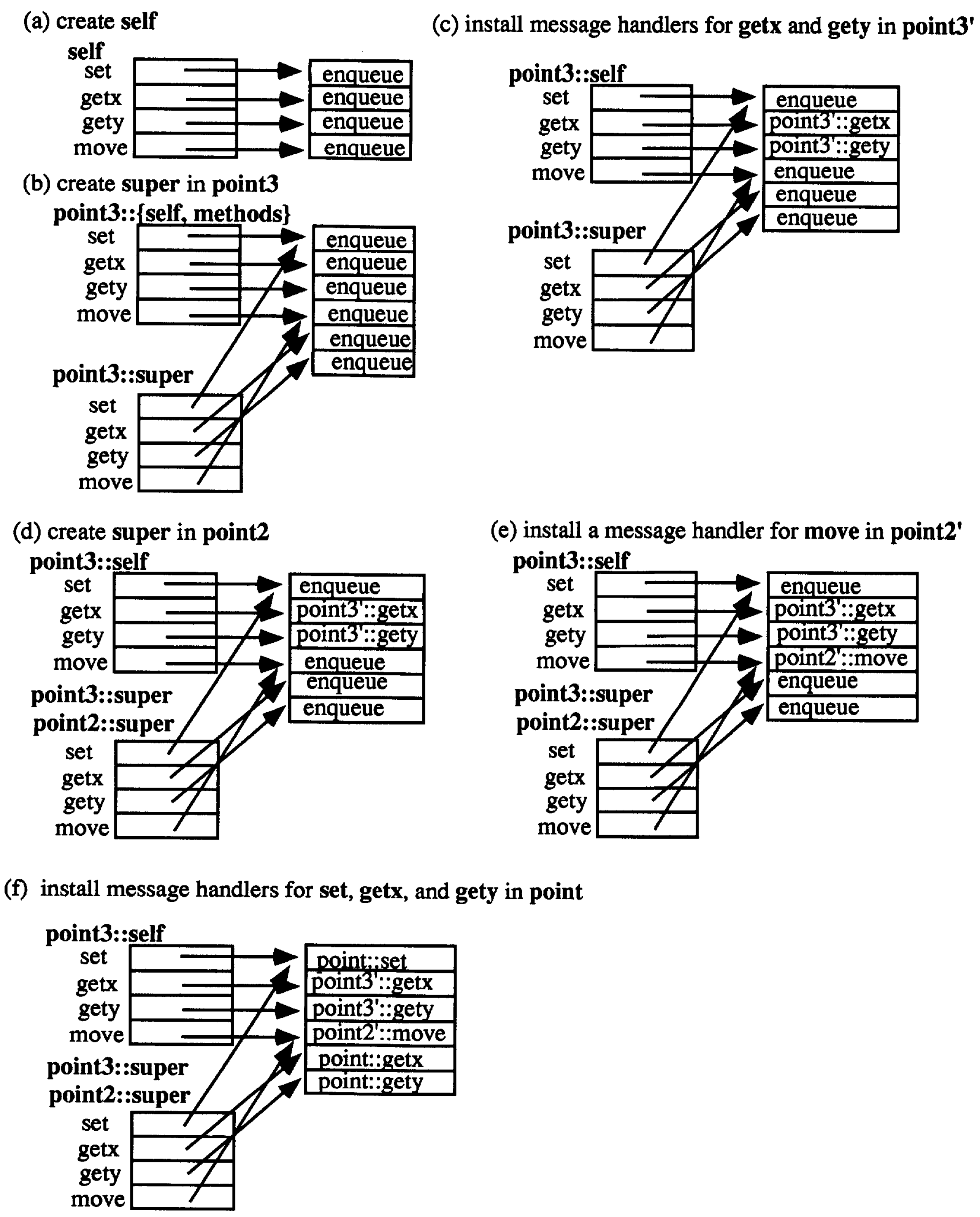
(a) create self
self
set
getx
gety
move
enqueue
(b) create super in point3
point3::{self, methods}
point3::super
(c) install message handlers for getx and gety in point3'
point3::self
point3'::getx
point3'::gety
(d) create super in point2
point2::super
(e) install a message handler for move in point2'
point2'::move
(f) install message handlers for set, getx, and gety in point
point::set
point::getx
point::gety

Figure 5: Creation of method dispatch table

1. Initially, `self` is created by the operator `$` (see Figure 5(a)). From the type of `new_point3`, we know that `self` is a record consisting of four fields `set`, `getx`, `gety` and `move`. Therefore, an array of size 4 is created. At this moment, since a message handler for each message is not ready, a procedure for enqueueing incoming messages is placed in each entry of the method lookup table.

2. `point3` is invoked and `super` is created by applying coercion and record concatenation. The result is shown in Figure 5(b).

3. `point3'` is invoked, and message handlers for `getx` and `gety` are placed (Figure 5(c)).

4. `point2` is invoked, and `super` is created by coercion. Because the coerced record can be shared with `super` in `point3`, `self` points to the same structure as `super` in `point3` points (Figure 5(d)).

5. `point2'` is invoked and a message handler `move` is placed (Figure 5(e)).

6. `point` is invoked, and message handlers `set`, `getx` and `gety` are placed (Figure 5(f)).

Thus, an implementation of method lookup tables for a concurrent object-oriented language is not a hacking task at all, but naturally derived from encoding of the language and type inference in HACL extended with polymorphic record calculus.[5]

## 5 Discussions

As far as the authors know, this is the first attempt to obtain a type system for concurrent object-oriented programming by directly extending typed concurrent calculus[9][11][16]. Most closely related is Vasconcelos' work on a calculus of objects[17][15]. Although his type system seems to be inspired by work on typed concurrent calculus[16], it stands alone and is not a direct extension of concurrent calculus. On the other hand, our type system is directly obtained by encoding a concurrent object-oriented language in a concurrent calculus extended with record operations. One of major advantages of our approach is its flexibility. We first gave a type system for a concurrent object-oriented language without inheritance, and later refined it to incorporate inheritance by just refining the translation; *we need not modify the basic type system of concurrent calculus.* This is not the case in Vasconcelos' system: In order to allow inheritance, his system seems to require major changes in the type system. Our approach can, therefore, account for a much wider range of mechanisms of concurrent object-oriented programming with less efforts. In fact, we can easily embed Vasconcelos' calculus of objects into Higher-Order ACL (see Appendix A).

The reason we chose Ohori's record calculus in extending Higher-Order ACL is that it was simple and powerful enough to capture mechanisms treated in this paper. Moreover, Ohori's record calculus is shown to be efficiently implemented[10]. On the other hand, although Higher-Order ACL extended with recent sophisticated type systems for object-oriented languages[13] is powerful and might be necessary to capture more complex mechanisms for concurrent object-oriented programming, there seems no established way for implementing it really efficiently. Pierce and Turner's recent work on PICT[12], which is a concurrent language based on Milner's $\pi$-calculus, is going into this direction. At least, it does not provides much insight on implementation techniques for the existing concurrent object-oriented languages. Our research aim is not only to design 'yet another concurrent object-oriented language' but to investigate foundations of language design, types, program analysis, and implementation techniques for concurrent object-oriented languages, through HACL.

Some work has been done to support object-oriented programming style based on *traditional* concurrent logic programming[6]. The purpose of

[5]For the sake of simplicity, we illustrated a naive scheme for the creation of method dispatch tables. The defect of this scheme is that a method dispatch table is created for each object, not for class. Actually, method dispatch tables can be shared among objects of the same class, using a similar scheme incorporated in ABCL/onAP1000[14].

this paper is not only to provide our specific language HACL with object-oriented programming facilities, but also to give foundations on type systems and implementation techniques for a wide range of (non-logical) concurrent object-oriented programming languages. Traditional concurrent logic programming does not seem to meet this requirement, because of several gaps, for example, a gap between stream-based communication and address-oriented communication; in order to realize many-to-one communication as in Actors, explicit stream merge operations are required.

Our ACL[8][7] is initially inspired by Andreoli and Pareschi's $LO$[2][3], and developed as a natural framework to capture actor-based concurrent object-oriented programming. The encoding of inheritance in the simpler case of without method overriding is essentially the same as that proposed in [2] (although they might look quite different from each other). However, the introduction of higher-order processes together with the name creation operator enhanced the encapsulation mechanism, and enabled method overriding.

## 6 Conclusion

This paper developed a type system for concurrent object-oriented programming from the type system of Higher-Order ACL, in a similar manner to the development of typed object-oriented programming from $\lambda$-calculus. We first gave the translation of a simple concurrent object-oriented language into Higher-Order ACL, and then refined it step by step to capture more complex mechanisms like inheritance and self and super pseudo variables. An advantage of such an approach is that we can capture a wide range of mechanisms for concurrent object-oriented programming without modifying the underlying concurrent calculus and its type system. We have also shown that type inference naturally realizes an efficient implementation for message passing mechanisms. The type inference system and interpreter for Higher-Order ACL extended with polymorphic record calculus have been already implemented[6]. Its compiler is currently under development. As for the design of concurrent object-oriented languages, much work is left to be done. For example, it is possible to separate a specification and an implementation of each class by utilizing higher-order processes of HACL. Future work also includes the implementation of a new high-performance concurrent object-oriented language on massively parallel processors, based on this work.

## Acknowledgment

We are especially grateful to Satoshi Matsuoka for his valuable comments. This line of our work has been much improved by conversations with many people including Atsushi Ohori, Jean-Marc Andreoli, Remo Pareschi, Dale Miller, Vijay Saraswat, Samson Abramsky, Simon Gay, and Vasco Vasconcelos. We are also thankful to the anonymous referees for helpful comments.

## References

[1] Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Andreoli, J.-M., and R. Pareschi, "Linear Objects: Logical processes with built-in inheritance," *New Generation Computing*, vol. 9, pp. 445–473, 1991.

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

[4] Bruce, K. B., "Safe Type Checking in a Statically-Typed Object-Oriented Programming Language," in *Proceedings of POPL*, pp. 285–298, 1993.

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

[6] Kahn, K., E. D. Tribble, and M. S. Miller, "Vulcan: Logical Concurrent Objects," in *Concurrent Prolog: Collected Papers* (E. Y. Shapiro, ed.), vol. 2, ch. 30, pp. 274–303, 1987.

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

[6] Available via anonymous ftp from `camille.is.s.u-tokyo.ac.jp: pub/hacl`

[8] 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.

[9] 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.

[10] Ohori, A., "A Compilation Method for ML-Style Polymorphic Record Calculi," in *Proceedings of POPL*, pp. 154–165, 1992.

[11] Pierce, B., and D. Sangiorgi, "Typing and Subtyping for Mobile Processes," in *Proceedings of LICS*, pp. 376–385, 1993.

[12] 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.

[13] Pierce, B. C., "Simple Type-Theoretic Foundations For Object-Oriented Programming," *Journal of Functional Programming*, 1993.

[14] Taura, K., S. Matsuoka, and A. Yonezawa, "An Efficient Implementation Scheme of Concurrent Object-Oriented Language on Stock Multicomputers," in *Proceedings of PPOPP*, 1993.

[15] Vasconcelos, V. T., "Typed Concurrent Objects," in *Proceedings of ECOOP'94*, vol. 821 of *LNCS*, pp. 100–117, Springer Verlag, 1994.

[16] 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.

[17] 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 *LNCS*, pp. 460–474, Springer Verlag, 1993.

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

[19] Yonezawa, A., S. Matsuoka, M. Yasugi, and K. Taura, "Implementing Concurrent Object-Oriented Languages on Multicomputers," *IEEE Parallel & Distributed Technology*, vol. 1, pp. 49–61, 1993.

# Appendix A Encoding of Vasconcelos' calculus of objects into Higher-Order ACL

This section gives an encoding of Vasconcelos' calculus of objects[17] into the Higher-Order ACL extended with record calculus.

A set of processes, ranged over by $P$, in Vasconcelos' calculus is given as follows:

$P ::= a \triangleleft l(\vec{v})$ (sends a message $l(\vec{v})$ to $a$)
  $\mid a \triangleright [l_1(\vec{x_1}).P_1 \& \cdots \& l_n(\vec{x_n}).P_n]$ (receives a message $l_i(\vec{v_i})$ and then becomes $P_i[\vec{v_i}/\vec{x_i}]$)
  $\mid P_1, P_2$ (parallel composition)
  $\mid (\nu x)P$ (scope restriction)
  $\mid !P$ (replication)
  $\mid \mathbf{0}$ (inaction)

where $\vec{v}$ stands for a sequence $v_1, \ldots, v_n$.

Vasconcelos' calculus can be obviously encoded as follows:

$\mathcal{F}(a \triangleleft l(\vec{v})) = \mathtt{a.l}(\vec{v})$
$\mathcal{F}(a \triangleright [l_1(\vec{x_1}).P_1 \& \cdots \& l_n(\vec{x_n}).P_n]) =$
  $\mathtt{a.l}_1(\vec{x_1})\texttt{=>}\ \mathcal{F}(P_1)\texttt{\&} \cdots \texttt{\&}\ \mathtt{a.l}_n(\vec{x_n})\ \texttt{=>}\ \mathcal{F}(P_n)$
  where $\mathtt{a} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
    for some $\tau_1, \ldots, \tau_n \cdots (*)$
$\mathcal{F}(P_1, P_2) = \mathcal{F}(P_1) \mid \mathcal{F}(P_2)$
$\mathcal{F}((\nu x)P) = \texttt{\$x}.\mathcal{F}(P)$
$\mathcal{F}(!P) = ?\mathcal{F}(P)$
$\mathcal{F}(\mathbf{0}) = \_$

where the condition $(*)$ is attached to the translated expression as a partial specification of the type of **a** as follows:

```
(a:{l1:'a1,..., ln:'an}).l1 x => P1
& ...
& a.ln x => Pn
```

It is trivial that $P$ is well-typed in Vasconcelos' calculus if and only if $\mathcal{F}(P)$ is well typed in the extended Higher-Order ACL. Thus, Vasconcelos' calculus of objects can be considered an instance of the extended Higher-Order ACL.