An Interaction-based Language and its Typing System

Kaku Takeuchi Kohei Honda Makoto Kubo

Department of Computer Science
Keio University

3-14-1, Hiyoshi, Kohoku-ku, Yokohama 223, Japan

{kaku,kohei,kubo}@mt.cs.keio.ac.jp

Abstract. We present a small language $\mathcal{L}$ and its typing system, starting from the idea of interaction, one of the important notions in parallel and distributed computing. $\mathcal{L}$ is based on, apart from such constructs as parallel composition and process creation, three pairs of communication primitives which use the notion of a *session*, a semantically atomic chain of communication actions which can interleave with other such chains freely, for high-level abstraction of interaction-based computing. The three primitives enable programmers to elegantly describe complex interactions among processes with a rigorous type discipline similar to ML [4]. The language is given formal operational semantics and a type inference system, regarding which we prove that if a program is well-typed in the typing system, it never causes run-time error due to type inconsistent communication patterns, offering a new foundation for type discipline in parallel programming languages.

1 Introduction

The idea of *interaction*, that is, reciprocal actions between multiple parties, is one of the central notions in parallel and distributed computing, from protocol exchange between a client process and a server process in multi-processing environments, to communication between distributed software modules, to actions between a mobile robot and its environments, to reactive human-interface, etcetera. Moreover, in many such cases, the meaning of interaction between processes running on parallel becomes clear only when seen as a *chain* of reciprocal actions between two parties, sometimes interleaved with actions with other parties. One example is an application consisting of multiple, possibly distributed, parallel processes, which proceeds its computation by way of communication among component processes, some of them being various servers and some of them being services from operating systems. In such a situation, each interaction tends to last for some time, consisting of a chain of reciprocal actions between two parties.

Under such situations, the abstraction methodology to describe complex, dynamic interaction structures based on a rigorous mathematical discipline, is clearly required. The abstraction should allow, in addition to the usual "one-time only" style of communication (e.g. call-return, pass-away, etc.), the ways of *organizing* (or *composing*) several reciprocal communication actions into a semantically single operation. Such organization in turn may well be based on a type discipline, which ensures well-typed programs to behave in a principled fashion, e.g. not causing run-time errors due to inconsistency of communication patterns between two parties. This is especially true when one can send port names for communication to another party dynamically, as found in e.g. parallel object-oriented languages such as POOL[2] or Actors[1, 6], or ML-based concurrent languages such as CML[24], since, in such situations, communicating entities using a certain port name can be known only dynamically at run time.

The present paper is a proposal of elementary programming language constructs for such abstraction and the accompanying type discipline. The proposed constructs are simple, but allow programmers to concisely codify complex structures of reciprocal communication in a flexible way. While programming is done in an untyped fashion, a type discipline, which easily yields an effective typing procedure, provides us with a way of checking the consistency of communication patterns between interacting parties, similarly to typed functional programming languages such as ML [4]. One crucial language construct both for flexible programming and type abstraction is a ***session point***, which serves as a private channel for a course of interaction, newly created each time an interactive session between two parties starts. Thus, while the initiation of one session takes place through a usual port name, which may be shared by more than two parties, each action composing a session is done only through the associated private channel. This separation of the notion of session points from usual port names, suggested by two distinct ways of using names in name passing process calculi such as π-calculus [17] and ν-calculus [11] (see Section 2), turns out to be essential to *both* a flexible way of composing interactions *and* a clean type abstraction for resulting operational structures. This will be exhibited in the present paper using a small example language and its type system. Specifically we show that typability of a program ensures the lack of run-time errors including inconsistency between communication patterns, which we believe to be an important discipline in parallel, communication-based programming.

The structure of the paper follows. Section 2 develops basic ideas of interaction primitives, together with the motivations. Section 3 introduces a small language based on those primitives, where the syntax and an operational semantics is formulated. Section 4 is devoted to the formulation of a typing system and the study of its basic properties. Finally Section 5 gives comparison with other work and discusses possible future subjects of study.

2 Basic Idea

2.1 Communication Primitives

We present the basic idea underlying our interaction primitives informally. They will be given a more formal treatment in later sections. We begin with the simplest one, *synchronous input* and *output*.

Synchronous communication of values, as found in e.g. CSP [7, 8] or π-calculus [17], is one of the simplest primitives for interaction[1]. We write them:

(1) `k!v; A`
which sends a data `v` to a name `k` and then performs the action `A`.

[1] We note that, at the level of the basic calculus, [10, 11] have shown that a simpler primitive called asynchronous name passing, suffices to represent general interaction structures. However for the high-level description and type abstraction of interaction behaviour , the synchronous communication primitive is found to be essential.

(2) **k?x in B**
which assigns the data it received from a name **k** into a variable **x** and then performs the action **B**.

The operational idea underlying these primitives can be represented by the following notion of reduction:

```
k!v; A , k?x in B --> A , B[v/x]
```

where '-->' is the reduction relation and ',' denotes the parallel composition of processes. "B[v/x]" means substituting **v** for the free occurrences of **x** in an action **B**.

Now, when one considers successive interaction between two parties, which as a whole constitutes one structured communication, the issue of *interference*[12] becomes important. That is, if the third party can interfere with the communication through the shared port, the intended communication becomes impossible. In the practice of networked computing, this problem is resolved by utilizing the idea of the *session* which is private to the concerned communicating parties and which offers capability to perform interaction without interference from the third party. The idea turns out to be neatly formalizable by the name passing notion together with scope restriction of names, as found in name passing process calculi [10, 17]. Essentially processes initially exchange a private port name and subsequently interact through it. The following shows the case where two processes transmit a sequence of data consecutively between each other. Below "(c)A" denotes that the port name **c** is private (restricted) to **A**.

```
    a?x in x?y in x!125; A   , (c)(a!c; c!250; c?z in B)
--> (c)(c?y in c!125; A[c/x]  , c!250; c?z in B)
--> (c)(c!125; A[c/x][250/y]   , c?z in B)
--> (c)(A[c/x][250/y] , B[125/z])
```

Thus, by initially communicating a private name, a local communication link between two processes is established. Similar constructions are omnipresent in the encoding examples of data-control structures in name passing process calculi [10, 17]. This special use of the name passing primitive is so important that it had better been given its own status, separately from the usual name passing. This separate treatment is actually essential to have type abstraction for such "sessions", as we shall see soon. The dual primitives for link-creation, with which multiple communication actions can be dealt with as a (semantically) single interaction event, follow.

(3) **accept (a, k) in A**
which accepts the establishment of a local channel through a port name **a**, then, after creating a channel **k**, performs the next action **A**.

(4) **request (a, k) in B**
which requests a linkage through **a**, then, after acquiring the local channel **k**, performs the action **B**.

Such local channels may sometimes be called *session points*, or, more simply, just *channels*, separately from usual port names. After **accept** and **request** actions, the communication is performed through the created (private) channel. Thus names are persistent identifiers for communication, while channels are only used during each interactive session. **accept** and **request** together make it possible to perform private communication that other processes cannot access. Using these primitives, the preceding transition becomes:

```
accept (a, k) in k?y in k!125; A , request (a, k) in k!250; k?z in B
--> |k|(k?y in k!125; A ,  k!250; k?z in B)
--> |k|(k!125; A[250/y] ,  k?z in B)
--> |k|(A[250/y] ,  B[125/z])
```

where we assume **k** does not occur free in **A** and **B** and **|k|** denoting scope restriction of the channel **k**. Thus actions initiating one session and actions which constitute the session become separate.

Finally, the communication using such a private channel can not only be transmission of data but also include control-like features. One essential construct of the kind is branching/selection primitives. They roughly correspond to the field-selection in record-based functional calculi, or method invocation in object-oriented programming languages [29].

(5) $\mathtt{k}\lhd\mathtt{l};\ \mathtt{A}$
which selects one of options by a label **l** through the channel **k** and performs an action **A**.
(6) $\mathtt{k}\rhd\{\mathtt{l}_1:\mathtt{A}_1,\ \cdots,\ \mathtt{l}_n:\mathtt{A}_n\}$
which waits with multiple labels $\mathtt{l}_1\cdots\mathtt{l}_n$ for the next action through the channel **k**.

Later we formalize the above idea by giving operational semantics to these primitives, using a small language based on them. Here we proceed to the introduction of the idea of the type notion accompanying these communication primitives.

2.2 Type Abstraction

Type systems in functional programming languages such as ML allow us to type programs according to their applicative behavior, and helps us programming in a disciplined fashion, assuring, among others, that no type errors can occur during program execution. In the same way, here we need the type abstraction for programs based on their interactive behavior. This becomes possible mainly due to the introduction of the syntactic construct for explicitly establishing the local communication channel we discussed already. Take the example of the following fragment of a program:

```
(1)  request (a, k) in
(2)      k?x in
(3)          request (b, j) in
(4)              j?y in
(5)                  k!(x-y);
(6)                  j!true;
(7)                      ...
```

If we take a closer look, we know there are *two* flows of interaction, i.e.:

- With a, one interacts in the sequence (1)—(2)—(5)
- With b, one interacts in the sequence (3)—(4)—(6)

Note that the extraction of individual communication flows only becomes possible by the introduction of local channels separately from usual port names. Now the idea is to consider such a pattern of individual information flow as a type of interaction. We need some notations: we write $\uparrow p$ and $\downarrow p$ for output and input of the value of type p. Then, assuming **int** and **bool** stand for the types of the integer and the boolean values respectively, one gets types for interaction as follows.

- With a, one has a type $\downarrow$int $\uparrow$int
- With b, one has a type $\downarrow$int $\uparrow$bool

Each type abstraction, one for each name, is formally called a *session*. The notion of session is crucial in categorizing the basic behavior of communicating entities, or programs. As an example, we easily know that a process named a has a session $\uparrow$int $\downarrow$int, which is dual to what is given above.

For the branching constructs, we also have dual type abstractions.

$$k \rhd \{\ \text{'read:}k\,!\,value;\ \cdots,\ \ \text{'write:}k?x \textbf{ in } \cdots\ \}$$

In this example, the process sends *value* if another party likes to "read", and receives if another party likes to "write". Then the type abstraction for the fragment naturally becomes, regarding a channel k, $\&\langle\text{'read :}\uparrow\alpha..,\ \text{'write :}\downarrow\beta..\rangle$ (here α and β are type-variables). Now the other party who interacts with the program above should have a dual session, using the symbol $\oplus$, e.g. $\oplus\langle\text{'read :}\downarrow\alpha..,\ \text{'write :}\uparrow\beta..\rangle$. Note how the use of a local channel, k, is again crucial for the extracting type information.

The formal construction based on these ideas will be done in Section 4, centering on the typing system for a small language and its basic properties, including a weak form of Subject Reduction and Error-Free properties.

3 A language $\mathcal{L}$

Below we present a small example language which substantiates the six communication primitives we discussed in the previous section. It also utilizes the notion of process-creation and name/channel scope restriction, and, in addition, auxiliary commands such as assignment and if-then-else. The language is called $\mathcal{L}$. After introducing the syntax, we define operational semantics (reduction relation) over programs. Finally we formulate a natural notion of run-time errors in the present setting.

3.1 Syntax

We begin with basic entities used for the language. Since, as we discussed, we distinguish between *port names* and *session points*, the latter we shall simply call *channels*, we have distinct categories for each. $a, b, c, \cdots$ and $k, k', k'', \cdots$ range over port names and channels, respectively. Then $l, l', \cdots$ range over *labels* used for branching. We also assume the set of *expressions*, which will include, among others, names, name-variables, arithmetic expressions, and boolean expressions, including variables, but *not* channels (essentially this is to prohibit channels from being passed around in communication). The symbols $e', e'', \cdots$ range over the set of expressions. $\dot{x}, \dot{y}, \dot{z}, \cdots$ range over name-variables, and $u, u', u'', \cdots$ range over the union of name-variables and names. $x, y, z, \cdots$ range over variables for expressions in general. $\mathsf{c}, \mathsf{c}', \cdots$ range over *constants*, which are simply names, integers, or boolean values. We will also use *process variables* over which $X, Y, Z, ..$ range. The syntax follows.

Definition 3.1 (Syntax of $\mathcal{L}$)

$$
\begin{array}{rcl}
\textit{prog} & ::= & \textit{dec}_1 \ \cdots \ \textit{dec}_n \ P \\
\textit{dec} & ::= & \texttt{def} \ X \ (\ x_1, \cdots, x_n \ ) \ . \ P \\
P & ::= & \textit{act} \ \mid \ X \ (\ e_1, \cdots, e_n \ ) \ \mid \ P, Q \ \mid \ (\ a \ ) \ P \ \mid \ \mid k \mid P \\
\textit{act} & ::= & \texttt{accept} \ (\ u, \ k \ ) \ \texttt{in} \ \textit{act} \ \mid \ \texttt{request} \ (\ u, \ k \ ) \ \texttt{in} \ \textit{act} \\
& \mid & k \ ! \ e \ ; \ \textit{act} \ \mid \ k \ ? \ x \ \texttt{in} \ \textit{act} \\
& \mid & k \lhd l \ ; \ \textit{act} \ \mid \ k \rhd \{ \ l_1 : \textit{act}_1, \cdots, l_n : \textit{act}_n \ \} \\
& \mid & \texttt{create} \ P \ ; \ \textit{act} \ \mid \ x = e \ \texttt{in} \ \textit{act} \\
& \mid & \texttt{if} \ e \ \texttt{then} \ \textit{act} \ \texttt{else} \ \textit{act} \ \mid \ \texttt{inact}
\end{array}
$$

There are three main syntactic categories, *programs*, *processes*, and *actions*. A program consists of a set of declarations for distinct process variables (such a set of declarations form multiple recursion), and the main part, which is a process. A *process* is usually a concurrent composition of multiple actions and instantiations (these are explained below) together with name/channel scope restriction. An *instantiation*, "$X(e_1, \cdots, e_n)$", instantiates a process variable X into a process using its defining equation and its parameters $e_1, \cdots, e_n$. "P, Q" is a *concurrent composition* of P and Q. "$(a)P$" is a *name scope restriction*, cf. [17]. Next, the *channel scope restriction* is specific to $\mathcal{L}$, and indispensable to represent the usage of channels as a private interaction point between two processes. "$|k|P$" means that the channel k is private within P. "$|k_1 \cdots k_n|P$" and "$(a_1 \cdots a_n)P$" are abbreviations of "$|k_1| \cdots |k_n|P$" and "$(a_1) \cdots (a_n)P$" respectively.

An *action* is, roughly speaking, a process which engages itself in a thread of interaction, composed of a sequence of interaction primitives discussed in the previous section. Firstly, "`accept`$(u,\ k)$ `in` *act*" and "`request`$(u,\ k)$ `in` *act*" together establish a private channel between interacting parties and start communication by that channel. Here the initial k binds its free occurrences in *act*. Here and in the subsequent phrases, the keyword "`in`" is used to denote existence of binding. The next four primitives have also been discussed in the previous section: "$k\,!\,e;\,\textit{act}$" first sends the value of an expression e through a port k then does the next action, while "$k?x$ `in` *act*" receives a value through a channel k and instantiates it in the next action. We allow here a port name (not a channel) to be passed around, resulting in the dynamic communication topology as found in CML and Actors. "$k \lhd l;\ \textit{act}$" selects the l-field of the other party interacting through k, then does the next action, while in "$k \rhd \{\ l_1 : \textit{act}_1,\ \cdots,\ l_n : \textit{act}_n\ \}$", one offers the total n selections through a channel k. The next primitive, *process creation*, written "`create` P; *act*", dynamically generates a new process (then does the next action). *Assignment* and *conditional*[2] are standard except, in the assignment command, a variable is assigned only once. Finally "`inact`", the inaction, is the termination of action. "`inact`" is often omitted in later examples.

In the following, the set of programs is restricted to those which satisfy the following well-formedness (*w.f.* for short) condition. While some of such properties can be ensured by a typability condition, these properties are quite natural. Below and henceforth $\mathcal{FN}(P)$, $\mathcal{FC}(P)$ and $\mathcal{FV}(P)$ mean the sets of freely occurring names, channels and variables in P, respectively.

[2] We note that these two can be mapped to the language without them faithfully, both in terms of operational semantics and type structures. Without them, however, the program becomes much more difficult to understand.

Definition 3.2 (well-formedness)
(1) $dec_1, \cdots, dec_n\, P$ is $w.f.$ if dec_i is $w.f.$ $(1 \leq i \leq n)$ and $\mathcal{FC}(P) = \mathcal{FV}(P) = \emptyset$
(2) **def** $X\ (x_1, \cdots, x_n).P$ is $w.f.$ if $\mathcal{FN}(P) = \mathcal{FC}(P) = \emptyset$ and $\mathcal{FV}(P) \subseteq \{x_1, \cdots, x_n\}$

Subsequently we often treat a process as a program, implicitly assuming some fixed set of declarations for (distinct) process variables. The preceding well-formedness condition extends to such cases too.

Below we illustrate the usage of communication primitives by a simple example. To make the examples intelligible, strings of non-capital letters (e.g. cell, x) for port names or variables, and strings of capital letters (e.g. Cell) for process variables. We also use the string starting from "'" (e.g. 'write) for labels for branching.

Example 3.3 (Cell) An informal description of a cell is:

- When the client requires to *reset* the data of the cell, the cell resets its data and sets its status empty.
- When the client requires to *set* the new value to the cell, the cell sets the new value of clients to its data.
- When the client requires to *refer* to the data of the cell, then (1) if the data of the cell is empty then tell the client the failure of reference, (2) if not, the cell tells its data.

Now the following is the declaration of the behaviour of a cell.

```
1  def Cell ($\dot{x}$, $y$, $s$) .
2     accept ($\dot{x}$, $k$) in
3        $k \rhd$ {  'reset:  create Cell ($\dot{x}$, $y$, false)
4                  'set:    $k$?$z$ in create Cell ($\dot{x}$, $z$, true)
5                  'ref:    if $s$ then
6                              $k \lhd$'ok; $k$!$y$; create Cell ($\dot{x}$, $y$, $s$)
7                           else
8                              $k \lhd$'no; create Cell ($\dot{x}$, $y$, $s$)        }
```

In the above, line 1 defines a cell's action, where $\dot{x}$ is the name-variable of a cell and y is its content. Then s is the state of a cell, telling whether it is empty or full. Line 2 is the acceptance of a request from the user, creating a private channel k. In the rest it describes three branches and subsequent actions. In 'reset (line 3), it resets itself by regenerating an empty cell (but with the same name). In 'set (line 4), it reads a value and becomes a cell with a new content, again by process generation. In 'ref (lines 5-8), they describe somewhat a complex interaction sequence: when it is not empty, it sends a label 'ok and the value of $\dot{x}$ to the client through the channel k, else sends a label 'no to the client.Note how a certain kind of *exception handling* is thus expressed cleanly, going beyond usual one-direction method invocation.

Under the declaration, Cell (cell, 125, **true**) is the process with a name "cell" and a value 125.

3.2 Operational Semantics

The semantics for $\mathcal{L}$-processes is based on a *transition system*, which defines the reduction relation over processes and programs. Following Milner[16], we use *structural congruence*,

$\equiv$, to make the formulation of reduction relation much easier. The basic rules are given in Definition 3.4.

Definition 3.4 (Structural Congruence) $\equiv$ is the smallest congruence relation generated by:

$$P,Q \equiv Q,P \quad (P,Q),R \equiv P,(Q,R) \quad P,\mathbf{inact} \equiv P$$
$$P \equiv P' \ (if\ P \equiv_\alpha P') \quad (a)|k|P \equiv |k|(a)P$$
$$(a)P,Q \equiv (a)(P,Q) \ \ (if\ a \notin \mathcal{FN}(Q)) \quad |k|P,Q \equiv |k|(P,Q) \ \ (if\ k \notin \mathcal{FC}(Q))$$

where $P \equiv_\alpha Q$ means that P is α-convertible to Q, in terms of bound names, bound variables and bound channels. The reduction relation for processes is given next. We divide the definition into two parts: one for basic rules and the other for compositional rules. The reduction relation over processes under a suitable sequence of declarations is defined as the smallest relation closed under rules in Definitions 3.4 and 3.5.

Definition 3.5 (Reduction Rules 1)

[Link]	$(\mathtt{accept}(a,k)\ \mathtt{in}\ act_1),\ (\mathtt{request}(a,k)\ \mathtt{in}\ act_2) \to |k|(act_1,\ act_2)$
[Com]	$(k!\mathtt{c}; act_1),\ (k?x\ \mathtt{in}\ act_2) \to act_1, act_2[\mathtt{c}/x]$
[Label]	$(k \lhd l_i; act),\ k \rhd \{l_1: act_1, \cdots, l_n: act_n\} \to act,\ act_i \ \ (1 \le i \le n)$
[Ass]	$(x = \mathtt{c}\ \mathtt{in}\ act) \to act[\mathtt{c}/x]$
[Crea]	$(\mathtt{create}\ P; act) \to act,\ P$
[Cond]	$\mathtt{if\ true\ then}\ act_1\ \mathtt{else}\ act_2 \to act_1 \quad \mathtt{if\ false\ then}\ act_1\ \mathtt{else}\ act_2 \to act_2$
[ProcVar]	$X(\mathtt{c}_1, \cdots, \mathtt{c}_n) \to P[\mathtt{c}_1/x_1]\cdots[\mathtt{c}_n/x_n]$ (under def $X(x_1,\cdots,x_n).P$)
[Eval]	$e_1 \to_e e_2 \Rightarrow P(e_1) \to P(e_2)$

Definition 3.5 shows the basic transition rules for processes, substantiating our informal discussions on the primitives in Section 2. As was already said, we implicitly assume that a set of declarations is given, so that "$P \to P'$" actually means "$P \to P'$ under some declarations $dec_1, dec_2, .., dec_n$ ".

$P(e)$ denotes a prime process which contains e at the top level, i.e. which has one of the forms: $x = e$ in act', $k!e; act'$, or if e then act_1 else act_2, and the reduction for expressions, $e_1 \to_e e_2$, is assumed to be given in the usual way(cf. [5]). Note that, by the above rules, values are only transmitted in communication if they are evaluated to constants.

Definition 3.6 defines auxiliary transition rules, closing the relation under syntactic constructors such as name/channel scope restriction and parallel composition.

Definition 3.6 (Reduction Rules 2)

[NameScop] $P \to P' \Rightarrow (a)P \to (a)P'$ [Par] $P \to P' \Rightarrow P,Q \to P',Q$
[ChanScop] $P \to P' \Rightarrow |k|P \to |k|P'$ [Str] $P \equiv P'\ \underline{\text{and}}\ P' \to Q'\ \underline{\text{and}}\ Q' \equiv Q \Rightarrow P \to Q$

This defines reduction over processes under a set of declarations, which can easily be extended to the definition of reduction over the whole program, which we omit.

Below we present a simple example of reduction, using the code of a cell which appeared already.

Example 3.7 Let the declaration for a *client* be:

```
def Clnt (ċ).
   request(ċ, k) in
      k ◁ 'ref; k ▷ {'ok : k?x in act, 'no : ···}
```

Then, assuming the declaration of a cell in Example 3.3, we have the following reduction sequence.

$$
\begin{array}{ll}
 & \text{Cell(cell, 150, }\mathbf{true}\text{), Clnt(cell)} \\
\rightarrow & |k|(k \rhd \{\cdots\}, k \lhd \text{'ref}; \ k \rhd \{\text{'ok} : k?x \ \mathtt{in} \ act, \text{'no} : \cdots\} \\
\rightarrow & |k|(\mathbf{if\ true\ then}\ k \lhd \text{'ok}; \ k!150; \ \mathbf{create}\ \text{Cell(cell,150,}\mathbf{true}\text{)}\ \mathbf{else} \cdots, \\
 & k \rhd \{\text{'ok} : k?x \ \mathtt{in} \ act, \text{'no} : \cdots\} \\
\rightarrow & |k|(k \lhd \text{'ok}; \ k!150; \ \mathbf{create}\ \text{Cell(cell,150,}\mathbf{true}\text{)}, \ k \rhd \{\text{'ok} : k?x \ \mathtt{in} \ act, \text{'no} : \cdots\} \\
\rightarrow & |k|(k!150; \ \mathbf{create}\ \text{Cell (cell,150,}\mathbf{true}\text{)}, \ k?x \ \mathtt{in} \ act \\
\rightarrow^{*} & \text{Cell (cell,150,}\mathbf{true}\text{)}, \ act[150/x]
\end{array}
$$

which shows that the cell successfully transmitted the value to the client, as expected.

3.3 Errors in Values and Communication

To think of type constructor errors in the present setting, two kinds should be considered. One is the usual one due to the mismatch between an operator and its operands. Another is new in the present situation, denoting the incompatibility of communication patterns. We begin with the former, which is standard; let an expression be *immediately wrong* when wrong values are applied to an operator in the expression, e.g. $\mathbf{true} + 78$.

Another, and more interesting, type error arises when two communication patterns do not match. This is the notion of the type error specific to the present situation. This time the error concerns a pair of actions.

Definition 3.8 gives the set of process which contains both kinds of immediate errors.

Definition 3.8 (Immediate Error)

1. *e is immedately wrong* $\Longrightarrow$ $P(e) \in ImErr$.
2. if both *act* and *act′* start from actions with the same channel k and, moreover, do *not* take the following forms:
   $(act \ \equiv \ k!\mathrm{c}; act_1 \ \underline{and} \ act' \ \equiv \ k?y \ \mathtt{in} \ act_2)$ $\underline{or}$
   $(act \ \equiv \ k \lhd l; act_1 \ \underline{and} \ act \ \equiv \ k \ \rhd \{l_1 : act_1, \cdots, l_n : act_n\} \ \underline{and} \ l \in \{l_1, \cdots, l_n\})$
   then $act, act' \in ImErr$ (and their symmetric cases).
3. $P \in ImErr \implies (a)P \in ImErr \ \underline{and} \ |k|P \in ImErr \ \underline{and} \ P, Q \in ImErr$
4. $P \in ImErr \ \underline{and} \ P \equiv Q \implies Q \in ImErr$.

Finally, assuming implicitly a fixed set of declarations, we define *wrong processes.*

Definition 3.9 (Error) A process P is wrong, written $P \in Err$, if there exists Q such that $P \rightarrow^{*} Q$ and $Q \in ImErr$.

A simple example of a "wrong" process follows.

Example 3.10 (Cell and Wrong user) We again assume the declaration of a cell in Example 3.3. Now define:

$$
\begin{array}{l}
\mathbf{def}\ \text{WrongUser}\ (\dot{c}). \\
\quad \mathbf{request}(\dot{c}, k)\ \mathbf{in} \\
\quad\quad k \lhd \text{'set}; \ k?y \ \mathtt{in} \ \cdots \text{any action} \cdots
\end{array}
$$

Then we have:

WrongUser(cell),Cell(cell,125,**true**)$\in$ *Err*

Because:

$$\begin{array}{l} \text{WrongUser(cell),Cell(cell,125,}\mathbf{true}\text{)} \\ \to \mathbf{request}(\text{cell}, k)\ \mathbf{in}\, k \lhd \text{'set}; k?y\, \mathbf{in} \cdots,\ \mathbf{accept}\ (\text{cell}, k)\ \mathbf{in}\, k \rhd \{\cdots\} \\ \to |k|((k \lhd \text{'set}; k?y\, \mathbf{in} \cdots),\ k \rhd \{\text{'reset:}\cdots, \text{'set:}\cdots, \text{'ref:}\cdots\}) \\ \to |k|((k?y\, \mathbf{in} \cdots),\ (k?z\, \mathbf{in}\, \mathbf{create}\ \text{Cell}\,(\text{cell}, z, \mathbf{true}))) \\ \in \mathit{ImErr} \end{array}$$

In the next section, we show a typing system which detects this kind of type errors.

4 Typing System

In this section we present a type discipline for interaction-based computing, taking the language $\mathcal{L}$ as an example. After introducing the basic notions of types, we present a type system for untyped $\mathcal{L}$ programs. Then we study basic properties of well-typed programs, including a weak form of Subject Reduction, and show that well-typedness ensures that the program does not run into type-constructor errors.

4.1 Type Scheme

We first define type schemes for $\mathcal{T}$, a typing system for $\mathcal{L}$. We let $p, q, r, \ldots$ range over the type schemes. For simplicity we do not include recursive types, though its inclusion is straightforward, see [28].

Definition 4.1 (Type Scheme)

$$p ::= \epsilon \mid \alpha \mid c \mid\ \uparrow p \cdot q \mid\ \downarrow p \cdot q \mid\ \oplus \langle l_1 : p_1, \cdots, l_n : p_n \rangle \mid \& \langle l_1 : p_1, \cdots, l_n : p_n \rangle$$

In the above, ϵ is the empty communication type. $\alpha, \beta, \gamma, \ldots$ denote type variables. c means constant type like int, bool, $\cdots$. '$\cdot$' denotes a sequential composition of types, e.g. $\uparrow p \cdot q$ is the sequential composition of the types $\uparrow p$ and q. We often abbreviate this as $\uparrow p\ q$. $\uparrow p\ q$ denotes a type of firstly sending a value of type p, then does q. Similarly $\downarrow p\ q$ denotes a type of firstly receiving a value of type p then doing an action of type q. $\oplus\langle l_1 : p_1, \cdots, l_n : p_n \rangle$ and $\&\langle l_1 : p_1, \cdots, l_n : p_n \rangle$ are types of label-selecting and label-branching. The former is a type which will select with one of the specified labels and then does the subsequent action, while the latter is a type which offer multiple options to another party.

The idea of duality of two complementary communication patterns is essential to our type discipline. This is formalized as the notion of *co-types*.

Definition 4.2 (co-type) The type $\overline{p}$, the *co-type* of p, is inductively defined as follows:

$$\overline{\epsilon} \overset{\text{def}}{=} \epsilon \qquad \overline{c} \overset{\text{def}}{=} c \qquad \overline{\uparrow p \cdot q} \overset{\text{def}}{=} \downarrow p \cdot \overline{q} \qquad \overline{\downarrow p \cdot q} \overset{\text{def}}{=} \uparrow p \cdot \overline{q}$$

$$\overline{\oplus\langle l_1 : p_1, \cdots, l_n : p_n \rangle} \overset{\text{def}}{=} \&\langle l_1 : \overline{p_1}, \cdots, l_n : \overline{p_n} \rangle \quad \overline{\&\langle l_1 : p_1, \cdots, l_n : p_n \rangle} \overset{\text{def}}{=} \oplus \langle l_1 : \overline{p_1}, \cdots, l_n : \overline{p_n} \rangle$$

The co-type of the atomic types like ϵ, int, etc. are the same as the original type. Note that the co-type of the co-type of a type is always the original type itself. Also note that, in the *co-types* of sending/receiving types, there is no change in the carried types.

4.2 Typing System $\mathcal{T}$

One channel is used by two parties in reduction. To distinguish one party from another in types, we stipulate that the party which starts a session with "request" has a negative polarity, while the party starting with "accept" has a positive polarity. For the purpose, we introduce the set $\{+,-\}$ ranged by θ, θ'.

Using polarities, we can write k^-: q to denote a type of interaction through k for the "requesting" party, and dually k^+: p for the "accepting" party. Roughly speaking, if p and q satisfy $\overline{q} = p$, then we can say k^+: p and k^-: q are consistent with each other.

Based on the idea, we formulate the notion of type assignment for programs, processes, and actions. Essentially, we have, as an assumption, hypothetical type assignment for names, variables, and process variables, and, as a conclusion, type assignment for channels. While a single type is given to a term in the traditional functional types [4, 14, 15], many channel-type pairs are given to a program in our typing for interaction, showing existence of multiple interaction points (cf. [9, 28]). Thus the main sequent for processes takes a form: $\Gamma, B \vdash P \succ \Delta$ where Γ is a set of type assignments to names and variables, B is a set of type assignments to process variables, and Δ is a set of type assignments for channels[3], each of which is explained below. As said, types for names and channels show that $\mathcal{L}$-processes have many interfaces of communications, each of which is given a specific type.

Let ν stand for names or variables. Then we have a single type assignment, $\nu : p$, where we call ν the *subject* of assignment and p its *predicate*. If ν is a name, say a, $\nu : p$ roughly means $a^+ : p$, i.e. we only think of one polarity, corresponding to $k^+ : p$ (the use of one polarity is enough for checking consistency; also, usage of positive polarity is just for convenience). Let $\Gamma, \Gamma', \ldots$ range over the family of finite sets of such type assignments, called *typing*. For such Γ, we write the set of subjects occurring in Γ as $Subj(\Gamma)$.

A process variable is given a list of types $(p_1, \cdots, p_n)$, denoting types of expressions with which one can instantiate the variable. For example, given **def** $X\,(x_1, \cdots, x_n).P$, the type of process variable X is $(p_1, \cdots, p_n)$ when each parameter x_i has a type p_i in P. Let B be a finite set of type assignments for process variables, where each process variable only occurs once, like $B = \{X_1 : (p'_1, p''_1, \cdots), \cdots, X_n : (p'_n, p''_n, \cdots)\}$.

$\Delta, \Delta' \ldots$ range over the family of finite sets of such assignments, called *channel typing*. $Subj(\Delta)$ is the set of subjects in Δ. For the channel typings, the following consistency notion is important.

Definition 4.3 (Channel Compatibility) Δ and Δ' is consistent each other (denoted as $\Delta \asymp \Delta'$), iff
$\underline{\textit{If}}$ $k^\theta : p \in \Delta$ $\underline{\textit{then}}$ $(k^{\overline{\theta}} : \overline{p} \in \Delta'$ $\underline{\textit{or}}$ $\{k^+, k^-\} \cap Subj(\Delta') = \emptyset)$
$\underline{\textit{and}}$ $\underline{\textit{if}}$ $k^\theta : q \in \Delta'$ $\underline{\textit{then}}$ $(k^{\overline{\theta}} : \overline{q} \in \Delta$ $\underline{\textit{or}}$ $\{k^+, k^-\} \cap Subj(\Delta') = \emptyset)$

Our type inference system deals with five kinds of sequents, consisting of four auxiliary ones in addition to the main sequent.

1. $\Gamma \vdash e : p$ is a sequent for an expression, which reads: under the assumption Γ, $e : p$ can be derived.

[3] While the set of type assignments for names can be in the right-hand side of the turn-stile, this way is more convenient due to their treatment regarding weakening rules.

2. $\Gamma, B \vdash act \succ \Delta$ is a sequent for an action, which reads: under the assumption Γ, B, $act \succ \Delta$ can be derived. This is understood as the main sequent we already discussed.
3. $\Gamma, B \vdash P \succ \Delta$ which is a sequent for a process and was discussed already.
4. $B \vdash \mathbf{def}\ X(x_1, \cdots, x_n).P$, which is a sequent for a declaration which reads: under the assumption B, the declaration $\mathtt{def}\ X\ (x_1, \cdots, x_n).P$ is typable.
5. $\Gamma \vdash dec_1 \cdots dec_n P$ is a sequent for a program, which reads: under the assumption Γ, $dec_1 \cdots dec_n P$ is typable.

Now the type inference system for $\mathcal{L}$, called $\mathcal{T}$, follows. In the typing rules below, $\Gamma \cdot \{a : p\}$ means that the subject a does not occur in Γ. Similarly for $\Delta \cdot \{k^\theta : q\}$ and $B \cdot \{X : (p_1, \cdots, p_n)\}$. Finally $\Gamma / \{\nu_1, \cdots, \nu_n\}$ means the result of removing the assignments whose subject is $\nu_1, \cdots, \nu_n$. Except weakening rules, each rule corresponds to one syntactic category defined in Definition 3.1.

Definition 4.4 (Type inference rules of $\mathcal{T}$)

$$[\text{Name}]\ \Gamma \cdot \{a : p\} \vdash a : p \quad [\text{Var}]\ \Gamma \cdot \{x : p\} \vdash x : p \quad [\text{Exp}]\ \Gamma \vdash e^p : p$$

$$[\text{Acc}]\ \frac{\Gamma \cdot \{u : p\}, B \vdash act \succ \Delta \cdot \{k^+ : p\}}{\Gamma \cdot \{u : p\}, B \vdash \mathtt{accept}(u, k)\ \mathtt{in}\ act \succ \Delta}$$

$$[\text{Req}]\ \frac{\Gamma \cdot \{u : \overline{p}\}, B \vdash act \succ \Delta \cdot \{k^- : p\}}{\Gamma \cdot \{u : \overline{p}\}, B \vdash \mathtt{request}(u, k)\ \mathtt{in}\ act \succ \Delta}$$

$$[\text{Send}]\ \frac{\Gamma \vdash e : p \quad \Gamma, B \vdash act \succ \Delta \cdot \{k^\theta : p'\}}{\Gamma, B \vdash k!e;\ act \succ \Delta \cdot \{k^\theta : \uparrow p \cdot p'\}} \quad [\text{Recv}]\ \frac{\Gamma \cdot \{x : p\}, B \vdash act \succ \Delta \cdot \{k^\theta : p'\}}{\Gamma, B \vdash k?x\ \mathtt{in}\ act \succ \Delta \cdot \{k^\theta : \downarrow p \cdot p'\}}$$

$$[\oplus]\ \frac{\Gamma, B \vdash act \succ \Delta \cdot \{k^\theta : p\}}{\Gamma, B \vdash k \lhd l;\ act \succ \Delta \cdot \{k^\theta : \oplus\langle l : p, \cdots \rangle\}}$$

$$[\&]\ \frac{\Gamma, B \vdash act_i \succ \Delta \cdot \{k^\theta : p_i\} \ \ (\text{for } 1 \leq i \leq n)}{\Gamma, B \vdash k \rhd \{l_1 : act_1, \cdots, l_n : act_n\} \succ \Delta \cdot \{k^\theta : \&\langle l_1 : p_1, \cdots, l_n : p_n \rangle\}}$$

$$[\text{Crea}]\ \frac{\Gamma, B \vdash P \succ \emptyset \quad \Gamma, B \vdash act \succ \Delta}{\Gamma, B \vdash \mathtt{create}\ P;\ act \succ \Delta}$$

$$[\text{Cond}]\ \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma, B \vdash act_1 \succ \Delta \quad \Gamma, B \vdash act_2 \succ \Delta}{\Gamma, B \vdash \mathtt{if}\ e\ \mathtt{then}\ act_1\ \mathtt{else}\ act_2 \succ \Delta}$$

$$[\text{LocalVar}]\ \frac{\Gamma \vdash e : p \quad \Gamma \cdot \{x : p\}, B \vdash act \succ \Delta}{\Gamma, B \vdash x{=}e\ \mathtt{in}\ act \succ \Delta} \quad [\text{Inact}]\ \frac{-}{\Gamma, B \vdash \mathtt{inact} \succ \emptyset}$$

$$[\text{Init}]\ \frac{\Gamma \vdash e_i : p_i\ (\text{for } 1 \leq i \leq n)}{\Gamma, B \cdot \{X : (p_1, \cdots, p_n)\} \vdash X(e_1, \cdots, e_n) \succ \emptyset}$$

$$[\text{Par}]\ \frac{\Gamma, B \vdash P \succ \Delta_1 \quad \Gamma, B \vdash Q \succ \Delta_2}{\Gamma, B \vdash P, Q \succ \Delta_1 \cup \Delta_2}\ (\Delta_1 \asymp \Delta_2) \quad [\text{NameScop}]\ \frac{\Gamma \cdot \{a : p\}, B \vdash P \succ \Delta}{\Gamma, B \vdash (a)P \succ \Delta}$$

$$[\text{ChanScop}]\ \frac{\Gamma, B \vdash P \succ \Delta}{\Gamma, B \vdash |k|P \succ \Delta / \{k^+,\ k^-\}}\ (k^+ : p \in \Delta \Leftrightarrow k^- : \overline{p} \in \Delta)$$

$$[\text{Dec}]\ \frac{\{x_1 : p_1, \cdots, x_n : p_n\}, B \cdot \{X : (p_1, \cdots, p_n)\} \vdash P \succ \emptyset}{B \cdot \{X : (p_1, \cdots, p_n)\} \vdash \mathtt{def}\ X\ (x_1, \cdots, x_n).P}$$

$$[\text{Prog}]\ \frac{B \vdash dec_i\ (\text{for } 1 \leq i \leq n) \quad \Gamma, B \vdash P \succ \emptyset}{\Gamma \vdash dec_1 \cdots dec_n P}$$

$$[\text{WeakChan}]\ \frac{\Gamma, B \vdash act \succ \Delta}{\Gamma, B \vdash act \succ \Delta \cdot \{k^\theta : \epsilon\}}\ (\{k^+, k^-\} \cap Subj(\Delta) = \emptyset)$$

Given the typing system, the following gives a simple example of a typing assignment. For more examples, see [25].

Example 4.5 Assuming the declaration in Example 3.3, if we assign a type to Cell(cell, 125, true), the port name "cell" is given a type:

$$cell : \langle\text{'reset} : \epsilon,\ \text{'set} :\downarrow\text{int},\ \text{'ref} : \oplus\langle\text{'ok} :\uparrow\text{int},\ \text{'no} : \epsilon\rangle\rangle$$

which clearly shows the structure of interaction to be done at the port "cell", including the exception handling we mentioned already.

4.3 Basic Properties of Type system $\mathcal{T}$

The aim of this subsection is to prove the error-free property for typable terms. To show the property, we need two steps. First, we show a weak form of Subject Reduction Property, i.e., if P is typable by $\mathcal{T}$ and $P \to P'$ then P' is also typable. Secondly, we show that if a process P is typable by $\mathcal{T}$ then P does not have an immediate type error. For the sake of space, the detailed proofs are left to [25].

Firstly, we show lemmas used to prove these properties.

Lemma 4.6
1. $\underline{Let}$ $\Gamma\cdot\{x\!:\!p\}, B \vdash P \succ \Delta$, $\underline{if}$ $a \notin Subj(\Gamma)$ $\underline{or}$ $\{a\!:\!p\} \asymp \Gamma$ $\underline{then}$ $\{a\!:\!p\}\cup\Gamma, B \vdash P[a/x] \succ \Delta$
2. $\underline{Let}$ $\Gamma\cdot\{x\!:\!p\}, B \vdash P \succ \Delta$, $\underline{if}$ $\Gamma \vdash \mathrm{c}\!:\!c$ $\underline{then}$ $\Gamma, B \vdash P[\mathrm{c}/x] \succ \Delta$
3. $\underline{If}$ $\Gamma, B \vdash P \ \succ \Delta$ $\underline{and}$ $P \equiv Q$ $\underline{then}$ $\Gamma, B \vdash Q \ \succ \Delta$.

1 and 2 are mechanically proved by induction on the structure of *act*, and 3 is proved by induction on the rules of $\equiv$. For expressions, under a suitable definition of $\to_e$, we have:

Lemma 4.7 $\underline{If}$ $\Gamma \vdash e_1\!:\!c$ $\underline{and}$ $e_1 \to_e^* e_2$ $\underline{then}$ $\Gamma \vdash e_2\!:\!c$.

Now, generally speaking, the typing of an $\mathcal{L}$-process may change its type whenever there is communication inside. Thus Subject Reduction Property in the usual sense does not hold. However, the next theorem tells us that, if an $\mathcal{L}$-process is typable, then an process after some reduction from the original process is again typable. This weak form of Subject Reduction is enough for our purpose.

Theorem 4.8 $\underline{If}$ $\Gamma, B \vdash P \succ \Delta$ $\underline{and}$ $P \to P'$ $\underline{then}$ $\underline{there\ exists}$ Δ' $\underline{such\ that}$ $\Gamma, B \vdash P' \succ \Delta'$

We can prove this property by induction on the structure of typable and reducible process using Lemmas 4.6 and 4.7. To reach the main theorem, we need the following lemma which says that a typable process does not have an immediate error.

Lemma 4.9 $\underline{If}$ $\Gamma, B \vdash P \succ \Delta$ $\underline{then}$ $P \notin ImErr$.

Using Theorem 4.8 and Lemma 4.9, we finally obtain:

Theorem 4.10 (Error Free) $\underline{If}$ $\Gamma, B \vdash P \succ \Delta$ $\underline{then}$ $P \notin Err$.

Thus typability ensures the lack of run-time errors including the mismatch in communication patterns. This not only helps programmers write correct programs, but also free the possible execution mechanisms from taking care of type errors, thus eliminating unnecessary codes. On other aspects of typing, including a typing algorithm for the present construction as well as more refined type disciplines, please consult Section 5.2 below.

5 Discussion

5.1 Comparison

In $\mathcal{L}$-programming, we use the channels explicitly to describe various patterns of interaction, so the programs look like CSP [7]. Separation of port names from private channels, hence the idea of a session of interaction, however, is not given in CSP, thus making type abstraction like $\mathcal{L}$ impossible. Moreover we cannot describe the dynamic link creation in CSP. While the concurrent object-oriented paradigm [1, 2, 12, 29, 30] gives a simple programming method based on the call/return style communication, our $\mathcal{L}$ can realize more flexible interaction patterns (cf. Section 3) and can abstract the chained continuous interaction flows by types, neither of which seem to have been proposed in those languages.

$\mathcal{L}$-processes are, in essence, sequential processes running on parallel, communicating via the persistent port names. In this sense, concurrent process calculi (like Theoretical CSP [8], π-calculus [16], and ν-calculus [10, 11]) present similar notions. However the distinct treatment between names and channels are not done in these calculi. In fact, while the channels in $\mathcal{L}$ are locally made by `accept/request` between two processes which are used to perform continuous interaction, these primitives do not exist in process calculi as such but are mixed with the general name passing notion if any. Hence the type abstraction as we presented in this paper may not be easily done.

The types in concurrent programming paradigm look more complex than those in traditional functional programming, and we paid attention to the *session* of processes as abstraction in those interaction. In this context, several proposals for typed abstraction of the communication between processes, have been recently done, including [21, 24, 3, 19, 23, 26, 9, 27]. [21, 24] propose concurrency and interaction primitives and associated type notions to be incorporated into ML-like functional programming languages[18]. In another approach [22], a programming language and a typing system as an extension of (polyadic) π-calculus is proposed. This approach tries to present a new language design and its typing system along the line of primitives inherent in π-calculus, and may be nearer to ours in that both try to develop language primitives and type notions solely based on the idea of interaction rather than extending the extant framework (though the present authors believe that two approaches will commute). In these precursors, however, only one-time communication (including multiple value passing) is considered and thus the way of freely combining multiple interactions and of type-abstracting the resulting interaction flow has not been considered, in contrast to our approach. This seems also to result in the low level description of the communication behaviour in these approaches. Apart from the difference, many fruitful interplays between our framework and those precursors seem possible, some of which are mentioned in 5.2. We also note that the type structure presented in [9] is similar, but, in $\mathcal{L}$, due to the explicit introduction of channels and accept/request primitives, the description of the interleaved interactions becomes possible, which is crucial for flexible programming based on interaction.

5.2 Future Work

While we have not presented a typing algorithm in this paper, it is easy to formulate a typing algorithm corresponding to our typing system. [25] gives an example of such an algorithm in the way similar to what are given in [28, 27], using the notion of *kinding* as presented in [20]. In [25], the kinding is also used to gain the principal types for processes in the typing system.

As discussed already, various typing notions have been studied for extensions of ML and name passing process calculi [3, 9, 19, 21, 23, 24, 26]. These can be combined to gain a more refined type discipline for our system. For example, information about communication topology will be extracted in the same way as in [19]. One important challenge in this regard is to ensure a certain form of a deadlock free property by typing. In fact, in the present system, even if a process is typable, its communication will be stopped in the middle of a session; since it can be interleaved by, say, another request command for which there is no corresponding party. The issue will be one of important subjects of future study. This can be related with a formal verification methodology in the name passing context as found in [12].

As an application of $\mathcal{L}$, we can consider the description and a type discipline for concurrent object-based programming languages using $\mathcal{L}$-primitives. For examples, the basic primitives used by concurrent objects, like message passing and method branch, can be described by our primitives. Moreover, description by $\mathcal{L}$ enables us to check the consistency of interaction of concurrent objects statically, such as incompatibility between PAST-type communication and NOW-type communication in ABCL[30].

Regarding the implementation, we note that $\mathcal{L}$-processes can be encoded into ν-calculus [10, 11] preserving their essential operational behavior. Since the calculus is based on asynchronous name passing, this means that, at least theoretically, the language can be implemented only using asynchronous message passing, and maybe suitable for distributed implementations. Presently we plan to implement the language primitives and their extensions on the abstract architecture [13] which uses the idea of [11].

Acknowledgement

The authors thank Vasco Vasconcelos and Nobuko Yoshida for discussions and comments on the paper. They also thank Professor Mario Tokoro for support. The second author is partially supported by JSPS Fellowships for Japanese Junior Scientists for which he is grateful.

References

1. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.
2. P. America, Operational Semantics of a Parallel Object-Oriented Language. In *POPL'86*, 1986.
3. D. Berry, R. Milner and D. Turner, A semantics for ML concurrency primitives. In *POPL'92*, 1992.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL'82*, 1982.
5. M. Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.
6. C. Hewitt, P. Bishop and R. Steiger, A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, 1973.

7. C. A. R. Hoare, Communicating sequential processes. *Communications of ACM*, 1978.
8. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
9. K. Honda, Types for Dyadic Interaction. In *CONCUR'93*, LNCS 612, Springer-Verlag, 1993.
10. K. Honda and M. Tokoro, An Object Calculus for Asynchronous Communication. In *ECOOP'91*, LNCS 512, Springer-Verlag, 1991.
11. K. Honda and N. Yoshida, Combinatory Representation of Mobile Processes. In *POPL'94*, 1994.
12. C. B. Jones, Constraining interference in an object-based design method, In *TAPSOFT'93*, LNCS 668, Springer-Verlag, 1993.
13. M. Kubo and A. Sashino, On Some Interaction Machines. *in preparation*, 1994.
14. J. C. Mitchell, Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science* B, MIT press, 1990.
15. R. Milner, A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, 1978.
16. R. Milner, Polyadic π-Calculus: a tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1992.
17. R. Milner, J. Parrow and D. Walker, A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100, 1992.
18. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT press, 1990.
19. H. R. Nielson and F. Nielson, Higher-Order Concurrent Programs with Finite Communication Topology. In *POPL'94*, 1994.
20. A. Ohori, A compilation method for ML-style polymorphic record calculi. In *POPL'92*, 1992.
21. A. Ohori and K. Kato, Semantics for Communication Primitives in a Polymorphic Language. In *POPL'93*, 1993.
22. B. C. Pierce, D. Remy and D. N. Turner, A Typed Higher-Order Programming Language Based on the Pi-Calculus. *Manuscript*, 1993.
23. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *1993 IEEE Symposium on Logic in Computer Science*, 1993
24. J. H. Reppy. CML: A Higher-Order Concurrent Language. In *PLDI'91*, 1991.
25. K. Takeuchi, Interaction-Based Programming Language and its Typing System. Master Thesis, Keio University, March 1993 (in Japanese). The English version in preparation.
26. B. Thomsen. A Calculus of Higher Order Communicating Systems. In *POPL'89*, 1989.
27. V. T. Vasconcelos. Predicative Polymorphism in π-Calculus. In *PARLE'94*, LNCS, Springer-Verlag, 1994.
28. V. T. Vasconcelos and K. Honda. Principal Typing-Schemes in a Polyadic π-Calculus. In *CONCUR'93*, LNCS 715, Springer-Verlag, 1993.
29. Y. Yokote and M. Tokoro. The Design and Implementation of ConcurrentSmalltalk. In *OOPSLA'86*, 1986.
30. A. Yonezawa, *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.