# Type Systems for Concurrent Programs

Naoki Kobayashi

Department of Computer Science
Tokyo Institute of Technology
kobayasi@kb.cs.titech.ac.jp

**Abstract.** Type systems for programming languages help reasoning about program behavior and early finding of bugs. Recent applications of type systems include analysis of various program behaviors such as side effects, resource usage, security properties, and concurrency. This paper is a tutorial of one of such applications: type systems for analyzing behavior of concurrent processes. We start with a simple type system and extend it step by step to obtain more expressive type systems to reason about deadlock-freedom, safe usage of locks, etc.

## 1 Introduction

Most of modern programming languages are equipped with type systems, which help reasoning about program behavior and early finding of bugs. This paper is a tutorial of type systems for concurrent programs.

Functional programming language ML [19] is one of the most successful applications of a type system that are widely used in practice. The type system of ML automatically infers what type of value each function can take, and checks whether an appropriate argument is supplied to the function. For example, if one defines a function to return the successor of an integer, the type system of ML infers that it should take an integer and return an integer:

```
fun succ x = x+1;
val succ = fn : int -> int
```

Here, the line in the italic style shows the system's output. If one tries to apply the function to a string by mistake, the type system reports an error before executing the program:

```
f "a";
Error: operator and operand don't agree ...
```

Thanks to the type system, most of the bugs are found in the type-checking phase.

Type systems for concurrent programming languages have been, however, less satisfactory. For example, consider the following program in CML [25].

```
fun f(x:int) = let val y=channel() in recv(y)+x end;
```

Function `f` takes an integer as an argument. It first creates a new communication channel `y` (by `channel()`) and then tries to receive a value from the channel. It is blocked forever since there is no process to send a value on `y`. This function is, however, type-checked in CML and given a type $\mathit{int} \to \mathit{int}$.

To improve the situation above, type systems for analyzing usage of concurrency primitives have been extensively studied in the last decade [2, 4–6, 10–13, 20–22, 31]. Given concurrent programs, those type systems analyze whether processes communicate with each other in a disciplined manner, so that a message is received by the intended process, that no deadlock happens, that no race condition occurs, etc.

The aim of this paper is to summarize the essence of type systems for analyzing concurrent programs. Since concurrent programs are harder to debug than sequential programs, we believe that type systems for concurrent programs should be applied more widely and play more important roles in debugging and verification of programs. We hope that this paper serves as a guide for those who are interested in further extending type systems for concurrent programs or incorporating some of the type systems into programming languages and tools.

We use the $\pi$-calculus [17, 18, 27] as the target language of type systems in this paper. Since the $\pi$-calculus is simple but expressive enough to express various features of real concurrent programming languages, it is not difficult to extend type systems for the $\pi$-calculus to those for full-scale programming languages.

Section 2 introduces the syntax and operational semantics of the $\pi$-calculus. In Sections 3–8, we first present a simple type system, and extend it step by step to obtain more advanced type systems. Section 9 concludes this paper.

## 2 Target Language

We use a variant of the $\pi$-calculus [17, 18, 27] as the target language. The $\pi$-calculus models processes interacting with each other through communication channels. Processes and communication channels can be dynamically created, and references to communication channels can be dynamically exchanged among processes so that the communication topology can change dynamically.

**Definition 1 (processes, values).** *The sets of* expressions*,* process expressions*, and* value expressions*, ranged over by $A$, $P$, and $v$ respectively, are defined by the following syntax.*

$$\begin{array}{ll} A & ::= P \mid v \\ P & ::= \mathbf{0} \mid x![v_1, \ldots, v_n] \mid x?[y_1 : \tau_1, \ldots, y_n : \tau_n].\, P \mid (P \,|\, Q) \\ & \quad \mid (\nu x : \tau)\, P \mid *P \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ v & ::= x \mid \mathbf{true} \mid \mathbf{false} \end{array}$$

In the definition above, $\tau$ denotes a type introduced in later sections. The type information need not be specified by a programmer (unless the programmer wants to check the type); As in ML [19], it can be automatically inferred in most of the type systems introduced in this paper.

Process $\mathbf{0}$ does nothing. Process $x![y_1, \ldots, y_n]$ sends a tuple $[v_1, \ldots, v_n]$ of values on channel $x$. Process $x?[y_1 : \tau_1, \ldots, y_n : \tau_n].\, P$ waits to receive a tuple $[v_1, \ldots, v_n]$ of values, binds $y_1, \ldots, y_n$ to $v_1, \ldots, v_n$, and behaves like $P$. $P \,|\, Q$ runs $P$ and $Q$ in parallel. Process $(\nu x)\, P$ creates a fresh communication channel, binds $x$ to it, and behaves like $P$. Process $*P$ runs infinitely many copies of $P$ in parallel. Process $\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ behaves like $P$ if $v$ is **true** and behaves like $Q$ if $v$ is **false**. For simplicity, we assume that a value expression is either a boolean (**true**, **false**) or a variable, which is bound to a boolean or a channel by an input prefix ($x?[y_1, \ldots, y_n].$) or a $\nu$-prefix.

We write $P \longrightarrow Q$ if $Q$ is reduced to $P$ in one step (by a communication or reduction of a conditional expression). The formal operational semantics is found in the literature on the $\pi$-calculus [17, 27].

We give below simple examples, which we will use later to explain type systems. In some of the examples, we use integers and operations on them.

*Example 1 (ping server).* The process $*ping?[r].\, r![\,]$ works as a ping server. It waits to receive a message on channel *ping* and sends a null tuple on the received channel. A typical client process is written as: $(\nu reply)\, (ping![reply] \,|\, reply?[\,].\, P)$. It creates a fresh channel *reply* for receiving a reply, checks whether the ping server is alive by sending the channel, waits to receive a reply, and then executes $P$. Communications between the server and the client proceed as follows:

$$\begin{array}{l} \quad *ping?[r].\, r![\,] \,|\, (\nu reply)\, (ping![reply] \,|\, reply?[\,].\, P) \\ \longrightarrow *ping?[r].\, r![\,] \,|\, (\nu reply)\, (reply![\,] \,|\, reply?[\,].\, P) \\ \longrightarrow *ping?[r].\, r![\,] \,|\, (\nu reply)\, P \end{array}$$

In the second line, $(\nu reply)$ denotes the fact that the channel *reply* is a new channel and known by only the processes in the scope.

*Example 2 (recursive processes).* Recursive processes can be defined using replications ($*P$). Consider a process of the form $(\nu p)\, (*p?[x_1, \ldots, x_n].\, P \,|\, Q)$. Each time $Q$ sends a tuple $[v_1, \ldots, v_n]$ along $p$, the process $[v_1/x_1, \ldots, v_n/x_n]P$ is executed. So, the process $*p?[x_1, \ldots, x_n].\, P$ works as a process definition. We write $\mathbf{let\ proc}\ p[x_1, \ldots, x_n] = P\ \mathbf{in}\ Q$ for $(\nu p)\, (*p?[x_1, \ldots, x_n].\, P \,|\, Q)$ below. For example, the following expression defines a recursive process that takes a pair consisting of an integer $n$ and a channel $r$ as an argument and sends $n$ messages on $r$.

$$\mathbf{let\ proc}\ p[n, r] = \mathbf{if}\ n \leq 0\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ (r![\,] \,|\, p![n-1, r])\ \ \mathbf{in} \cdots$$

*Example 3 (locks and objects).* A concurrent object can be modeled by multiple processes, each of which handles each method of the object [12, 16, 23]. For example, the following process models an object that has an integer as a state and provides services to set and read the state.

$$\begin{array}{l} (\nu s)\, (s![0] \,|\, *set?[new].\, s?[old].\, (s![new] \,|\, r![\,]) \\ \qquad\qquad \,|\, *read?[r].\, s?[x].\, (s![x] \,|\, r![x])) \end{array}$$

The channel $s$ is used to store the state. The process above waits to receive request messages on channels *set* and *read*. For example, when a request $\mathit{set}![3]$ arrives, it sets the state to 3 and sends an acknowledgment on $r$.

Since more than one processes may access the above object concurrently, some synchronization is necessary if a process wants to increment the state of the object by first sending a *read* request and then a *set* request. A lock can be implemented using a communication channel. Since a receiver on a channel is blocked until a message becomes available, the locked state can be modeled by the absence of a message in the lock channel, and the unlocked state can be modeled by the presence of a message. The operation to acquire a lock is implemented as the operation to receive a message along the lock channel, and the operation to release the lock as the operation to send a message on the channel. For example, the following process increment the state of the object using a lock channel *lock*.

$$\mathit{lock}?[\,].\,(\nu r)\,(\mathit{read}!\,[r]\,|\,r?[x].\,(\nu r')\,(\mathit{set}![x+1,r']\,|\,r'?[\,].\,\mathit{lock}![\,]))$$

## 3 A Simple Type System

In this section, we introduce a simple type system [7, 30] for our language. It prevents simple programming errors like: $*\mathit{ping}?\,[r].\,r![\,]\,|\,\mathit{ping}![\mathbf{true}]$, which sends a boolean instead of a channel along channel *ping*, and $*\mathit{ping}?\,[r].\,r![\,]\,|\,\mathit{ping}![x,y]$, which sends a wrong number of values on *ping*. Most of the existing programming languages that support concurrency primitives have this kind of type system.

In order to avoid the confusion between booleans and channels and the arity mismatch error above, it is sufficient to classify values into booleans and channels, and to further classify channels according to the shape of transmitted values. We define the syntax of types as follows.

$$\begin{array}{l}\tau ::= \mathbf{bool} \mid [\tau_1,\ldots,\tau_n]\ \mathbf{chan}\\ \sigma ::= \tau \mid \mathbf{proc}\end{array}$$

Type **bool** is the type of booleans, and $[\tau_1,\ldots,\tau_n]$ **chan** is the type of channels that are used for transmitting a tuple of values of types $\tau_1,\ldots,\tau_n$. For example, if $x$ is used for sending a pair of booleans, $x$ must have type $[\mathbf{bool},\mathbf{bool}]$ **chan**. A special type **proc** is the type of processes. The programming errors given in the beginning of this section are prevented by assigning to *ping* a type $[\mathbf{bool}]$ **chan**.

An expression is called *well-typed* if each value is consistently used according to its type. The notion of well-typeness is relative to the assumption about free variables, represented by a *type environment*. It is a mapping form a finite set of variables to types. We use a meta-variable $\Gamma$ to denote a type environment. We write $\emptyset$ for the typing environment whose domain is empty, and write $\mathit{dom}(\Gamma)$ for the domain of $\Gamma$. When $x \notin \mathit{dom}(\Gamma)$, we write $\Gamma, x:\tau$ for the type environment obtained by extending the type environment $\Gamma$ with the binding of $x$ to $\tau$. We write $\Gamma \leq \Gamma'$ when $\mathit{dom}(\Gamma) \supseteq \mathit{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for each $x \in \mathit{dom}(\Gamma')$.

$$\frac{b \in \{\mathbf{true}, \mathbf{false}\}}{\emptyset \vdash b : \mathbf{bool}} \quad (\text{ST-Bool}) \qquad \frac{\Gamma \vdash P : \mathbf{proc} \quad \Gamma \vdash Q : \mathbf{proc}}{\Gamma \vdash P \,|\, Q : \mathbf{proc}} \quad (\text{ST-Par})$$

$$x : \tau \vdash x : \tau \quad (\text{ST-Var})$$

$$\frac{\Gamma' \vdash A : \sigma \qquad \Gamma \leq \Gamma'}{\Gamma \vdash A : \sigma} \quad (\text{ST-Weak}) \qquad \frac{\Gamma, x : \tau \vdash P : \mathbf{proc} \quad \tau \text{ is a channel type}}{\Gamma \vdash (\nu x : \tau)\, P : \mathbf{proc}} \quad (\text{ST-New})$$

$$\emptyset \vdash \mathbf{0} : \mathbf{proc} \quad (\text{ST-Zero}) \qquad \frac{\Gamma \vdash P : \mathbf{proc}}{\Gamma \vdash *P : \mathbf{proc}} \quad (\text{ST-Rep})$$

$$\frac{\Gamma \vdash x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \qquad \Gamma \vdash v_i : \tau_i \ (\text{for each } i \in \{1, \ldots, n\})}{\Gamma \vdash x![v_1, \ldots, v_n] : \mathbf{proc}} \quad (\text{ST-Out})$$

$$\frac{\Gamma \vdash x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \qquad \Gamma, y : \tau_1, \ldots, y : \tau_n \vdash P : \mathbf{proc}}{\Gamma \vdash x?[y_1 : \tau_1, \ldots, y_n : \tau_n].\, P : \mathbf{proc}} \quad (\text{ST-In})$$

$$\frac{\Gamma \vdash v : \mathbf{bool} \qquad \Gamma \vdash P : \mathbf{proc} \qquad \Gamma \vdash Q : \mathbf{proc}}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ \ : \mathbf{proc}} \quad (\text{ST-If})$$

**Fig. 1.** Typing rules for the simple type system

Intuitively, $\Gamma \leq \Gamma'$ means that $\Gamma$ represents a stronger type assumption about variables.

We write $\Gamma \vdash A : \sigma$ if an expression $A$ (which is either a value expression or a process expression) is well-typed and has type $\sigma$ under the type environment $\Gamma$. The relation $\Gamma \vdash A : \sigma$ is defined by the set of inference rules shown in Figure 1.

Most of the rules should be self-explanatory for those who are familiar with type systems for sequential programming languages. The rule (ST-Weak) means that we can replace a type environment with a stronger assumption. It is equivalent to the usual weakening rule for adding an extra type binding to the type environment. We use (ST-Weak) since it is more convenient for extending the type system later. The rule (ST-New) checks that $x$ is indeed used as a channel of the intended type in $P$.

The rule (ST-Out) checks that the destination channel $x$ indeed has a channel type, and that each argument $v_i$ has the type $\tau_i$, as specified by the type of $x$. The rule (ST-In) checks that $x$ has a channel type, and that the continuation part $P$ is well-typed provided that each formal parameter $y_i$ is bound to a value of the type $\tau_i$ as specified by the type of $x$. Those rules are analogous to the rules for function application and abstraction.

The above type system guarantees that if a process is well-typed, there is no confusion between booleans and channels or arity mismatch error.

## 4 A Type System with Input/Output Modes

Even if a program is type-checked in the simple type system in the previous section, the program may still contain a lot of simple programming errors. For example, the ping server in Example 1 may be written as $*ping?\,[r].\,r?[\,].\,\mathbf{0}$ by mistake. Then, clients cannot receive any reply from the server. Similarly, a client of the server may receive a message along *ping* instead of sending a message either by mistake or maliciously. In Example 3, a user of the object may receive a message along the interface channels *set* and *read* instead of sending a message.

We can prevent the above-mentioned errors by classifying the types of channels according to whether the channels can be used for input (receiving a value) or output (sending a value) [21]. We redefine the syntax of types as follows:

$$\begin{array}{l}\tau ::= \mathbf{bool} \mid [\tau_1,\dots,\tau_n]\ \mathbf{chan}_M \\ M \text{ (mode) } ::= !\ \mid ?\ \mid !?\end{array}$$

A mode $M$ denotes for which operations channels can be used. A channel of type $[\tau_1,\dots,\tau_n]\ \mathbf{chan}_M$ can be used for output (input, resp.) only if $M$ contains the output capability ! (the input capability ?, resp.). The wrong ping server $*ping?\,[r].\,r?[\,].\,\mathbf{0}$ is rejected by assigning to *ping* the type $[[\,]\ \mathbf{chan}_!]\ \mathbf{chan}_?$.

As in type systems for sequential programming languages, we write $\tau_1 \leq \tau_2$ when a value of type $\tau_1$ may be used as a value of type $\tau_2$. It is defined as the least reflexive relation satisfying $[\tau_1,\dots,\tau_n]\ \mathbf{chan}_{!?} \leq [\tau_1,\dots,\tau_n]\ \mathbf{chan}_?$ and $[\tau_1,\dots,\tau_n]\ \mathbf{chan}_{!?} \leq [\tau_1,\dots,\tau_n]\ \mathbf{chan}_!$. It is possible to relax the subtyping relation by allowing, for example, $[\tau_1,\dots,\tau_n]\ \mathbf{chan}_!$ to be co-variant in $\tau_1,\dots,\tau_n$ (see [21]). We do not do so in this paper for the sake of simplicity.

The binary relation $\leq$ on type environments is re-defined as: $\Gamma \leq \Gamma'$ if and only if $dom(\Gamma) \supseteq dom(\Gamma')$ and $\Gamma(x) \leq \Gamma'(x)$ for each $x \in dom(\Gamma')$.

The new typing rules are obtained by replacing only the rules (ST-Out) and (ST-In) of the previous type system with the following rules:

$$\frac{\Gamma \vdash x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_! \qquad \Gamma \vdash v_i : \tau_i \text{ for each } i \in \{1,\dots,n\}}{\Gamma \vdash x![v_1,\dots,v_n] : \mathbf{proc}} \qquad \text{(MT-Out)}$$

$$\frac{\Gamma \vdash x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_? \qquad \Gamma, y:\tau_1,\dots,y:\tau_n \vdash P : \mathbf{proc}}{\Gamma \vdash x?[y_1:\tau_1,\dots,y_n:\tau_n].\,P : \mathbf{proc}} \qquad \text{(MT-In)}$$

## 5 A Linear Type System

The type system in Section 4 prevents a ping server from using a reply channel for input, but it does not detect a mistake that the server forgets to send a reply. For example, the process $*ping?\,[r].\,\mathbf{if}\ b\ \mathbf{then}\ r![\,]\ \mathbf{else}\ \mathbf{0}$ forgets to send a reply in the else-branch: Another typical mistake would be to send more than one reply messages: $*ping?\,[r].\,(r![\,] \mid r![\,])$.

$$\frac{\Gamma \vdash P : \mathbf{proc} \qquad \Delta \vdash Q : \mathbf{proc}}{\Gamma \,|\, \Delta \vdash P \,|\, Q : \mathbf{proc}} \quad (\text{LT-PAR}) \qquad \frac{\Gamma \vdash P : \mathbf{proc}}{\omega\Gamma \vdash *P : \mathbf{proc}} \quad (\text{LT-REP})$$

$$\frac{\Gamma_i \vdash v_i : \tau_i \text{ for each } i \in \{1,\dots,n\}}{(x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^0,!^1)}) \,|\, \Gamma_1 \,|\, \cdots \,|\, \Gamma_n \vdash x![v_1,\dots,v_n] : \mathbf{proc}} \quad (\text{LT-OUT})$$

$$\frac{\Gamma, y : \tau_1, \dots, y : \tau_n \vdash P : \mathbf{proc}}{(x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^1,!^0)}) \,|\, \Gamma \vdash x?[y_1 : \tau_1,\dots,y_n : \tau_n].\, P : \mathbf{proc}} \quad (\text{LT-IN})$$

$$\frac{\Gamma \vdash v : \mathbf{bool} \qquad \Delta \vdash P : \mathbf{proc} \qquad \Delta \vdash Q : \mathbf{proc}}{\Gamma \,|\, \Delta \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : \mathbf{proc}} \quad (\text{LT-IF})$$

**Fig. 2.** Typing rules for the linear type system

We can prevent the errors above by further classifying the channel types according to how often channels are used [13]. The syntax of types is redefined as follows:

$$\begin{aligned} &\tau ::= \mathbf{bool} \mid [\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^{m_1},!^{m_2})} \\ &m \text{ (multiplicity) } ::= 0 \mid 1 \mid \omega \end{aligned}$$

Multiplicities $m_1$ and $m_2$ in the channel type $[\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^{m_1},!^{m_2})}$ describes how often the channel can be used for input and output respectively. Multiplicity 0 means that the channel cannot be used at all for that operation, 1 means that the channel should be used once for that operation, and $\omega$ means that the channel can be used for that operation an arbitrary number of times. By assigning to *ping* a type $[[]\ \mathbf{chan}_{(?^0,!^1)}]\ \mathbf{chan}_{(?^\omega,!^0)}$, we can detect programming errors like $*ping?\,[r].\,(r![\,]\,|\,r![\,])$ and $*ping?\,[r].\ \mathbf{if}\ b\ \mathbf{then}\ r![\,]\ \mathbf{else}\ \mathbf{0}$ above.

We define the binary relation $m_1 \leq m_1'$ as the least partial order that satisfies $\omega \leq 0$ and $\omega \leq 1$. The subtyping relation is re-defined as the least reflexive relation satisfying the rule:

$$\frac{m_1 \leq m_1' \qquad m_2 \leq m_2'}{[\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^{m_1},!^{m_2})} \leq [\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^{m_1'},!^{m_2'})}}$$

The subtyping relation allows, for example, a channel of type $[\,]\ \mathbf{chan}_{(?^\omega,!^\omega)}$ to be used as a channel of type $[\,]\ \mathbf{chan}_{(?^1,!^0)}$, but it does not allow a channel of type $[\,]\ \mathbf{chan}_{(?^0,!^1)}$ (which *must* be used once for output) to be used as a channel of type $[\,]\ \mathbf{chan}_{(?^0,!^0)}$ (which must not be used for output).

We re-define $\Gamma \leq \Gamma'$ by: $\Gamma \leq \Gamma'$ if and only if (i) $dom(\Gamma) \supseteq dom(\Gamma')$, (ii) for each $x \in dom(\Gamma')$, $\Gamma(x) \leq \Gamma'(x)$, and (iii) for each $x \in dom(\Gamma)\backslash dom(\Gamma')$, $\Gamma(x)$ is **bool** or a channel type of the form $[\tau_1,\dots,\tau_n]\ \mathbf{chan}_{(?^0,!^0)}$. Note that $x : \tau, y : [\,]\ \mathbf{chan}_{(?^0,!^1)} \leq x : \tau$ does not hold, since the type environment in the lefthand side indicates that $y$ should be used for output.

Typing rules are shown in Figure 2 (Only the modified rules are shown: The other rules are the same as those of the previous type system). Notice the changes in the rules (LT-Out), (LT-In), (LT-Par), etc. In the rules (XX-Par) in the previous type systems, a type environment is shared by sub-processes. The sharing of a type environment is invalid in the linear type system, since the type environment contains information about how often channels are used. For example, if $x$ has type $[\,]$ **chan**$_{(?^0,!^1)}$ both in $P$ and $Q$, $x$ is used *twice* in $P\,|\,Q$, and therefore $x$ should have type $[\,]$ **chan**$_{(?^0,!^\omega)}$. The operation $\Gamma\,|\,\Delta$ in rule (LT-Par) represents this kind of calculation. It is defined by:

$$(\Gamma\,|\,\Delta)(x) = \begin{cases} \Gamma(x)\,|\,\Delta(x) & \text{if } x \in \mathit{dom}(\Gamma)\cap\mathit{dom}(\Delta) \\ \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma)\backslash\mathit{dom}(\Delta) \\ \Delta(x) & \text{if } x \in \mathit{dom}(\Delta)\backslash\mathit{dom}(\Gamma) \end{cases}$$

$$\mathbf{bool}\,|\,\mathbf{bool} = \mathbf{bool}$$

$$([\tau_1,\ldots,\tau_n]\ \mathbf{chan}_{(?^{m_1},!^{m_2})})\,|\,([\tau_1,\ldots,\tau_n]\ \mathbf{chan}_{(?^{m'_1},!^{m'_2})}) = [\tau_1,\ldots,\tau_n]\ \mathbf{chan}_{(?^{m_1+m'_1},!^{m_2+m'_2})}$$

$$m_1+m_2 = \begin{cases} m_2 \text{ if } m_1 = 0 \\ m_1 \text{ if } m_2 = 0 \\ \omega \quad \text{otherwise} \end{cases}$$

The operation $\omega\Gamma$ in rule (LT-Rep) is defined by:

$$(\omega\Gamma)(x) = \omega(\Gamma(x))$$

$$\omega\mathbf{bool} = \mathbf{bool}$$

$$\omega([\tau_1,\ldots,\tau_n]\ \mathbf{chan}_{(?^{m_1},!^{m_2})}) = [\tau_1,\ldots,\tau_n]\ \mathbf{chan}_{(?^{\omega m_1},!^{\omega m_2})}$$

$$\omega m = \begin{cases} 0 \text{ if } m = 0 \\ \omega \text{ otherwise} \end{cases}$$

In rule (T-If), the type environment $\Delta$ is shared between the then-clause and the else-clause because either the then-clause or the else-clause is executed.

We can check that a ping server does not forget to send a reply by type-checking the server under the type environment $\mathit{ping}:[[\,]\ \mathbf{chan}_{(?^0,!^1)}]\ \mathbf{chan}_{(?^\omega,!^0)}$. On the other hand, the wrong server $*\mathit{ping}?\,[r].\,\mathbf{if}\ b\ \mathbf{then}\ r![\,]\ \mathbf{else}\ \mathbf{0}$ fails to type-check under the same type environment: In order for the server to be well-typed, it must be the case that $\mathbf{if}\ b\ \mathbf{then}\ r![\,]\ \mathbf{else}\ \mathbf{0}$ is well-typed under the assumption $r:[\,]\ \mathbf{chan}_{(?^0,!^1)}$, but the else-clause violates the assumption.

Note, however, that in general the type system does not guarantee that a channel of type $[\,]$ **chan**$_{(?^0,!^1)}$ is used for output exactly once. Consider the process: $(\nu y)\,(\nu z)\,(y?[\,].\,z![\,]\,|\,z?[\,].\,(y![\,]\,|\,x![\,]))$. It is well-typed under the type environment $x:[\,]\ \mathbf{chan}_{(?^0,!^1)}$, but the process does not send a message on $x$ because it is deadlocked. This problem is solved by the type system for deadlock-freedom in Section 7.

## 6 A Type System with Channel Usage

As mentioned in Section 2 (Example 3), a channel can be used as a lock. It, however, works correctly only if the channel is used in an intended manner:

When the channel is created, one message should be put into the channel (to model the unlocked state). Afterwards, a process should receive a message from the channel to acquire the lock, and after acquiring the lock, it should eventually release the lock. The linear type system in Section 5 cannot guarantee such usage of channels: Since a lock channel is used more than once, it is given type $[\,]\,\mathbf{chan}_{(?^\omega,!^\omega)}$, which means that the channel may be used in an arbitrary manner. Therefore, the type system cannot detect programming errors like:

$$lock?[\,].\,\langle critical\_section\rangle(lock![\,]\,|\,lock![\,])$$

which allows two processes to acquire the lock simultaneously, and

$$lock?[\,].\,\langle critical\_section\rangle\mathbf{if}\ b\ \mathbf{then}\ lock![\,]\ \mathbf{else}\ \mathbf{0}$$

which forgets to release the lock in the else-clause.

We can prevent the errors above by putting into channel types information about not only how often channels are used but also *in which order* channels are used for input and output. We redefine the syntax of types as follows.

$$\begin{array}{l} \tau ::= \mathbf{bool} \mid [\tau_1,\ldots,\tau_n]\ \mathbf{chan}_U \\ U\ \text{(usages)} ::= 0 \mid \alpha \mid ?.U \mid !.U \mid (U_1\,|\,U_2) \mid U_1 \,\&\, U_2 \mid \mu\alpha.U \end{array}$$

A channel type is annotated with a *usage* [14, 28], which denotes how channels can be used for input and output. Usage 0 describes a channel that cannot be used at all. Usage $?.U$ describes a channel that is first used for input and then used according to $U$. Usage $!.U$ describes a channel that is be first used for output and then used according to $U$. Usage $U_1\,|\,U_2$ describes a channel that is used according to $U_1$ and $U_2$ possibly in parallel. Usage $U_1 \,\&\, U_2$ describes a channel that is used according to either $U_1$ or $U_2$. Usage $\mu\alpha.U$ describes a channel that is used recursively according to $[\mu\alpha.U/\alpha]U$. For example, $\mu\alpha.(0 \,\&\, (!.\alpha))$ describes a channel that can be sequentially used for output an arbitrary number of times.

We often write ? and ! for ?.0 and !.0 respectively. We also write $*U$ and $\omega U$ for $\mu\alpha.(0 \,\&\, (U\,|\,\alpha))$ and $\mu\alpha.(U\,|\,\alpha)$ respectively. Usage $*U$ describes a channel that can be used according to $U$ an arbitrary number of times, while usage $\omega U$ describes a channel that should be used according to $U$ infinitely often.

We can enforce the correct usage of a lock channel by assigning the usage $!\,|\,*?.!$ to it. We can also express linearity information of the previous section: $(?^{m_1}, !^{m_2})$ is expressed by usage $m_1?\,|\,m_2!$ where $1U = U$ and $0U = 0$.

Before defining typing rules, we introduce a subusage relation $U \leq U'$, which means that a channel of usage $U$ can be used as a channel of usage $U'$. Here, we define it using a simulation relation. We consider a reduction relation $U \xrightarrow{\varphi} U'$ on usages, where $\varphi$ is a multiset consisting of ! and ?. It means that the operations described by $\varphi$ can be simultaneously applied to a channel of usage $U$, and the resulting usage becomes $U'$. The reduction relation is defined by the rules in Figure 3. In the figure, $\uplus$ denotes the operator for multiset union. We also define the unary relation $U^{\downarrow}$ as the least relation that satisfies the following rules:

$$!.U \xrightarrow{\{!\}} U \qquad \frac{U_1 \xrightarrow{\varphi_1} U_1' \qquad U2 \xrightarrow{\varphi_2} U_2'}{U_1 \,|\, U_2 \xrightarrow{\varphi_1 \uplus \varphi_2} U_1' \,|\, U_2'} \qquad \frac{U_2 \xrightarrow{\varphi} U_2'}{U_1 \,\&\, U_2 \xrightarrow{\varphi} U_2'}$$

$$?.U \xrightarrow{\{?\}} U \qquad \frac{U_1 \xrightarrow{\varphi} U_1'}{U_1 \,\&\, U_2 \xrightarrow{\varphi} U_1'} \qquad \frac{[\mu\alpha.U/\alpha]U \xrightarrow{\varphi} U'}{\mu\alpha.U \xrightarrow{\varphi} U'}$$

**Fig. 3.** Usage reduction rules

$$0^{\downarrow} \qquad \frac{{U_1}^{\downarrow} \qquad {U_2}^{\downarrow}}{(U_1 \,|\, U_2)^{\downarrow}} \qquad \frac{{U_1}^{\downarrow} \vee {U_2}^{\downarrow}}{(U_1 \,\&\, U_2)^{\downarrow}} \qquad \frac{([\mu\alpha.U/\alpha]U)^{\downarrow}}{\mu\alpha.U^{\downarrow}}$$

Intuitively, $U^{\downarrow}$ means that a channel of usage $U$ need not be used at all. Using the above relations, the subusage relation is defined as follows.

**Definition 2 (subusage relation).** *The subusage relation $\leq$ is the largest relation that satisfies the following two conditions.*

1. *If $U_1 \leq U_2$ and $U_2 \xrightarrow{\varphi} U_2'$, then $U_1 \xrightarrow{\varphi} U_1'$ and $U_1' \leq U_2'$ for some $U_1'$.*
2. *If $U_1 \leq U_2$ and ${U_2}^{\downarrow}$, then ${U_1}^{\downarrow}$.*

For example, $U_1 \& U_2 \leq U_1$ and $! \,|\, ! \leq !.!$ hold. We re-define the subtyping relation so that $[\tau_1, \ldots, \tau_n]\ \mathbf{chan}_U \leq [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{U'}$ if $U \leq U'$. We write $\Gamma_1 \leq \Gamma_2$ if (i)$dom(\Gamma_1) \supseteq dom(\Gamma_2)$, (ii)$\Gamma_1(x) \leq \Gamma_2(x)$ for each $x \in dom(\Gamma_2)$, and (iii)$\Gamma(x)$ is either **bool** or a channel type of the form $[\tau_1, \ldots, \tau_n]\ \mathbf{chan}_U$ with $U \leq 0$ for each $x \in dom(\Gamma_1) \backslash dom(\Gamma_2)$.

The operations $|$ and $\omega$ on types and type environments are similar to those in the previous type system, except that for channel types, they are defined by:

$$([\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{U_1}) \,|\, ([\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{U_2}) = [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{U_1 \,|\, U_2}$$
$$\omega([\tau_1, \ldots, \tau_n]\ \mathbf{chan}_U) = [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{\omega U}$$

The new typing rules are obtained by replacing (LT-Out) and (LT-In) of the previous type system with the following rules:

$$\frac{\begin{array}{c}\Gamma_i \vdash v_i : \tau_i \text{ (for each } i \in \{1, \ldots, n\}) \\ (\Gamma_1 \,|\, \cdots \,|\, \Gamma_n \,|\, \Delta) = \Gamma, x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_U\end{array}}{\Gamma, x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{!.U} \vdash x![v_1, \ldots, v_n] : \mathbf{proc}} \qquad \text{(UT-Out)}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_U, y : \tau_1, \ldots, y : \tau_n \vdash P : \mathbf{proc}}{\Gamma, x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_{?.U} \vdash x?[y_1 : \tau_1, \ldots, y_n : \tau_n].\, P : \mathbf{proc}} \qquad \text{(UT-In)}$$

In rule (UT-In), the assumption $\Gamma, x : [\tau_1, \ldots, \tau_n]\ \mathbf{chan}_U, y : \tau_1, \ldots, y : \tau_n \vdash P :$ **proc** implies that the channel $x$ is used according to $U$ after the input succeeds. So, $x$ is used according to $?.U$ in total. Similar ordering information is taken into account in rule (UT-Out).

*Example 4.* The process $lock?[\,].\,\textbf{if}\ b\ \textbf{then}\ lock![\,]\ \textbf{else}\ \mathbf{0}$ is well-typed under the type environment $b : \textbf{bool}, lock : [\,]\ \textbf{chan}_{?.(!\&0)}$ but not under $b : \textbf{bool}, lock : [\,]\ \textbf{chan}_{?.!}$. It implies that the lock may not be released correctly.

*Example 5.* The wrong CML program in Section 1 is expressed as:

$$\textbf{proc}\ f[x : \textbf{int}, r : [\textbf{int}]\ \textbf{chan}_{!}] = (\nu y)\, y?[n].\, r![n + x].$$

The usage of $y$ is inferred to be ?. Therefore, we know that the process will be blocked on the input on $y$ forever.

## 7 A Type System for Deadlock-Freedom

The type systems presented so far do not guarantee that the ping server eventually returns a reply, that a lock is eventually released, etc. For example, the type system in the previous section accepts the process

$$lock?[\,].\,(\nu x)\,(\nu y)\,(x?[\,].\,y![\,] \,|\, y?[\,].\,(lock![\,] \,|\, x![\,])),$$

which does not release the lock because of deadlock on channels $x$ and $y$.

We can prevent deadlocks by associating with each input (?) or output usage (!) an *obligation level* and a *capability level*.[1] Intuitively, the obligation level of an action denotes the degree of the necessity of the action being executed, while the capability level of an action denotes the degree of the guarantee for the success of the action.

We extend the syntax of types as follows.

$$\begin{aligned}
&\tau ::= \textbf{bool} \mid [\tau_1, \ldots, \tau_n]\ \textbf{chan}_U \\
&U ::= 0 \mid \alpha \mid ?^{t_o}_{t_c}.U \mid !^{t_o}_{t_c}.U \mid (U_1 \,|\, U_2) \mid U_1 \,\&\, U_2 \mid \mu\alpha.U \\
&t\ (\text{level}) ::= \infty \mid 0 \mid 1 \mid 2 \mid \cdots
\end{aligned}$$

The two levels $t_o$ and $t_c$ in $!^{t_o}_{t_c}.U$ denote the obligation level and the capability level of the output action respectively. Suppose that a channel has the usage $!^{t_o}_{t_c}.U$. Its obligation level $t_o$ means that a process can wait for the success of only actions with capability levels of less than $t_o$ before using the channel for output. For example, if $y$ has usage $!^{2}_{0}$ in $x?[\,].\,y![\,]$, then the capability level of the input on $x$ must be 0 or 1. If the obligation level is 0, the channel must be used for output immediately. If the obligation level is $\infty$, arbitrary actions can be performed before the channel is used for output. The capability level $t_c$ means that the success of an output on the channel is guaranteed by a corresponding input action with an obligation level of less than or equal to $t_c$. In other words, some process will use the channel for input before waiting for the success of an action whose obligation level is greater than or equal to $t_c$. If the capability level is $\infty$, the success of the output is not guaranteed. The meaning of the capability and obligation levels of an input action is similar.

[1] They were called *time tags* in earlier type systems for deadlock-freedom [12, 14, 28].

Using the obligation and capability levels, we can prevent cyclic dependencies between communications. For example, recall the example above:

$$\mathit{lock}?[\,].\,(\nu x)\,(\nu y)\,(x?[\,].\,y![\,]\,|\,y?[\,].\,(\mathit{lock}![\,]\,|\,x![\,])),$$

Suppose that the usage of $x$ and $y$ are $?^{t_0}_{t_1}\,|\,!^{t_1}_{t_0}$ and $?^{t_2}_{t_3}\,|\,!^{t_3}_{t_2}$. From the process $x?[\,].\,y![\,]$, we get the constraint $t_1 < t_3$. From the process $y?[\,].\,(\mathit{lock}![\,]\,|\,x![\,])$, we get the constraint $t_3 < t_1$. Therefore, it must be the case that $t_1 = t_3 = \infty$. (Here, we define $t < t$ holds if and only if $t = \infty$.) Since the output on $\mathit{lock}$ is guarded by the input on $y$, the obligation level of the output on $\mathit{lock}$ must also be $\infty$, which means that the lock may not be released.

Typing rules are the same as those for the type system in the previous section, except for the following rules:

$$\frac{\begin{array}{c}\Gamma_i \vdash v_i : \tau_i \text{ (for each } i \in \{1,\dots,n\}) \qquad \Delta \vdash P : \mathbf{proc}\\ (\Gamma_1 \mid \cdots \mid \Gamma_n \mid \Delta) = \Gamma, x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_U \qquad t_{\mathbf{c}} < \Gamma\end{array}}{\Gamma, x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_{!^{t_{\mathbf{c}}}_{t_{\mathbf{o}}}.U} \vdash x![v_1,\dots,v_n].\,P : \mathbf{proc}} \quad \text{(DT-Out)}$$

$$\frac{\Gamma, x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_U, y : \tau_1,\dots,y : \tau_n \vdash P : \mathbf{proc} \qquad t_{\mathbf{c}} < \Gamma}{\Gamma, x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_{?^{t_{\mathbf{c}}}_{t_{\mathbf{o}}}.U} \vdash x?[y_1 : \tau_1,\dots,y_n : \tau_n].\,P : \mathbf{proc}} \quad \text{(DT-In)}$$

$$\frac{\Gamma, x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_U \vdash P : \mathbf{proc} \qquad \mathit{rel}(U)}{\Gamma \vdash (\nu x : [\tau_1,\dots,\tau_n]\ \mathbf{chan}_U)\,P : \mathbf{proc}} \quad \text{(DT-New)}$$

The side condition $t_{\mathbf{c}} < \Gamma$ in the rules (DT-Out) and (DT-In) expresses constraints on obligation and capability levels. It means that $t_{\mathbf{c}}$ must be less than all the obligation levels appearing at the top level (those which are not guarded by ?. and !.).

The side condition $\mathit{rel}(U)$ in the rule (DT-New) means that all the obligation levels and the capability levels in $U$ are consistent. For example, there must not be the case like $?^{\infty}_{0}\,|\,?^{1}_{\infty}$, where there is an input action of capability level 0 but there is no corresponding output action of obligation level 0.

The type system guarantees that any closed well-typed process is deadlock-free in the sense that unless the process diverges, no input or output action with a finite capability level is blocked forever.

*Example 6.* The usage of a lock is refined as $!^{0}_{\infty}\,|\,*?^{\infty}_{t}.!^{t}_{\infty}$. The part $!^{0}_{\infty}$ means that a value must be put into the channel immediately (so as to simulate the unlocked state). The part $?^{\infty}_{t}$ means that any actions may be performed before acquiring the lock and that once a process tries to acquire the lock, the process can eventually acquire the lock. The part $!^{t}_{\infty}$ means that once a process has acquired the lock, it can only perform actions with capability levels less than $t$ before releasing the lock. Suppose that locks $l_1$ and $l_2$ have usages $*?^{\infty}_{1}.!^{1}_{\infty}$ and $*?^{\infty}_{2}.!^{2}_{\infty}$ respectively. Then, it is allowed to acquire the lock $l_2$ first and then acquire the lock $l_1$ before releasing $l_2$: $l_2?[\,].\,l_1?[\,].\,(l_1![\,]\,|\,l_2![\,])$, but it is not allowed to lock $l_1$ and $l_2$ in the reverse order: $l_1?[\,].\,l_2?[\,].\,(l_1![\,]\,|\,l_2![\,])$. Thus, capability and obligation levels for lock channels correspond to the locking levels in [5].

## 8 Putting All Together

In this section, we illustrate how the type systems introduced in this paper may be applied to programming languages. The language we use below does not exist. We borrow the syntax from ML [19], Pict [24], and HACL [15].

First, the ping server in Example 1 can be written as follows:

```
type 'a rep_chan = 'a chan(!o);
proc ping[r: [] rep_chan] = r![];
```
*val ping = ch: ([] rep_chan) chan(∗!c)*

Here, the first line defines an abbreviation for a type. The part `!o` is the channel usage introduced in Section 6 and `o` means that the obligation level introduced in Section 7 is finite. In the second line, the type annotation for `r` asserts that `r` should be used as a reply channel. (In the syntax of ML, `[]` in the type annotation is `unit`.) The third line is the output of the type system. It says that `ping` can be used an arbitrary number of times for sending a reply channel, and it is guaranteed that the channel is received ($c$ means that the capability level is finite) and a reply will eventually come back.

The following program forgets to send a reply in the else-clause:

```
proc ping2[b, r: [] rep_chan] = if b then r![] else 0;
```

Then, the system's output would be:

*Error: r must have type [] rep_chan*
*but it has type [] chan(!&0) in expression "if b then r![] else 0"*

The following program defines a process to create a new lock:

```
type Lock = [] chan(*?c.!o);
proc newlock[r: Lock rep_chan] = (new l)(l![] | r![l]);
```
*val newlock: (Lock rep_chan) chan(∗!c)*

The process `newlock` takes a channel $r$ as an argument, creates a new lock channel, sets its state to the unlocked state, and returns the lock channel through $r$. The system's output says that one can send a request for creating locks an arbitrary number of times, that the request will be eventually received, and that a lock will be sent back along the reply channel.

If a lock is used in a wrong manner, the program will be rejected:

```
(new r)(newlock![r] | r?[l].l?[].0)
```
*Error: l must have type Lock*
*but it has type [] chan(?) in expression "l?[].0"*

Since the lock `l` is not released in the program, the usage of `l` is not consistent with the type `Lock`.

## 9 Conclusion

In this paper, we gave an overview of various type systems for the $\pi$-calculus, from a simple type system to more advanced type systems for linearity, deadlock-freedom, etc. We have mainly discussed the type systems from a programmer's point of view, and focused on explaining how they can help finding of bugs of concurrent programs. Other applications of type systems include formal reasoning about program behavior through process equivalence theories [13, 21, 22, 26, 31], analysis of security properties [1, 8, 9] and optimization of concurrent programs [10, 29].

We think that type systems for concurrent programs are now mature enough to be applied to real programming languages or analysis tools. To apply the type systems, several issues need to be addressed, such as how to let programmers annotate types, how to report type errors, etc. Some type systems have already been applied; Pict [24] incorporates channel types with input/output modes and higher-order polymorphism, and Flanagan and Freund [6] developed a tool for race detection for Java.

Integration with other program verification methods like model checking [3] and theorem proving would be useful and important. Recent type systems [2, 11] suggest one of such directions.

## References

1. M. Abadi. Secrecy by typing in security protocols. *JACM*, 46(5):749–786, 1999.
2. S. Chaki, S. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. of POPL*, pages 45–57, 2002.
3. J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.
4. C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *LNCS*, pages 288–303. Springer-Verlag, 1999.
5. C. Flanagan and M. Abadi. Types for safe locking. In *Proc. of ESOP 1999*, volume 1576 of *LNCS*, pages 91–108, 1999.
6. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proc. of PLDI*, pages 219–232, 2000.
7. S. J. Gay. A sort inference algorithm for the polyadic $\pi$-calculus. In *Proc. of POPL*, pages 429–438, 1993.
8. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*, pages 145–159. IEEE Computer Society Press, 2001.
9. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. of POPL*, pages 81–92, 2002.
10. A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Info. Comput.*, 161:1–44, 2000.
11. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proc. of POPL*, pages 128–141, January 2001.
12. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Prog. Lang. Syst.*, 20(2):436–482, 1998.

13. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Prog. Lang. Syst.*, 21(5):914–947, 1999.
14. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR2000*, volume 1877 of *LNCS*, pages 489–503. Springer-Verlag, August 2000. The full version is available as technical report TR00-01, Dept. Info. Sci., Univ. Tokyo.
15. N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, volume 907 of *LNCS*, pages 137–166. Springer-Verlag, 1995.
16. N. Kobayashi and A. Yonezawa. Towards foundations for concurrent object-oriented programming – types and language design. *Theory and Practice of Object Systems*, 1(4):243–268, 1995.
17. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.
18. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100:1–77, September 1992.
19. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
20. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proc. of POPL*, pages 84–97, 1994.
21. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
22. B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *JACM*, 47(5):531–584, 2000.
23. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, volume 907 of *LNCS*, pages 187–215. Springer-Verlag, 1995.
24. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
25. J. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
26. D. Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999.
27. D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.
28. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, volume 16(3) of *ENTCS*, pages 55–77, 1998.
29. D. T. Turner. The polymorphic pi-calculus: Theory and implementation. PhD Thesis, University of Edinburgh, 1996.
30. V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic $\pi$-calculus. In *CONCUR'93*, volume 715 of *LNCS*, pages 524–538. Springer-Verlag, 1993.
31. N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, volume 1180 of *LNCS*, pages 371–387. Springer-Verlag, 1996.