An Implicitly-Typed Deadlock-Free Process Calculus

Naoki Kobayashi, Shin Saito, and Eijiro Sumii

Department of Information Science, University of Tokyo
{koba, shin, sumii}@is.s.u-tokyo.ac.jp

Abstract. We extend Kobayashi and Sumii's type system for the deadlock-free π-calculus and develop a type reconstruction algorithm. Kobayashi and Sumii's type system helps high-level reasoning about concurrent programs by guaranteeing that communication on certain channels will eventually succeed. It can ensure, for example, that a process implementing a function really behaves like a function. However, because it lacked a type reconstruction algorithm and required rather complicated type annotations, applying it to real concurrent languages was impractical. We have therefore developed a type reconstruction algorithm for an extension of the type system. The key novelties that made it possible are generalization of *usages* (which specifies how each communication channel is used) and a *subusage* relation.

1 Introduction

General Background. With increasing opportunities of distributed programming, static guarantee of program safety is becoming extremely important, because (i) distributed programs are inherently concurrent and exhibit more complex behavior than sequential programs, (ii) it is hard to debug the whole distributed systems, and (iii) distributed programs usually involve many entities, some of which may be malicious. Lack of static guarantee results in unsafe or slow (due to expensive run-time check) program execution. Among various issues of program safety such as security, this paper focuses on problems caused by concurrency, in particular, deadlock (in a broad sense).

Traditional type systems are insufficient to guarantee the correctness of concurrent/distributed programs. Consider the following program of CML [14]:

```
fun f n = let val ch=channel() in recv(ch)+n+1 end;
```

The function `f` creates a new channel `ch` (by `channel()`), waits for a value v from the channel (by `recv(ch)`), and returns $v + n + 1$. Since there is no sender on the channel `ch`, the application `f(1)` is blocked forever. Thus, `f` actually does not behave like a function, but the type system of CML assigns to `f` a function type *int* → *int*.

Our Previous Type Systems for Deadlock-freedom and Their Problem. To overcome problems like above, a number of type systems [6, 11, 18] have been studied through π-calculus [9]. Our type systems for deadlock-freedom [5, 16] are among the most powerful type systems: They can guarantee partial deadlock-freedom in the sense that communication on certain channels will eventually succeed. In addition to the usual meaning of deadlock-freedom where communications are blocked due to some circular dependencies, they also detect the situation like above, where there exists no communication partner from the beginning. Through the guarantee of deadlock-freedom, they can uniformly ensure that functional processes really behave like functions, that concurrent objects will eventually accept a request for method execution and send a reply, and that binary semaphores are really used like binary semaphores (a process that has acquired a semaphore will eventually release it unless it diverges).

In spite of the attractive features of the deadlock-free type systems, however, their applications to real concurrent programming languages have been limited. The main reason is that there was no reasonable type reconstruction algorithm and therefore programmers had to explicitly annotate programs with rather complex types.

Contributions of This Paper. To solve the above-mentioned problem, this paper develops an *implicitly-typed* version of the deadlock-free process calculus and its type reconstruction algorithm. Programmers no longer need to write complex type expressions; Instead, they just need to declare which communication they want to succeed. (Programmers may still want to partially annotate programs with types for documentation, etc.: Our algorithm can be easily modified to allow such partial type annotation.) For example, a process that sends a request to a function server or a concurrent object can be written as $(\nu r)\,(s![arg, r] \,|\, r?^{\mathbf{c}}[x]. \ldots)$. Here, (νr) creates a fresh channel r. $s![arg, r]$ sends a pair $[arg, r]$ to the server through channel s, and in parallel to this, $r?^{\mathbf{c}}[x]. \ldots$ waits on channel r to receive a reply from the server. The **c** attached to ? indicates that this input from r should eventually succeed, i.e., a reply should eventually arrive on r. If the whole system of processes (including the server process) is judged to be well typed in our type system, then it is indeed guaranteed that the input will eventually succeed, unless the whole system diverges.

Our new technical contributions are summarized as follows. (Those who are unfamiliar with our previous type systems can skip the rest of this paragraph.)

- Generalization of the previous type systems for deadlock-freedom — It is not possible to construct a reasonable type reconstruction algorithm for the previous type systems. So, we generalized them by introducing a subusage relation and new usage constructors such as recursive usages and the greatest lower bound of usages (which roughly correspond to the subtype relation, recursive types, and intersection types in the usual type system). A usage [16] is a part of a channel type and describes for which operations (input or output) and in which order channels can and/or must be used. It can be considered an extension of input/output modes [11] and multiplicities [6].

- Constraint-based type reconstruction algorithm — We have developed a type reconstruction algorithm, which inputs an implicitly-typed process and checks whether it is well typed or not. The algorithm is a non-trivial extension of Igarashi and Kobayashi's type reconstruction algorithm [4] for the linear π-calculus [6], where a principal typing is expressed as a pair of a type environment and a set of constraints on type/usage variables.

Limitations of This Paper. The type system and type reconstruction algorithm described in this paper have the following limitations.

- Incompleteness of the type reconstruction algorithm — The algorithm is sound but incomplete: Although it never accepts ill-typed processes, it rejects some well-typed processes. This is just because we want to reject some well-typed but bad processes that may livelock (i.e., diverge with keeping some process waiting for communication forever). So, our algorithm is actually preferable to a complete algorithm (if there is any).
- Naive treatment of time tags — The treatment of time tags and tag relations, which are key features of the deadlock-free type systems [5, 16], is very naive in this paper. As a result, the expressive power is very limited. This is just for clarifying the essence of new ideas of this paper. It is easy to replace the naive treatment of time tags in this paper with the sophisticated one in the previous papers [5, 16] and extend the type reconstruction algorithm accordingly. The resulting deadlock-free process calculus is more expressive than the previous calculi [5, 16], which have already been shown to be expressive enough to encode the simply-typed λ-calculus with various evaluation strategies, semaphores, and typical concurrent objects.

The Rest of This Paper. Section 2 introduces the syntax and operational semantics of processes, and defines what we mean by deadlock. Section 3 gives a generalized type system. Section 4 describes a type reconstruction algorithm. Section 5 discusses related work, and Section 6 concludes this paper. For the space restriction, we omit proofs, some definitions, and details of the type reconstruction algorithm. They are given in the full version of this paper [7].

2 The Syntax and Operational Semantics of Processes

Our process calculus is a subset of the polyadic π-calculus [8]. Each input/output process can be annotated with the programmer's intention on whether or not the communication should succeed. After introducing its syntax and operational semantics, we define what we mean by deadlock.

2.1 Syntax and Operational Semantics of Processes

We first define the syntax of processes. The metavariables x and y_i range over a countably infinite set of variables.

Definition 1 (processes).

$$\begin{array}{lll}
P \ \textit{(processes)} & ::= & \mathbf{0} \mid x!^{b}[v_1,\ldots,v_n].\,P \mid x?^{b}[y_1,\ldots,y_n].\,P \mid (P \,|\, Q) \mid (\nu x)\,P \\
 & & \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \mid *P \\
v \ \textit{(values)} & ::= & \textit{true} \mid \textit{false} \mid x \\
b \ \textit{(annotations)} & ::= & \emptyset \mid \mathbf{c}
\end{array}$$

Notation 2. We write $\tilde{y}$ for a sequence $y_1,\ldots,y_n$. As usual, $\tilde{y}$ in $x?^{b}[\tilde{y}].\,P$ and x in $(\nu x)\,P$ are called bound variables. The other variables are called free variables. We assume that α-conversions are implicitly applied so that bound variables are always different from each other and from free variables. $[\tilde{x} \mapsto \tilde{v}]P$ denotes a process obtained from P by replacing all free occurrences of x_i with v_i. We often write $x!^{b}[\tilde{y}]$ for $x!^{b}[\tilde{y}].\,\mathbf{0}$. We often omit the empty annotation $\emptyset$ and just write $x![\tilde{y}].\,P$ and $x?[\tilde{y}].\,P$ for $x!^{\emptyset}[\tilde{y}].\,P$ and $x?^{\emptyset}[\tilde{y}].\,P$ respectively.

$\mathbf{0}$ denotes inaction. A process $x!^{b}[\tilde{v}].\,P$ sends a tuple $[\tilde{v}]$ on x and then (after the tuple is received by some process) behaves like P. The annotation b expresses the programmer's intention: If it is $\mathbf{c}$, then the programmer expects that the output eventually succeeds, i.e., the tuple is received by some process. A process $x?^{b}[\tilde{y}].\,P$ receives a tuple $[\tilde{v}]$ on x, binds $\tilde{y}$ to $\tilde{v}$, and executes P. $P \,|\, Q$ executes P and Q in parallel, and $(\nu x)\,P$ creates a fresh channel x and executes P. $\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ executes P if v is *true* and executes Q if v is *false*. $*P$ executes infinitely many copies of the process P in parallel.

Remark 3. In an earlier version of this paper [7], we included another annotation **o**, which means that the annotated input/output operation must be executed. We removed it because it is not so useful and also because it complicates the type system.

The operational semantics is fairly standard: It is defined by using two relations: a structural congruence relation and a reduction relation [8].

Definition 4. *The* structural congruence relation $\equiv$ *is the least congruence relation closed under the rules: (i)*$P \,|\, \mathbf{0} \equiv P$, *(ii)*$P \,|\, Q \equiv Q \,|\, P$, *(iii)*$P \,|\, (Q \,|\, R) \equiv (P \,|\, Q) \,|\, R$, *and (iv)*$(\nu x)\,(P \,|\, Q) \equiv (\nu x)\,P \,|\, Q$ *(x not free in Q). The* reduction relation $\longrightarrow$ *is the least relation closed under the rules in Figure 1.*

2.2 Deadlock

We regard deadlock as a state where (i) processes can no longer be reduced, and (ii) a process is trying to perform an input or output operation annotated with **c**, but has not succeeded to do so (because there is no corresponding output or input process). The latter condition is formally defined as follows.

Definition 5. *A predicate Waiting on processes is the least unary relation satisfying the following conditions: (i) Waiting*$(x!^{\mathbf{c}}[\tilde{v}].\,P)$, *(ii) Waiting*$(x?^{\mathbf{c}}[\tilde{y}].\,P)$, *and (iii) Waiting*$(P)$ *implies Waiting*$(P \,|\, Q)$, *Waiting*$(Q \,|\, P)$, *Waiting*$(*P)$, *and Waiting*$((\nu x)\,P)$.

$$x!^{b}[v_1,\ldots,v_n].\,P \mid x?^{b'}[z_1,\ldots,z_n].\,Q \longrightarrow P \mid [z_1 \mapsto v_1,\ldots,z_n \mapsto v_n]Q$$

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \qquad \frac{P \mid *P \mid Q \longrightarrow R}{*P \mid Q \longrightarrow R}$$

$$\frac{P \longrightarrow Q}{(\nu x)\,P \longrightarrow (\nu x)\,Q} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

$$\textbf{if}\ \mathit{true}\ \textbf{then}\ P\ \textbf{else}\ Q \longrightarrow P \qquad \textbf{if}\ \mathit{false}\ \textbf{then}\ P\ \textbf{else}\ Q \longrightarrow Q$$

Fig. 1. Reduction Rules

Definition 6 (deadlock). *A process P is* in deadlock *if (i) there exists no P' such that $P \longrightarrow P'$ and (ii) Waiting(P) holds.*

This definition slightly differs from, but subsumes the usual definition of deadlock, which refers to a state where processes are blocked because of circular dependencies.

Example 7. Both $(\nu x)\,(x?^{\mathbf{c}}[\,].\,\mathbf{0})$ and $(\nu x)\,(\nu y)\,(x?^{\mathbf{c}}[\,].\,y![\,] \mid y?[\,].\,x![\,])$ are in deadlock because the input from x is annotated with **c**, but the input cannot succeed. On the other hand, neither $(\nu x)\,(x?[\,].\,\mathbf{0})$ nor $(\nu x)\,(x![\,] \mid x?^{\mathbf{c}}[\,].\,\mathbf{0})$ is in deadlock.

3 Type System

Now, we give a type system that can guarantee freedom from the deadlock defined in the previous section. The basic idea of the type system is the same as that of our previous type system [16]: We augment ordinary channel types with *usages*, describing for what operations (input or output) and in which order each channel is used. To enable type reconstruction, this paper extends usages with new constructors and a relation between usages.

3.1 Usages

Definition 8 (usages). *The set of usages is given by the following syntax.*

$$\begin{array}{ll} U\ \textit{(usages)} & ::= 0 \mid \alpha \mid O_a.U \mid I_a.U \mid (U_1 || U_2) \mid U_1 \sqcap U_2 \mid \mathbf{rec}\,\alpha.U \mid *U \\ a\ \textit{(attributes)} & ::= \emptyset \mid \mathbf{c} \mid \mathbf{o} \mid \mathbf{co} \end{array}$$

Here, α ranges over a countably infinite set of variables called usage variables.

Notation 9. $\mathbf{rec}\,\alpha.U$ binds α in U. Usage variables that are not bound are called *free* usage variables. $[\alpha \mapsto U']U$ denotes the usage obtained from U by replacing all free occurrences of α with U'. We give a higher precedence to prefixes ($I_a.$, $O_a.$, $\mathbf{rec}\,\alpha.$, and $*$) than to $||$ and $\sqcap$. We also give a higher precedence to $\sqcap$ than to $||$.

0 is the usage of a channel that cannot be used at all. $O_a.U$ denotes the usage of a channel that can be first used for output, and then used according to U. The attribute $\mathbf{c}$ is called a *capability* and $\mathbf{o}$ an *obligation*. If a contains $\mathbf{c}$ (i.e., if a is $\mathbf{c}$ or $\mathbf{co}$), then the output is guaranteed to succeed. If a contains $\mathbf{o}$, then the channel must be used for output (even though the output may not succeed). Similarly, $I_a.U$ denotes the usage of a channel that can be first used for input with attribute a, and then used according to U. $U_1||U_2$ denotes the usage of a channel that can be used according to U_1 by one process and according to U_2 by another process, possibly in parallel. $U_1 \sqcap U_2$ denotes the usage of a channel that can be used according to either U_1 or U_2. For example, if x is a channel of the usage $I_\emptyset.0 \sqcap O_\emptyset.0$, then x can be used either for input or for output, but not for both. $\mathbf{rec}\,\alpha.U$ denotes the usage of a channel that can be used according to the infinite expansion of $\mathbf{rec}\,\alpha.U$ by $\mathbf{rec}\,\alpha.U = [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$. For example, $\mathbf{rec}\,\alpha.I_\emptyset.\alpha$ denotes the usage of a channel that can be used for input an infinite number of times sequentially. $*U$ denotes the usage of a channel that can be used according to U by infinitely many processes. For example, the usage of a binary semaphore is denoted by $O_\mathbf{o}.0||*I_\mathbf{c}.O_\mathbf{o}.0$, meaning that (i) there must be one initial output, (ii) there can be infinitely many input processes, and (iii) each input is guaranteed to eventually succeed (unless the whole process diverges), and it must be followed by output.

Remark 10. One may think that $*U$ can be replaced by $\mathbf{rec}\,\alpha.(\alpha||U)$. For a subtle technical reason, however, we need to distinguish between them.

Example 11. In the CML program given in Section 1, the usage of the channel `ch` is expressed as $I_a.0$. Because there is no output use, a cannot be $\mathbf{c}$ or $\mathbf{co}$, which implies that `recv(ch)` may be blocked forever. For another example, in the process $x?[\,].\,(x![\,]\,|\,x![\,])$, the usage of x can be expressed as $I_{a_1}.(O_{a_2}.0||O_{a_3}.0)$.

The constructors $\sqcap$ and $\mathbf{rec}\,\alpha.U$ are newly introduced in this paper. Although they are not necessary for the previous explicitly-typed calculus [16] (they would only add a little more expressive power), they are crucial for the implicitly-typed calculus in this paper. Suppose that a process P uses a channel x according to a usage U_1 and Q uses x according to U_2. Then, how can we express the usage of x by the process **if** b **then** P **else** Q? With the choice constructor, we can express the most general usage of x as $U_1 \sqcap U_2$. (There is no problem in the case of *type check*: In order to check that **if** b **then** P **else** Q uses x according to U, we just need to check that both P and Q use x according to U.) Similarly, we do need a recursive usage, for example, to perform type reconstruction for a process $*x?[y].\,y![\,].\,x![y]$. Suppose that x is used to communicate a channel that should be used according to U. After receiving the channel y on x, the above process uses y for output, and then send it through x. The channel y will then be used according to U again. So, we have an equation $U = O.U$. With the recursive usage constructor, we can express a solution of this equation as $\mathbf{rec}\,\alpha.O.\alpha$.

$$U \succeq U||0 \qquad U_1||U_2 \succeq U_2||U_1 \qquad \frac{U_1 \succeq V_1 \qquad U_2 \succeq V_2}{U_1||U_2 \succeq V_1||V_2}$$

$$\begin{array}{ccc} U_1 \sqcap U_2 \succeq U_i & (U_1||U_2)||U_3 \succeq U_1||(U_2||U_3) & \\ *U \succeq *U||U & \mathbf{rec}\,\alpha.U \succeq [\alpha \mapsto \mathbf{rec}\,\alpha.U]U & \dfrac{U \succeq V}{*U \succeq *V} \end{array}$$

$$I_{a_1}.U_1||O_{a_2}.U_2||U_3 \longrightarrow U_1||U_2||U_3 \qquad \frac{U_1 \succeq V_1 \qquad V_1 \longrightarrow V_2 \qquad V_2 \succeq U_2}{U_1 \longrightarrow U_2}$$

Fig. 2. Usage Reduction

Usage Reduction. The usage of a channel changes during reduction of a process. For example, a process $x?[\,].x![\,]\,|\,x![\,]$ uses x as $I.O.0||O.0$. After the communication on x, however, the reduced process $x![\,]$ uses x as $O.0$. To express this change of a usage, we introduce a reduction relation on usages. Thus, usages themselves form a small process calculus, which has only one pair of co-actions I and O.

Following the usual reduction semantics of process calculi [8], we define usage reduction by using a structural relation on usages. For technical convenience, however, we do not require that the structural relation is symmetric.

Definition 12. *A usage preorder $\succeq$ and a usage reduction relation $\longrightarrow$ are the least binary relations on usages closed under the rules in Figure 2. $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.*

Usage Reliability. To avoid deadlock, we must require that the usage of each channel must be consistent (called *reliable*) in the sense that each input/output capability is always matched by a corresponding obligation. For example, $I_{\mathbf{c}}.0||O_{\mathbf{o}}.0$ is reliable, but $I_{\mathbf{c}}.0||O_{\mathbf{c}}.0$ is not. To define the reliability of a usage formally, we use the following predicate $ob_{\mathbf{I}}(U)$, which means that the usage U contains an input obligation and that there is no way to discard the obligation.

Definition 13. *Unary predicates $ob_{\mathbf{I}}, ob_{\mathbf{O}} (\subseteq \mathcal{U})$ on usages are defined by:*

$$\begin{aligned} ob_{\mathbf{I}}(U) &\iff \forall U_1.(U \succeq U_1 \Rightarrow \exists a, U_2, U_3.((U_1 \succeq I_a.U_2||U_3) \wedge (a \in \{\mathbf{o}, \mathbf{co}\}))) \\ ob_{\mathbf{O}}(U) &\iff \forall U_1.(U \succeq U_1 \Rightarrow \exists a, U_2, U_3.((U_1 \succeq O_a.U_2||U_3) \wedge (a \in \{\mathbf{o}, \mathbf{co}\}))) \end{aligned}$$

Definition 14 (reliability). *A usage U is* reliable, *written $rel(U)$, if the following conditions hold for every U' with $U \longrightarrow^* U'$.*

1. *If $U' \succeq I_a.U_1||U_2$ and $a \in \{\mathbf{c}, \mathbf{co}\}$, then $ob_{\mathbf{O}}(U_2)$.*
2. *If $U' \succeq O_a.U_1||U_2$ and $a \in \{\mathbf{c}, \mathbf{co}\}$, then $ob_{\mathbf{I}}(U_2)$.*

Remark 15. The reliability of a usage is decidable: It can be reduced to the reachability problem of Petri nets [3].

Subusage. Some usage expresses more general use of a channel than other usages. For example, a channel of the usage $I_\emptyset.0||I_\emptyset.0$ can be used as that of the usage $I_\emptyset.I_\emptyset.0$, because the former usage allows two input operations to be executed in parallel. To express such a relation, we introduce a subusage relation $U_1 \leq U_2$, meaning that a channel of usage U_1 may be used as that of usage U_2.

The introduction of the subusage relation is essential for type reconstruction. For example, the usage of x by a process $x?[\,].\,x!^{\mathbf{c}}[\,]$ can be expressed both as $I_a.0||O_{\mathbf{c}}.0$ and as $I_a.O_{\mathbf{c}}.0$. Thanks to the subusage relation $I_a.0||O_{\mathbf{c}}.0 \leq I_a.O_{\mathbf{c}}.0$, however, we only need to consider the usage $I_a.O_{\mathbf{c}}.0$ during type reconstruction.

We first introduce a relation $a_1 \leq a_2$ between attributes, which means that a channel that should be used for an input/output operation with the attribute a_1 can be used for the operation with the attribute a_2. Note that $\mathbf{c} \leq \emptyset$ holds but $\mathbf{o} \leq \emptyset$ does not: For deadlock-freedom, it is fine not to use capabilities, but it should be disallowed not to fulfill obligations.

Definition 16 (sub-attribute). *The relation $\leq$ is the least partial order satisfying $\mathbf{c} \leq \emptyset$ and $\mathbf{co} \leq \mathbf{o}$.*

Because we have recursive usages, we need to define the subusage relation co-inductively. Because usages themselves are mini-processes, it is natural to define it using a simulation relation.

Definition 17 (usage simulation). *A binary relation $\mathcal{R}(\subseteq \mathcal{U} \times \mathcal{U})$ on usages is called a* usage simulation *if the following conditions are satisfied for each $(U, U') \in \mathcal{R}$:*

1. *If $U' \succeq I_{a'}.U_1'||U_2'$, then there exist U_1, U_2, and a such that (i) $U \succeq I_a.U_1||U_2$, (ii) $U_2\mathcal{R}U_2'$, (iii) $(U_1||U_2)\mathcal{R}(U_1'||U_2')$, and (iv) $a \leq a'$.*
2. *If $U' \succeq O_{a'}.U_1'||U_2'$, then there exist U_1, U_2, and a such that (i) $U \succeq O_a.U_1||U_2$, (ii) $U_2\mathcal{R}U_2'$, (iii) $(U_1||U_2)\mathcal{R}(U_1'||U_2')$, and (iv) $a \leq a'$.*
3. *$ob_{\mathbf{I}}(U)$ implies $ob_{\mathbf{I}}(U')$, and $ob_{\mathbf{O}}(U)$ implies $ob_{\mathbf{O}}(U')$.*
4. *If $U' \longrightarrow U_1'$, then there exists U_1 such that $U \longrightarrow U_1$ and $U_1\mathcal{R}U_1'$.*

The first and second conditions mean that in order for U to simulate U', U must allow any input/output operations that U' allows. The third condition means that U' must provide any obligations that U provides. The fourth condition means that such conditions are preserved even after reductions.

Definition 18. *A* subusage relation $\leq$ *on usages is the largest usage simulation.*

Example 19. $I_{\mathbf{c}}.U \leq 0$ and $I_{\mathbf{c}}.0||I_{\mathbf{c}}.0 \leq I_{\mathbf{c}}.I_{\mathbf{c}}.0$ hold, but neither $I_{\mathbf{o}}.U \leq 0$ nor $I_{\mathbf{o}}.0||I_{\mathbf{c}}.0 \leq I_{\mathbf{c}}.I_{\mathbf{o}}.0$ holds. $U_1 \sqcap U_2 \leq U_i$ holds for any U_1 and U_2.

3.2 Types, Type Environments, and Type Judgment

The syntax of types is defined as follows. The metavariable t ranges over a countable set $\mathbf{T}$ of labels called *time tags*.

Definition 20 (types). $\tau ::= \mathit{bool} \mid [\tau_1, \ldots, \tau_n]^t/U$

$[\tau_1, \ldots, \tau_n]^t/U$ denotes the type of a channel that can be used for communicating a tuple of values of types $\tau_1, \ldots, \tau_n$. The channel must be used according to the usage U. As in the previous type systems, the time tag t is used to control the order between communications on different channels. The allowed order is specified by the following tag ordering.

Definition 21. *A* tag ordering, *written $\mathcal{T}$, is a strict partial order (i.e., a transitive and irreflexive binary relation) on* $\mathbf{T}$.

Intuitively, $s\mathcal{T}t$ means that a process can use capabilities to communicate on a channel tagged with s before fulfilling obligations to communicate on a channel tagged with t. For example, if a channel x has type $[\mathit{bool}]^{t_x}/I_{\mathbf{c}}.0$ and y has type $[\mathit{bool}]^{t_y}/O_{\mathbf{o}}.0$, and if $t_x\mathcal{T}t_y$ holds, then a process can wait to receive a boolean on x before fulfilling the obligation to send a boolean on y.

Type Environment. A type environment is a mapping from a finite set of variables to types. We use a metavariable Γ for a type environment. If $\tau_i = \mathit{bool}$ for each i such that $v_i = \mathit{true}$ or false, then $v_1 : \tau_1, \ldots, v_n : \tau_n$ denotes the type environment Γ such that $\mathit{dom}(\Gamma) = \{v_1, \ldots, v_n\} \backslash \{\mathit{true}, \mathit{false}\}$ and $\Gamma(v_i) = \tau_i$ for each $v_i \in \mathit{dom}(\Gamma)$. We write $\emptyset$ for the type environment whose domain is empty. When $x \notin \mathit{dom}(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Γ' satisfying $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for $y \in \mathit{dom}(\Gamma)$. $\Gamma \backslash \{x_1, \ldots, x_n\}$ denotes the type environment Γ' such that $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \backslash \{x_1, \ldots, x_n\}$ and $\Gamma'(x) = \Gamma(x)$ for each $x \in \mathit{dom}(\Gamma')$.

Type Judgment. A type judgment is of the form $\Gamma; \mathcal{T} \vdash P$. It means that P uses each channel as specified by Γ, and that P obeys the constraints on the order of communications specified by $\mathcal{T}$. For example, let $\Gamma = x : [\,]^{t_x}/I_{\mathbf{co}}.0, y : [\,]^{t_y}/O_{\mathbf{co}}.0$ and $\mathcal{T} = \{(t_x, t_y)\}$. Then, $\Gamma; \mathcal{T} \vdash x?^{\mathbf{c}}[\,].\, y!^{\emptyset}[\,]$ is a valid judgment, but neither $\Gamma; \mathcal{T} \vdash x!^{\mathbf{c}}[\,].\, y!^{\emptyset}[\,]$ nor $\Gamma; \mathcal{T} \vdash y!^{\mathbf{c}}[\,].\, x?^{\emptyset}[\,].\, \mathbf{0}$ is: The process $x!^{\mathbf{c}}[\,].\, y!^{\emptyset}[\,]$ wrongly uses x for output, and $y!^{\mathbf{c}}[\,].\, x?^{\emptyset}[\,].\, \mathbf{0}$ communicates on x and y in a wrong order.

Operations and Relations on Types and Type Environments. Constructors and relations on usages are extended to operations and relations on types and type environments, as defined in Figure 3. Note that binary operations on types are partial: For example, $\mathit{bool}||[\tilde{\tau}]^t/U$ is undefined.

Intuitively, the type environment $\Gamma_1||\Gamma_2$ indicates that channels can be used according to Γ_1 by one process and used according to Γ_2 by another process in parallel. So, if P_1 uses channels according to Γ_1 (i.e., P_1 is well typed under Γ_1) and P_2 uses them according to Γ_2, then $P_1 \,|\, P_2$ uses them according to $\Gamma_1||\Gamma_2$ in total. Similarly, **if** *true* **then** P_1 **else** P_2 and $*P_1$ use channels according to $\Gamma_1 \sqcap \Gamma_2$ and $*\Gamma_1$ respectively.

The relation $t\mathcal{R}\Gamma$ means that a process is allowed to use a capability on a channel tagged with t before fulfilling obligations contained in Γ.

- Unary/binary operations ($\mathbf{op} = ||, \sqcap$)

$$*\mathit{bool} = \mathit{bool} \qquad *[\tilde{\tau}]^t/U = [\tilde{\tau}]^t/{*U} \qquad (*\Gamma)(x) = *(\Gamma(x))$$
$$\mathit{bool}\,\mathbf{op}\,\mathit{bool} = \mathit{bool} \qquad ([\tilde{\tau}]^t/U_1)\,\mathbf{op}\,([\tilde{\tau}]^t/U_2) = [\tilde{\tau}]^t/(U_1\,\mathbf{op}\,U_2)$$
$$(\Gamma_1\,\mathbf{op}\,\Gamma_2)(x) = \begin{cases} \Gamma_1(x)\,\mathbf{op}\,\Gamma_2(x) & \text{if } x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2) \\ \Gamma_i(x) & \text{if } x \in \mathit{dom}(\Gamma_i)\backslash\mathit{dom}(\Gamma_{3-i}) \end{cases}$$

- Subtyping

$$\mathit{bool} \leq \mathit{bool} \qquad [\tilde{\tau}]^t/U_1 \leq [\tilde{\tau}]^t/U_2 \text{ if } U_1 \leq U_2$$

- Unary predicates

$$\mathit{noob}(\mathit{bool}) \qquad \mathit{noob}([\tilde{\tau}]^t/U) \text{ if } U \leq 0$$
$$\mathit{noob}(\Gamma) \text{ if } \mathit{noob}(\Gamma(x)) \text{ for each } x \in \mathit{dom}(\Gamma)$$

- Tag ordering

$$t\mathcal{T}\tau \text{ if } \mathit{noob}(\tau) \vee (\tau = [\tilde{\tau'}]^s/U \wedge t\mathcal{T}s)$$
$$t\mathcal{T}\Gamma \text{ if } t\mathcal{T}(\Gamma(x)) \text{ for each } x \in \mathit{dom}(\Gamma)$$

Fig. 3. Operations and Relations on Types and Type Environments

3.3 Typing Rules

The set of typing rules for deriving valid type judgments are given in Figure 4.

In the rule for **0**, we require that the type environment contains no obligation, because **0** does nothing. As explained in the previous subsection, in the rules for $P\,|\,Q$, **if** b **then** P **else** Q, and $*P$, the type environment of the processes are computed by combining the type environments of their sub-processes with operations $||$, $\sqcap$, and $*$. In the rule for $(\nu x)\,P$, we require that x has a channel type and its usage is reliable.

The rule for output processes is a key rule. The premise $\Gamma, x : [\tilde{\tau}]^t/U; \mathcal{T} \vdash P$ implies that P uses x according to U. Because the process $x!^b[\tilde{v}].\,P$ uses x for output before doing so, the total usage of x is expressed by $O_a.U$. The other variables may be used by P or by a receiver on x, possibly in parallel. The former use is expressed by Γ, while the latter use is by $v_1 : \tau_1 || \cdots || v_n : \tau_n$. Thus, the type environment of the whole process is given by $x : [\tilde{\tau}]^t/O_a.U || v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma$. If the annotation b is $\mathbf{c}$, the input must succeed, hence the condition $a \in \{\mathbf{c}, \mathbf{co}\}$. Moreover, we require the conditions $(\neg \mathit{noob}(v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma)) \Rightarrow a \in \{\mathbf{c}, \mathbf{co}\}$ and $t\mathcal{T}(v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma)$ to enforce the consistency among different channels. The first condition means that if the process P or the tuple $[v_1, \ldots, v_n]$ contains some obligations, i.e., if $\mathit{noob}(v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma)$ does not hold, then the output on x must be guaranteed to succeed (so that it does not block the fulfillment of the obligations). The second condition is required because this output process uses the capability to output on x *before* fulfilling the obligations possibly contained in P and $[v_1, \ldots, v_n]$: Such dependency must be allowed by the tag ordering $\mathcal{T}$. The rule for input processes is similar.

$$\frac{noob(\Gamma)}{\Gamma;\mathcal{T}\vdash \mathbf{0}} \qquad \frac{\Gamma, x:[\tau_1,\ldots,\tau_n]^t/U;\mathcal{T}\vdash P \qquad rel(U)}{\Gamma;\mathcal{T}\vdash (\nu x)\,P}$$

$$\frac{\Gamma_1;\mathcal{T}\vdash P_1 \qquad \Gamma_2;\mathcal{T}\vdash P_2}{\Gamma_1||\Gamma_2;\mathcal{T}\vdash P_1 \mid P_2} \qquad \frac{\Gamma_1;\mathcal{T}\vdash P \qquad \Gamma_2;\mathcal{T}\vdash Q}{(\Gamma_1 \sqcap \Gamma_2)||v\colon bool;\mathcal{T}\vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

$$\frac{\Gamma;\mathcal{T}\vdash P}{*\Gamma;\mathcal{T}\vdash *P} \qquad \frac{\Gamma, x:\tau';\mathcal{T}\vdash P \qquad \tau\leq\tau'}{\Gamma, x:\tau;\mathcal{T}\vdash P}$$

$$\frac{\begin{array}{c}\Gamma, x:[\tau_1,\ldots,\tau_n]^t/U;\mathcal{T}\vdash P\\ b=\mathbf{c}\Rightarrow a\in\{\mathbf{c},\mathbf{co}\} \qquad t\mathcal{T}(v_1:\tau_1||\cdots||v_n:\tau_n||\Gamma)\\ (\neg noob(v_1:\tau_1||\cdots||v_n:\tau_n||\Gamma))\Rightarrow a\in\{\mathbf{c},\mathbf{co}\}\end{array}}{x:[\tau_1,\ldots,\tau_n]^t/O_a.U||v_1:\tau_1||\cdots||v_n:\tau_n||\Gamma;\mathcal{T}\vdash x!^b[v_1,\ldots,v_n].\,P}$$

$$\frac{\begin{array}{c}\Gamma, x:[\tau_1,\ldots,\tau_n]^t/U, y_1:\tau_1,\ldots,y_n:\tau_n;\mathcal{T}\vdash P\\ b=\mathbf{c}\Rightarrow a\in\{\mathbf{c},\mathbf{co}\} \qquad t\mathcal{T}\Gamma \qquad (\neg noob(\Gamma))\Rightarrow a\in\{\mathbf{c},\mathbf{co}\}\end{array}}{\Gamma, x:[\tau_1,\ldots,\tau_n]^t/I_a.U;\mathcal{T}\vdash x?^b[y_1,\ldots,y_n].\,P}$$

Fig. 4. Typing Rules

Example 22. A process $P = *f?[x,r].\,r![x]$ implements the identity function, since it just forwards the argument x to the reply address r. We can obtain the following judgment:

$$\emptyset;\{(t_f,t_y)\}\vdash (\nu f)\,(P \mid (\nu y)\,f!^{\mathbf{c}}[true,y].\,y?^{\mathbf{c}}[z].\,\mathbf{0}).$$

The process $f!^{\mathbf{c}}[true,y].\,\cdots$ calls the function located at f and waits for a reply. t_f and t_y are time tags of channels f and y. The judgment indicates that the caller process can eventually receive a reply.

3.4 Deadlock Freedom Theorem

Theorem 23. *If $\emptyset;\mathcal{T}\vdash P$ and $P \longrightarrow^* Q$, then Q is not in deadlock.*

As in the previous type systems [5, 16], this theorem is proved as a corollary of the subject reduction property and lack of immediate deadlock. A proof is given in the full paper [7]. The intuitive reasons why the deadlock-freedom holds are: (i) each rule correctly estimates the usage of each channel, and (ii) the side condition $rel(U)$ of (T-New) guarantees that each channel is consistently used, and (iii) the tag ordering guarantees that there is no cyclic dependency between different channels.

4 Type Reconstruction

Thanks to the generalization of the previous type systems [5, 16] made in the last section, it is now possible to develop a type reconstruction algorithm. Type reconstruction proceeds in a manner similar to Igarashi and Kobayashi's type reconstruction algorithm for linear π-calculus [4]. We first transform the typing rules into syntax-directed typing rules, so that there is only one applicable rule for each process expression. Then, by reading the syntax-directed rules in a bottom-up manner, we obtain an algorithm for extracting a principal typing. Finally, we decide the typability of a process by solving the constraint part of the principal typing. For lack of space, we explain the algorithm only through an example. The concrete description of the algorithm is given in the full paper [7].

The key properties of our new type system that enabled type reconstruction are (i) there is only one rule for each process constructor except for the subsumption rule (the right rule in the third line in Figure 4, which can be merged with other rules), and (ii) a most general typing can be expressed by using the new usage constructors and subusage relation. The property (i) does not hold for our earlier type system [5] and other type systems that guarantee certain deadlock-freedom properties [2, 15, 18]: They have different rules for input/output on different types of channels.

4.1 Principal Typing

As in Igarashi and Kobayashi's type system [4], a principal typing can be expressed by using constraints. We introduce variables ranging over attributes, usages, and types, and accordingly extend the syntax of attributes, usages, types, and type environments.

A principal typing of a process P is defined as a pair (Γ, C) of an extended type environment and a set of constraints, satisfying the following conditions: (i) Any type judgment obtained by substituting a solution of C for $\Gamma; \mathcal{T} \vdash P$ is valid (i.e., (Γ, C) expresses only valid typings). (ii) Every valid type judgment can be obtained by substituting a solution of C for $\Gamma; \mathcal{T} \vdash P$ (i.e., (Γ, C) expresses all the valid typings). A formal definition is given in the full paper [7].

4.2 Algorithm for Computing a Principal Typing

We can easily eliminate the subsumption rule by combining it with other rules. By reading the resulting syntax-directed rules in a bottom-up manner, we can construct an algorithm for computing a principal typing. For example, we obtain the following rule by combining the rules for $(\nu x)\, P$ and subsumption:

$$\frac{\Gamma; \mathcal{T} \vdash P \qquad (\Gamma(x) = [\tau_1, \ldots, \tau_n]^t / U \wedge \mathit{rel}(U)) \vee x \notin \mathit{dom}(\Gamma)}{\Gamma \backslash \{x\}; \mathcal{T} \vdash (\nu x)\, P}$$

This implies that a principal typing (Γ, C) of $(\nu x)\, P$ can be computed from a principal typing (Γ', C') of P as follows:

$$(\Gamma, C) = \begin{cases} (\Gamma' \backslash \{x\}, C \cup \{\mathit{rel}(\Gamma'(x))\}) & \text{if } x \in \mathit{dom}(\Gamma) \\ (\Gamma', C') & \text{otherwise} \end{cases}$$

Example 24. For the process $(\nu f)\,(P \,|\, (\nu y)\; f!^{\mathbf{c}}[\mathit{true}, y].\, y?^{\mathbf{c}}[z].\,\mathbf{0})$ given in Example 22, the following pair is a principal typing:

$$\begin{aligned}(\emptyset, \{&\rho_f \leq (*([\rho_x, \rho_r]^{t_f}/I_{a_1}.0)||[\mathit{bool}, \rho_r]^{t_f}/O_{a_2}.0), \mathit{rel}(\rho_f), a_2 \in \{\mathbf{c}, \mathbf{co}\}\\ &\rho_y \leq (\rho_r||[\rho_z]^{t_y}/I_{a_4}.0), \mathit{rel}(\rho_y), a_4 \in \{\mathbf{c}, \mathbf{co}\}\\ &\rho_r \leq [\rho_x]^{t_y}/O_{a_3}.0, \mathit{noob}(\rho_z), (\neg \mathit{noob}(\rho_r) \vee a_4 \in \{\mathbf{o}, \mathbf{co}\}) \Rightarrow t_f \mathcal{T} t_y\})\end{aligned}$$

Here, ρ_f, ρ_x, ρ_y, ρ_z, and ρ_r are type variables representing the types of f, x, y, z, and r respectively. The constraints in the first line are those on the uses of the channel f. Because f is used by P as a value of type $*([\rho_x, \rho_r]^{t_f}/I_{a_1}.0)$ and used by the other process as a value of type $[\mathit{bool}, \rho_r]^{t_f}/O_{a_2}.0$, the type ρ_f of f is constrained by $\rho_f \leq *([\rho_x, \rho_r]^{t_f}/I_{a_1}.0)||[\mathit{bool}, \rho_r]^{t_f}/O_{a_2}.0$. The second line shows constraints on the uses of the channel y. The last constraint in the third line comes from the dependency between f and y.

4.3 Constraint Solving

We can decide the typability of a process by reducing the constraints in its principal typing and checking their satisfiability. We reduce the set of constraints on types, those on usages, those on attributes and those on time tags step by step in this order, in a similar (but more complex) manner to Igarashi and Kobayashi's algorithm [4].

The algorithm for reducing constraints on usages is actually incomplete. The completeness is lost in the second step explained in Example 25 below, where a subusage constraint $\alpha \leq U$ is replaced by $\alpha = \mathbf{rec}\,\alpha.U$. As mentioned in Section 1, this is because we want to reject some well-typed but bad processes. (So, we do not want to require the completeness.) The other transformation steps are sound and complete: In those steps, constraints can be transformed into simpler, equivalent constraints. For the efficiency reason, however, our current prototype type inference system use an approximate (sound but incomplete) algorithm also in the third step.

Example 25. Consider the constraint set shown in Example 24. Our algorithm roughly proceeds as follows.

1. Reduce constraints on types: In a subtyping constraint $\tau_1 \leq \tau_2$ and an expression $\tau_1\,\mathbf{op}\,\tau_2$, τ_1 and τ_2 must be identical except for usages. By instantiating type variables so that this condition is met (which is performed by the first-order unification), we get the following constraint set on usages (ρ_f, ρ_y, ρ_r, ρ_x, and ρ_z were instantiated with $[\mathit{bool}, [\mathit{bool}]^{t_y}/\alpha_r]^{t_f}/\alpha_f$, $[\mathit{bool}]^{t_y}/\alpha_y$, $[\mathit{bool}]^{t_y}/\alpha_r$, bool, and bool, respectively):
$$\begin{aligned}\{&\alpha_f \leq *I_{a_1}.0||O_{a_2}.0, \mathit{rel}(\alpha_f), a_2 \in \{\mathbf{c}, \mathbf{co}\}, \alpha_y \leq \alpha_r||I_{a_4}.0, \mathit{rel}(\alpha_y),\\ &a_4 \in \{\mathbf{c}, \mathbf{co}\}, \alpha_r \leq O_{a_3}.0, (\neg \mathit{noob}(\alpha_r) \vee a_4 \in \{\mathbf{o}, \mathbf{co}\}) \Rightarrow t_f \mathcal{T} t_y\}\end{aligned}$$
2. Reduce subusage constraints: From the subusage constraints, we obtain $\alpha_f = *I_{a_1}.0||O_{a_2}.0$, $\alpha_y = O_{a_3}.0||I_{a_4}.0$, and $\alpha_r = O_{a_3}.0$ as a representative solution. By substituting it for the other constraints, we obtain: $\{\mathit{rel}(*I_{a_1}.0||O_{a_2}.0)$, $a_2 \in \{\mathbf{c}, \mathbf{co}\}, \mathit{rel}(O_{a_3}.0||I_{a_4}.0), a_4 \in \{\mathbf{c}, \mathbf{co}\}, (\{a_3, a_4\} \subseteq \{\mathbf{o}, \mathbf{co}\}) \Rightarrow t_f \mathcal{T} t_y\}$.

3. Reduce the other constraints on usages: By reducing the reliability constraints, we obtain: $\{a_1 \not\subseteq \{\mathbf{c}, \mathbf{co}\}, a_1 \in \{\mathbf{o}, \mathbf{co}\}, a_2 \in \{\mathbf{c}, \mathbf{co}\}, a_3 \in \{\mathbf{o}, \mathbf{co}\}, a_4 \in \{\mathbf{c}, \mathbf{co}\}, t_f \mathcal{T} t_y\}$.
4. Reduce the constraints on usage attributes: Start with $a_1 = a_2 = a_3 = \emptyset$, and increment the attributes step by step until the whole constraints are satisfied. In this case, we have $a_1 = \mathbf{o}$, $a_2 = \mathbf{c}$, $a_3 = \mathbf{o}$, and $a_4 = \mathbf{c}$ as a solution, and obtain $t_f \mathcal{T} t_y$.
5. Check whether the remaining constraints on the tag ordering is satisfiable. In this case, we have only the constraint $t_f \mathcal{T} t_y$, which is clearly satisfiable. The process is therefore accepted as a well-typed process.

Recursive usages and greatest lower bounds play an important role in solving usage constraints (the step 2 above). For example, given subusage constraints $\alpha \leq O_a.0$ and $\alpha \leq I_{a'}.\alpha$, we can first transform them into $\alpha \leq O_a.0 \sqcap I_{a'}.\alpha$, and then obtain $\alpha = \mathbf{rec}\, \alpha.(O_a.0 \sqcap I_{a'}.\alpha)$ as a representative solution. This is not always possible without recursive usage and greatest lower bound constructors.

5 Related Work

Several type systems guaranteeing certain deadlock-freedom properties have recently been proposed [1, 2, 5, 13, 15, 16, 18]. As far as we know, however, no type reconstruction algorithm has been developed for them so far. One of the main difficulties of type reconstruction for those type systems is that they use different rules for input/output on different types of channels. We solved that problem by generalizing our previous type systems.

One of the key ideas was to use a process-like term to describe the channel-wise behavior of a process. Similar ideas are found in earlier type systems [17, 10]: For example, Nierstrasz [10] used CCS-like terms as types of concurrent objects and defined subtyping relations.

Our type reconstruction algorithm can be considered a non-trivial extension of Igarashi and Kobayashi's type reconstruction algorithm [4] for the linear π-calculus [6].

6 Conclusion

We have extended our previous type systems for deadlock-freedom [5, 16] and developed its type reconstruction algorithm. A prototype type inference system is available at http://www.yl.is.s.u-tokyo.ac.jp/~shin/pub/.

There remain a number of issues in applying our type system and algorithm to real concurrent programming languages [12, 14], such as whether the type system is expressive enough, how to make the algorithm efficient, and how to present the result of type reconstruction to programmers. We plan to perform experiments using existing CML or Pict programs to answer these questions.

Acknowledgment We would like to thank Atsushi Igarashi for useful comments.

References

1. G. Boudol. Typing the use of resources in a concurrent calculus. In *Proceedings of ASIAN'97*, LNCS 1345, pages 239–253, 1997.
2. K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*, LNCS 1782, pp.180–199, 2000.
3. J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994.
4. A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation*. To appear. A preliminary summary appeared in Proceedings of SAS'97, LNCS 1302, pp.187-201.
5. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.
6. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999. A preliminary summary appeared in Proceedings of POPL'96, pp.358-371.
7. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. Technical Report TR00-01, Dept. Info. Sci., Univ. of Tokyo, January 2000. Available at http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html.
8. R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1993.
9. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100:1–77, September 1992.
10. O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995. A preliminary version appeared in Proceedings of OOPSLA'93, pp.1-15.
11. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
12. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. To appear in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, MIT Press, 2000
13. F. Puntigam. Coordination requirements expressed in types for active objects. In *Proceedings of ECOOP'97*, LNCS 1241, pages 367–388, 1997.
14. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of PLDI'91*, pages 293–305, 1991.
15. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1-2), pages 457–493, 1999.
16. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, *ENTCS* 16(3), pages 55–77, 1998.
17. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, LNCS 817, pages 398–413, 1994.
18. N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, LNCS 1180, pages 371–387, 1996.