Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness

Naoki Kobayashi

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

Abstract. In our previous papers [7,11,23], we presented advanced type systems for the π-calculus, which can guarantee deadlock-freedom in the sense that certain communications will eventually succeed *unless the whole process diverges*. Although such guarantee is quite useful for reasoning about the behavior of concurrent programs, there still remains a weakness that the success of a communication cannot be completely guaranteed due to the problem of divergence. For example, while a server process that has received a request message cannot discard the request, it is allowed to infinitely delegate the request to other processes, causing a *livelock*. In this paper, we show that we can guarantee not only deadlock-freedom but also livelock-freedom, by modifying our previous type systems for deadlock-freedom. The resulting type system guarantees that certain communications will eventually succeed under fair scheduling, no matter whether processes diverge. Moreover, it can also guarantee that some of those communications will succeed within a certain amount of time.

1 Introduction

It is an important and challenging task to statically guarantee the correctness of concurrent programs: Because concurrent programs are more complex than sequential programs (due to dynamic control, non-determinism, deadlock, etc.), it is hard for programmers to debug concurrent programs or reason about their behavior. Recent advanced use of the Internet is further increasing the importance of correctness of concurrent programs: It is now becoming common that programs (which may be written by untrusted or malicious programmers) are distributed through the Internet and they run concurrently at multiple sites, interacting with each other.

Unfortunately, existing concurrent/distributed programming languages or thread libraries provide only limited support for the correctness of concurrent programs. Some of them were proposed as extensions of *typed* functional languages [16,21], but their type systems do not give much information about the behavior of a concurrent program: for example, in CML [21], a term of a function type may not even behave like a function: it may get stuck or return non-deterministic results.

J. van Leeuwen et al. (Eds.): IFIP TCS 2000, LNCS 1872, pp. 365–389, 2000.

To improve the above situation, a number of type systems have been studied through process calculi, just as type systems for functional languages have been studied through λ-calculus. Our type systems [7,11,23] for the π-calculus [13,14] are among the most powerful ones. They can guarantee that every well-typed process is deadlock-free, in the sense that certain communications will eventually succeed unless they diverge. In the most recent type system [11], a programmer just needs to declare which communications they want to succeed, and if the whole process is judged to be well-typed, it is indeed guaranteed that those communications succeed unless the process diverges. For example, in the server-client model, a client process can be written as $(\nu r)\,(s![request, r] \mid r?^{\mathbf{c}}[reply].\,P)$ in the π-calculus-like language of [11]. (νr) creates a fresh communication channel for receiving a reply from the server. $s![request, r]$ sends a request and the channel to the location $\mathtt{s}$ (which is also a channel) of a server. In parallel to this, $r?^{\mathbf{c}}[reply].\,P$ waits for a reply from the server. The annotation $\mathbf{c}$ to ? indicates that a reply should eventually arrive. If the whole system of processes (including the server) is well typed, then a reply is indeed guaranteed to arrive (again, unless the whole system diverges): the server eventually receives the request message, and sends a reply. In this way, the type system can ensure that a process implementing a server really behaves like a server and that a channel implementing a semaphore is really used as a semaphore (a process that has acquired a semaphore will eventually release it). The type systems are reasonably expressive: In the previous paper [7], we have shown that our type system can guarantee deadlock-freedom of the simply-typed λ-calculus with various reduction strategies and typical concurrent objects. Nestmann [15] used our type system to show that his encoding of the choice operator into a choice-free fragment of the π-calculus does not introduce deadlock.

Although the above deadlock-freedom property is quite useful for reasoning about the behavior of concurrent programs, there is still a limitation: the success of a communication is not completely guaranteed because a process may diverge before the communication succeeds. For example, while a server process that has received a request message cannot discard the request, it is allowed to infinitely delegate the request to other processes, causing a *livelock*.

In this paper, we show that we can guarantee livelock-freedom as well as deadlock-freedom, by modifying our previous type systems for deadlock-freedom. The resulting type system guarantees that certain communications will eventually succeed under fair scheduling, *no matter whether processes diverge*. Moreover, it can also guarantee that some of those communications will succeed within a certain amount of time. Surprisingly, this modification to our previous type systems also has a good effect on formalization: Intuitions on types become clearer, and the typing rules become even simpler.

In the rest of this paper, we first introduce our target language and explain what we mean by deadlock and livelock in Section 2. Section 3 reviews basic ideas of our previous type systems for deadlock-freedom. Section 4 introduces our new type system for freedom from livelock and deadlock, and shows its soundness. Section 5 shows that with a minor modification, the type system

can also guarantee that certain communications succeed within a certain time bound. The type system in Section 4 is rather naive and cannot guarantee the livelock-freedom of recursive programs. In Section 6, we show that we can recover the expressive power to some extent by introducing dependent types. Section 7 discusses related work and Section 8 concludes this paper. Due to lack of space, most proofs are omitted: they are found in an accompanying technical report [9]. We do not discuss issues of type check or type reconstruction in this paper. We expect that those issues are basically similar to the case for the deadlock-free calculus [7,11], but complete type reconstruction would be unrealistic for an extension with dependent types.

2 Target Language

This section introduces the target language of our type system and defines deadlock and livelock. The target language is a subset of the polyadic π-calculus [13]. We drop the matching and choice operators from the π-calculus, but keep the other operators. In particular, channels are first-class citizens as in the usual π-calculus, in the sense that they can be dynamically created and passed through other channels. Although this makes it difficult to guarantee deadlock/livelock-freedom, we believe that first-class channels are important in modeling modern concurrent/distributed programming languages. In fact, for example, in concurrent object-oriented programming [25], an object is dynamically created and its reference is passed through messages. Because a reference to a concurrent object corresponds to a record of communication channels [12, 18], channels should be first-class data.

2.1 Syntax

Definition 1 (processes). *The set of processes is defined by the following syntax.*

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

Here, x and y_is range over a countably infinite set of variables.

Notation 2. As usual, $y_1,\ldots,y_n$ in $x?[y_1,\ldots,y_n].\,P$ 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. $[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]P$ denotes a process obtained from P by replacing all free occurrences of $x_1,\ldots,x_n$ with $v_1,\ldots,v_n$. We write $\tilde{x}$ for a sequence $x_1,\ldots,x_n$. $[\tilde{x} \mapsto \tilde{v}]$ and $(\nu\tilde{x})$ abbreviate $[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]$ and $(\nu x_1)\cdots(\nu x_n)$ respectively. We often omit an inaction $\mathbf{0}$ and write $x!^{a}[\tilde{y}]$ for $x!^{a}[\tilde{y}].\,\mathbf{0}$. When attributes are not important,

we omit them and just write $x![\tilde{y}].\,P$ and $x?[\tilde{y}].\,P$ for $x!^a[\tilde{y}].\,P$ and $x?^a[\tilde{y}].\,P$ respectively.

$\mathbf{0}$ denotes inaction. $x!^a[v_1,\ldots,v_n].\,P$ denotes a process that sends a tuple $[v_1,\ldots,v_n]$ on x and then (after the tuple is received by some process) behaves like P. An attribute a expresses the programmer's intention and it does not affect the operational semantics: $a = \mathbf{c}$ means that the programmer wants this output to succeed, i.e., once the output is executed and the tuple is sent, the tuple is expected to be received eventually. There is no such requirement when a is $\emptyset$. $x?^a[y_1,\ldots,y_n].\,P$ denotes a process that receives a tuple $[v_1,\ldots,v_n]$ on x and then behaves like $[y_1 \mapsto v_1,\ldots,y_n \mapsto v_n]P$. If a is $\mathbf{c}$, then this input is expected by the programmer to succeed eventually. $P \mid Q$ denotes a concurrent execution of P and Q, and $(\nu x)\,P$ denotes a process that creates a fresh channel x and then behaves like P. **if** v **then** P **else** Q behaves like P if v is *true* and behaves like Q if v is *false*; otherwise it is blocked forever. $*P$ represents infinitely many copies of the process P running in parallel.

Example 3. A process $*succ?[m,n,r].\,r![m+n]$ behaves as a function server computing the sum of two integers. (For clarity, we extend the language with integers and operations on them here.) It receives a triple consisting of two integers and a channel, and sends the sum of the integers to the channel. A client process can be written like $(\nu y)\,(succ![1,2,y] \mid y?^{\mathbf{c}}[x].\,P)$. The attribute $\mathbf{c}$ of the input process declares that the result should be eventually received.

Example 4. A binary semaphore (or lock) can be implemented by using a channel. Basically, we can regard the presence of a value in the channel as the unlocked state, and the absence of a value as the locked state. Then, creation of a semaphore corresponds to channel creation, followed by output of a null tuple $((\nu x)\,(x![] \mid P))$. The semaphore can be acquired by extracting a value from the channel $(x?[].\,Q)$, and released by putting a value back into the channel $(x![])$. If we want to make sure that the semaphore can be eventually acquired, we can annotate the input as $x?^{\mathbf{c}}[].\,Q$.

2.2 Operational Semantics

The operational semantics is essentially the same as the standard reduction semantics of the π-calculus [13]. For a subtle technical reason, we introduce a structural preorder instead of a structural congruence relation. The only differences from the usual structural congruence $\equiv$ are that $*P \mid P \succeq *P$ does not hold and that $\succeq$ is not closed under output and input prefixes and conditionals.

Definition 5. *The* structural preorder $\succeq$ *is the least reflexive and transitive relation closed under the following rules ($P \equiv Q$ denotes $(P \succeq Q) \wedge (Q \succeq P)$):*

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu x)\,(P \mid Q) \equiv (\nu x)\,P \mid Q (x \text{ not free in } Q)$$

$$*P \succeq *P \mid P \qquad \frac{P \succeq P' \qquad Q \succeq Q'}{P \mid Q \succeq P' \mid Q'}$$

$$\frac{P \succeq Q}{(\nu x)\, P \succeq (\nu x)\, Q} \qquad \frac{P \succeq Q}{*P \succeq *Q}$$

Now we define the reduction relation. Following the operational semantics of the linear π-calculus [10], we define the reduction relation as a ternary relation $P \xrightarrow{l} Q$. l expresses on which channel the reduction is performed: l is either ϵ, which means that the reduction is performed by communication on an internal channel or by the reduction of a conditional expression, or x, which means that the reduction is performed by communication on the free channel x.

Definition 6. *The reduction relation* $\xrightarrow{l}$ *is the least relation closed under the following rules:*

$$x!^{a}[\tilde{v}].\, P \mid x?^{a'}[\tilde{z}].\, Q \xrightarrow{x} P \mid [\tilde{z} \mapsto \tilde{v}]Q$$

$$\frac{P \xrightarrow{l} Q}{P \mid R \xrightarrow{l} Q \mid R} \qquad \frac{P \xrightarrow{x} Q}{(\nu x)\, P \xrightarrow{\epsilon} (\nu x)\, Q}$$

$$\frac{P \xrightarrow{l} Q \qquad l \neq x}{(\nu x)\, P \xrightarrow{l} (\nu x)\, Q} \qquad \frac{P \succeq P' \qquad P' \xrightarrow{l} Q' \qquad Q' \succeq Q}{P \xrightarrow{l} Q}$$

$$\textbf{if } \textit{true} \textbf{ then } P \textbf{ else } Q \xrightarrow{\epsilon} P \qquad \textbf{if } \textit{false} \textbf{ then } P \textbf{ else } Q \xrightarrow{\epsilon} Q$$

Notation 7. We write $P \longrightarrow Q$ if $P \xrightarrow{l} Q$ for some l. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. $P \xrightarrow{l}$ and $P \longrightarrow$ mean $\exists Q.(P \xrightarrow{l} Q)$ and $\exists Q.(P \longrightarrow Q)$ respectively.

2.3 Deadlock-Freedom and Livelock-Freedom

Based on the above operational semantics, we formally define deadlock-freedom and livelock-freedom below. Basically, we regard a process as deadlocked or livelocked if one of its subprocesses is trying to communicate with some process but blocked forever without finding a communication partner. However, not every process that is blocked forever should be regarded as being in a bad state. For example, it should be no problem that a server process waits for a request forever: it just means that no client process sends a request message. It is also fine that an output process remains forever on a channel implementing a semaphore (Example 4), because it means that no process tries to acquire the semaphore. Therefore, we focus our attention on communications annotated with the attribute **c**. A process is considered deadlocked or livelocked if it is trying to perform input or output but blocked forever, *and if the input or output is annotated with* **c**.

We first define deadlock.

Definition 8 (deadlock, deadlock-freedom). *A process P is in* deadlock *if (i) $P \succeq (\nu\tilde{y})\,(x?^{\mathbf{c}}[\tilde{z}].\,Q \mid R)$ or $P \succeq (\nu\tilde{y})\,(x!^{\mathbf{c}}[\tilde{z}].\,Q \mid R)$ and (ii) there is no P' such that $P \longrightarrow P'$. A process P is* deadlock-free *if there exists no Q such that $P \longrightarrow^* Q$ and Q is in deadlock.*

Remark 9. In the usual terminology, the deadlock often refers to a more restricted state, where processes are blocked forever because of *cyclic dependencies on communications*. As the above definition shows, in this paper, the deadlock refers to a state where processes are blocked forever, irrespectively of whether or not there are cyclic dependencies. Our definition of deadlock subsumes the usual, narrower definition of deadlock.

Example 10. $(\nu x)\,(x?^{\mathbf{c}}[\,].\,\mathbf{0})$ is in deadlock because the input from x is annotated with $\mathbf{c}$ but there is no output process. $(\nu x)\,(\nu y)\,(x?^{\mathbf{c}}[\,].\,y![\,] \mid y?[\,].\,x![\,])$ is also deadlocked because the input on x cannot succeed because of cyclic dependencies on communications on x and y. On the other hand, $(\nu x)\,(x?[\,].\,\mathbf{0})$ and $(\nu x)\,(x![\,] \mid x?^{\mathbf{c}}[\,].\,\mathbf{0})$ are not in deadlock.

Example 11. A process $(\nu x)\,(x?^{\mathbf{c}}[\,].\,\mathbf{0} \mid (\nu y)\,(y![x] \mid *y?[z].\,y![z]))$ is deadlock-free. Although the input from x never succeeds, the entire process is never blocked.

The last example shows a weakness of the deadlock-freedom property: Even if a process is deadlock-free, it is not completely guaranteed that communications eventually succeed. The livelock-freedom property defined below requires that communications eventually succeed, no matter whether the process diverges.

To give a reasonable definition of livelock, we need to assume that scheduling is *fair*, in the sense that every communication that is enabled infinitely many times eventually succeeds.

Definition 12 (fair reduction sequence). *A reduction sequence $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$ is fair if the following conditions hold.*

(i) If there exists an infinite increasing sequence $n_1 < n_2 < \ldots$ of natural numbers such that $P_{n_i} \succeq (\nu\tilde{w}_i)\,(x!^{a}[\tilde{v}].\,Q \mid x?^{a_i}[\tilde{y}].\,Q_i \mid R_i)$, then there exists $n \geq n_1$ such that $P_n \succeq (\nu\tilde{w})\,(x!^{a}[\tilde{v}].\,Q \mid x?^{a'}[\tilde{y}].\,Q' \mid R')$ and $(\nu\tilde{w})\,(Q \mid [\tilde{y} \mapsto \tilde{v}]Q' \mid R') \succeq P_{n+1}$.

(ii) If there exists an infinite increasing sequence $n_1 < n_2 < \ldots$ of natural numbers such that $P_{n_i} \succeq (\nu\tilde{w}_i)\,(x?^{a}[\tilde{y}].\,Q \mid x!^{a_i}[\tilde{v}_i].\,Q_i \mid R_i)$, then there exists $n \geq n_1$ such that $P_n \succeq (\nu\tilde{w})\,(x?^{a}[\tilde{y}].\,Q \mid x!^{a'}[\tilde{v}].\,Q' \mid R')$ and $(\nu\tilde{w})\,([\tilde{y} \mapsto \tilde{v}]Q \mid Q' \mid R') \succeq P_{n+1}$.

(iii) If $P_i \succeq (\nu\tilde{w})\,(\mathbf{if}\ v\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2 \mid R)$ and $v = true$ or $false$ for some i, then either the reduction sequence is finite or there exists $n \geq i$ such that $P_n \succeq (\nu\tilde{w})\,(\mathbf{if}\ v\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2 \mid R')$, $(\nu\tilde{w})\,(Q' \mid R') \succeq P_{n+1}$, and $Q' = Q_1$ if $v = true$ and $Q' = Q_2$ otherwise.

Note that by definition, every finite reduction sequence is fair.

Definition 13 (full reduction sequence). *A reduction sequence* $P_1 \longrightarrow P_2 \longrightarrow \cdots$ *is* full *if it is an infinite reduction sequence or a finite reduction sequence ending with* P_n *such that* $P_n \not\longrightarrow$,

Now we define the livelock-freedom property. Intuitively, a process is livelock-free if in any fair reduction sequence, a process trying to perform communication with capability annotation can eventually communicate with another process.

Definition 14 (livelock-freedom). *A process* P_0 *is livelock-free if the following conditions hold for any full fair reduction sequence* $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$:

1. *If* $P_i \succeq (\nu\tilde{w})\,(x!^{\mathbf{c}}[\tilde{v}].\,Q \mid R)$ *for some* $i \geq 0$, *there exists* $n \geq i$ *such that* $P_n \succeq (\nu\tilde{w'})\,(x!^{\mathbf{c}}[\tilde{v}].\,Q \mid x?^{a}[\tilde{y}].\,R_1 \mid R_2)$ *and* $(\nu\tilde{w'})\,(Q \mid [\tilde{y} \mapsto \tilde{v}]R_1 \mid R_2) \succeq P_{n+1}$.
2. *If* $P_i \succeq (\nu\tilde{w})\,(x?^{\mathbf{c}}[\tilde{y}].\,Q \mid R)$ *for some* $i \geq 0$, *there exists* $n \geq i$ *such that* $P_n \succeq (\nu\tilde{w'})\,(x?^{\mathbf{c}}[\tilde{y}].\,Q \mid x!^{a}[\tilde{v}].\,R_1 \mid R_2)$ *and* $(\nu\tilde{w'})\,([\tilde{y} \mapsto \tilde{v}]Q \mid R_1 \mid R_2) \succeq P_{n+1}$.

Again, our definition of livelock may defer from the usual terminology. In our definition, livelock-freedom is a strictly stronger property than deadlock-freedom.

Example 15. The process in Example 11 is deadlock-free but not livelock-free. A process $(\nu x)\,(x![\,] \mid x?^{\mathbf{c}}[\,].\,\mathbf{0}) \mid (\nu y)\,(y![\,] \mid *y?[\,].\,y![\,])$ is livelock-free: Under the fairness assumption, the input from x eventually succeeds.

3 Previous Type Systems for Deadlock-Freedom

The basic idea of our previous type systems for deadlock-freedom [7,11,23] is to extend ordinary channel types [5,24,17] with more precise information on how channels are used. The extra information can be classified into the channel-wise behavior, expressed by using *usages* [11,23], and the inter-channel dependency on the order of communications, expressed by using *time tags* [7]. We first explain usages and time tags, and then sketch the type system.

3.1 Usages

A usage of a channel describes in which order the channel can be used for input and/or output. For example, we denote by 0 the usage of a channel that cannot be used at all, and by $I.O.0$ the usage of a channel that can be first used for input and then used for output. $U_1 \,||\, U_2$ denotes the usage of a channel that can be used according to U_1 by one process and U_2 by another process. $*U$ denotes the usage of a channel that can be used according to U by infinitely many processes. The usage of an input-only channel [17] is expressed as $*I.0$, and that of a linear (use-once) channel [10] is expressed as $I.0 \,||\, O.0$. So, usages are generalization of polarities and multiplicities of the linear π-calculus [10]. A channel used as a binary semaphore (Example 4) has the usage $O.0 \,||\, *I.O.0$.

Capabilities and Obligations. Because we are concerned with whether each input or output succeeds, we associate each I and O with a set of two complementary attributes called *capabilities* (denoted by $\mathbf{c}$) and *obligations* (denoted by $\mathbf{o}$). The capability attribute indicates that the communication is guaranteed to succeed. A guaranteed input ($?^{\mathbf{c}}$) or output operation ($!^{\mathbf{c}}$) in Section 2 is allowed only when I or O is annotated with this input capability. We can refine the usage of a binary semaphore given above as $O.0 \,||\, *I_{\mathbf{c}}.O.0$, to indicate that the semaphore can be eventually acquired. In order for the semaphore to be always acquired, however, the output denoted by O *must* be executed. It is expressed by the obligation attribute. So, the correct usage of a binary semaphore is $O_{\mathbf{o}}.0 \,||\, *I_{\mathbf{c}}.O_{\mathbf{o}}.0$. Similarly, the usage of a linear channel is refined as $O_{\mathbf{co}}.0 \,||\, I_{\mathbf{co}}.0$, meaning that the channel must be used exactly once for input and output, and that the communication always succeeds. The usage of a channel used for client-server connection can be denoted by $*I_{\mathbf{o}}.0 \,||\, *O_{\mathbf{c}}.0$. $*I_{\mathbf{o}}.0$ means that a server must accept infinitely many request messages, and $*O_{\mathbf{c}}.0$ means that clients can successfully send infinitely many request messages.

Channel-Wise Consistency. Creating a channel of some bad usage results in deadlock. For example, creating a channel of usage $I_{\mathbf{c}}.0 \,||\, O.0$ may cause deadlock, because an input must succeed, but an output operation may not be executed. So, we require that the usage of each channel must be consistent (called *reliable*) in the sense that each input/output capability is guaranteed by the corresponding output/input obligation. This condition excludes out a deadlocked process $(\nu x)\, x!^{\mathbf{c}}[\,]$, which uses x according to the inconsistent usage $O_{\mathbf{c}}.0$.

3.2 Time Tags

Controlling the channel-wise behavior is not sufficient to guarantee deadlock-freedom. For example, consider the process $x?^{\mathbf{c}}[\,].\, y![\,] \,|\, y?^{\mathbf{c}}[\,].\, x![\,]$. It uses channels x and y according to a consistent usage $I_{\mathbf{c}}.0 \,|\, O_{\mathbf{o}}.0$ but it is in deadlock. This is because of the following cyclic dependencies between x and y: The condition that the input from x is a capability depends on the condition that the output on x is executed (i.e., is an obligation), which depends on the condition that the input from y succeeds (i.e., is a capability). But it further depends on the output obligation on y, which depends on the input capability on x.

To avoid such cyclic dependencies, we associate each channel with a time tag and maintain the order of time tags. Let t_x be the time tag of a channel x and t_y be that of y. Then, we allow a process to use capabilities on x before fulfilling obligations on y only if there is an ordering $t_x < t_y$. In the above example, $x?^{\mathbf{c}}[\,].\, y![\,]$ is allowed under the ordering $t_x < t_y$ but the other subprocess $y!^{\mathbf{c}}[\,].\, x![\,]$ is disallowed because it tries to use an input capability on y before fulfilling an output obligation on x.

3.3 Typing

By using usages and time tags, we can express the type of a channel in the form $[\tau_1, \ldots, \tau_n]^t/U$. The part $[\tau_1, \ldots, \tau_n]$ means that the channel is used for passing

a tuple of values of types $\tau_1, \ldots, \tau_n$ ($\tau_1, \ldots, \tau_n$ may also be channel types). U is the usage of the channel, and t is the time tag.

A type judgment is of the form $x_1 : \tau_1, \ldots, x_n : \tau_n; \mathcal{R} \vdash P$, where $\mathcal{R}$ is a strict partial order on time tags. It means that P uses $x_1, \ldots, x_n$ according to types $\tau_1, \ldots, \tau_n$, and obeys the order $\mathcal{R}$ in performing communications. We give examples of type judgments.

- $x : []^{t_x}/{*I_{\mathbf{c}}.O_{\mathbf{o}}.0}, y : []^{t_y}/{*I_{\mathbf{c}}.O_{\mathbf{o}}.0}; \{(t_x, t_y)\} \vdash P$: The usage ${*I_{\mathbf{c}}.O_{\mathbf{o}}.0}$ of x and y means that P uses x and y as binary semaphores. $\{(t_x, t_y)\}$ indicates that P acquires x first when it acquires both semaphores at the same time: If it acquired x (i.e., used the input capability on x) first, it would obtain the obligation to release x (i.e., the output obligation on x), so it could not use the capability to acquire the semaphore y before fulfilling the obligation to release x.
- $f : [[int]^{t_r}/O_{\mathbf{o}}.0]^{t_f}/{*I_{\mathbf{o}}.0}; \emptyset \vdash P$: The usage of f means that P tries to receive infinitely many messages from f. The type $[int]^{t_r}/O_{\mathbf{o}}.0$ means that a received value is a channel for which an integer must be sent. So, the above judgment indicates that P behaves like a server providing an integer: P waits to receive a channel repeatedly, and each time it receives a channel, it sends an integer back to the received channel.

The following typing rule for input best illustrates how usages and time tags are enforced by our type system [11]:

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]^{t_x}/U, y_1 : \tau_1, \ldots, y_n : \tau_n; \mathcal{R} \vdash P \qquad (hasob(\Gamma) \vee a = \mathbf{c}) \Rightarrow (b = \mathbf{c} \vee b = \mathbf{co}) \qquad t_x \mathcal{R} \Gamma}{\Gamma, x : [\tau_1, \ldots, \tau_n]^{t_x}/I_b.U; \mathcal{R} \vdash x?^a[y_1, \ldots, y_n].\, P}$$

Because the first condition of the premise implies that P uses x according to U and because $x?^a[y_1, \ldots, y_n].\, P$ uses x for input before that, the total usage of x is of the form $I_b.U$. The condition $hasob(\Gamma)$ means that Γ contains an obligation on some channel, i.e., that an input or output operation must be performed on some channel. If so, the input from x must succeed. The attribute b should therefore contain the capability attribute. The last condition $t_x \mathcal{R} \Gamma$ means that if Γ contains an obligation on a channel, t_x should be less than the time tag of that channel.

The following rule for channel creation makes sure that only channels of consistent usages are created.

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]^{t_x}/U; \mathcal{R} \vdash P \qquad reliable(U)}{\Gamma; \mathcal{R} \vdash (\nu x)\, P}$$

In this manner, it is enforced that each channel is consistently used and that there is no cyclic dependency on communications through different channels. Thus, every closed well-typed process is guaranteed to be deadlock-free (see [7, 11] for formal proofs).

4 Type System for Livelock-Freedom

This section introduces our new type system that can guarantee the livelock-freedom property. We first analyze weaknesses of our previous type systems and explain how to remove them (in Section 4.1). Then we define our new type system and show that it is sound in the sense that every closed well-typed process is livelock-free.

4.1 Basic Ideas

As mentioned in Section 1, a weakness of the type systems for deadlock-freedom [7,11,23] is that it cannot completely guarantee the success of a communication because of the problem of divergence. In fact, the process $(\nu x)\,(x?^{\mathbf{c}}[\,].\,\mathbf{0} \mid (\nu y)\,(y![x] \mid *y?[z].\,y![z]))$ in Example 11 is well typed [11, 23]: We can assign to x a type $[\,]^{t_x}/(I_{\mathbf{c}}.0 \,||\, O_{\mathbf{o}}.0)$ and to y a type $[[\,]^{t_x}/O_{\mathbf{o}}.0]^{t_y}/(O_{\mathbf{c}}.\mathbf{0} \,||\, *I_{\mathbf{o}}.O_{\mathbf{c}}.\mathbf{0})$. The type of y demands that any process that has received a channel from y must send a value to it, or delegate an obligation to do so to other processes (by forwarding the received channel). The above process $*y?[z].\,y![z]$ certainly obeys this requirement: it forwards the received channel z to y, and according to the type of y, a receiver of z is supposed to fulfill the obligation to send a value to z. However, the receiver is actually this process itself and therefore the obligation is never fulfilled.

The above problem comes from the fact that a received obligation x of type $[\,]^{t_x}/O_{\mathbf{o}}.0$ can be just forwarded as the same obligation of type $[\,]^{t_x}/O_{\mathbf{o}}.0$. Actually, however, because it takes some time to forward x, it should forward x as the obligation that should be fulfilled within a *shorter* time limit. This observation leads us to replace the binary information on whether or not an obligation exists with a time limit within which an obligation must be fulfilled. Absence of an obligation is the special case where the time limit is infinite. Similarly, the capability attribute is also replaced by the maximum amount of time required before the communication succeeds. Thus, a usage $O_a.0$ is replaced by $O^{t_o}_{t_c}.0$, which means that an output operation must be executed within time t_o (by an output or input operation being executed, we mean that a process becomes ready to output or input a value, not that a process succeeds to find the communication partner and complete the communication), and if the output is executed, it is guaranteed to succeed within time t_c.

Now, let us reconsider the process $(\nu x)\,(x?^{\mathbf{c}}[\,].\,\mathbf{0} \mid (\nu y)\,(y![x] \mid *y?[z].\,y![z]))$. Suppose that the channel y carries a channel for which an output must be executed within time t_o. Then, when the process $*y?[z].\,y![z]$ receives x, some time has already been spent for the communication on y. So, the process must fulfill the obligation to output on x within the time t_o *minus* the time spent for the communication on y. It therefore cannot forward x through y, which may require another time of length t_o until the output is performed. In this manner, infinite delegations of an obligation is forbidden, and livelock-freedom is guaranteed as a result.

The above change has also a good effect on formalization. An unpleasant point about the previous type systems was that the meaning of a channel type is not completely clear by itself because of time tags: The meaning of the time tags in a channel type is only determined by the order relative to time tags attached to other channel types. Now, the time tags are no longer required (actually they have been integrated into time bounds of capabilities and obligations). For example, the dependency between x and y in $x?[\,].\,y![\,]$ is captured by the condition "The input from x should succeed within a time limit shorter than the time limit of the fulfillment of an obligation to output on y." This condition can be expressed in the typing rules more neatly than the corresponding condition on time tags in the previous type systems [11].

4.2 Time Quantum

Now we define our type system. We first introduce *time quantums*, which are used for giving time bounds of capabilities and obligations. We could define a time quantum simply as a natural number, expressing the number of reduction steps. To keep the flexibility, however, we use the following slightly more complicated definition.

Definition 16 (time quantum). *The set of time quantums, ranged over by t, is defined by:*

$$t ::= 0 \mid \mathbf{t} \mid t_1 + t_2 \mid n \cdot t$$

Here, $\mathbf{t}$ ranges over the carrier set $\mathbf{T}$ of a well-founded order $\mathcal{T} = (\mathbf{T}, \ll)$. An element of $\mathbf{T}$ is called a time unit*. We assume that $\mathbf{T}$ contains the least time unit $\mathbf{t_{min}}$ and the greatest time unit $\mathbf{t}_\infty$. n ranges over the set* **Nat** *of natural numbers.*

Intuitively, $\mathbf{t}_1 \ll \mathbf{t}_2$ means that $\mathbf{t}_1$ is sufficiently shorter than $\mathbf{t}_2$ so that the summation of any finite number of $\mathbf{t}_1$s is shorter than $\mathbf{t}_2$. $\mathbf{t_{min}}$ represents the time required for one step reduction of a process. $\mathbf{t}_\infty$ represents an infinite length of time. These intuitions are expressed in the following definitions of the semantics of a time quantum and the order between time quantums.

Definition 17 (semantics of time quantum). *Let t be a time quantum. A mapping $[\![t]\!]$ from $\mathbf{T}$ to* **Nat** *is defined by:*

$$[\![0]\!](\mathbf{t}) = 0 \qquad [\![\mathbf{t}]\!](\mathbf{t}') = \begin{cases} 1 \text{ if } \mathbf{t} = \mathbf{t}' \\ 0 \text{ otherwise} \end{cases}$$

$$[\![t_1 + t_2]\!](\mathbf{t}) = [\![t_1]\!](\mathbf{t}) + [\![t_2]\!](\mathbf{t}) \qquad [\![n \cdot t]\!](\mathbf{t}) = n \times [\![t]\!]\mathbf{t}$$

Remark 18. We do not distinguish between time quantums whose semantics are the same. For example, we identify $(\mathbf{t}_1 + \mathbf{t}_2) + (\mathbf{t}_2 + \mathbf{t}_3)$ with $\mathbf{t}_1 + 2 \cdot \mathbf{t}_2 + \mathbf{t}_3$. With this identification, every time tag can be expressed in a normal form $\Sigma_{\mathbf{t}_i \in \mathbf{T}} n_i \cdot \mathbf{t}_i$.

Definition 19. *The binary relation $\leq$ on time quantums is defined by: $t_1 \leq t_2$ if and only if either (i) $[\![t_2]\!](\mathbf{t}_\infty) > 0$, or (ii) for each $\mathbf{t} \in \mathbf{T}$, either $[\![t_1]\!](\mathbf{t}) \leq [\![t_2]\!](\mathbf{t})$ or there exists $\mathbf{t}'$ such that $[\![t_1]\!](\mathbf{t}') < [\![t_2]\!](\mathbf{t}')$ and $\mathbf{t} \ll \mathbf{t}'$. We write $t_1 < t_2$ if $(t_1 \leq t_2) \wedge \neg(t_2 \leq t_1)$*

It is trivial that the binary relation $\leq$ on time tags is a preorder.

Example 20. Suppose $\mathbf{t}_1 \ll \mathbf{t}_2 \ll \mathbf{t}_\infty$. Then, $n \cdot \mathbf{t}_1 < \mathbf{t}_2$ for any $n \in \mathbf{Nat}$. $\mathbf{t}_\infty + t_1 \leq \mathbf{t}_\infty + t_2$ holds for any time quantums t_1 and t_2. So, $\mathbf{t}_\infty + t$ is essentially equivalent to $\mathbf{t}_\infty$.

The following lemma states an important property to guarantee livelock-freedom. It follows from the fact that the set of time units is well-founded.

Lemma 21. *The set of time quantums is well-founded with respect to $<$, i.e., there is no infinite decreasing sequence $t_0 > t_1 > t_2 > \cdots$.*

4.3 Usages and Types

As mentioned in Section 4.1, we replace the capability/obligation attributes of a usage with two time quantums t_o, t_c and remove the time tag from a channel type.

Definition 22 (usages). *The set $\mathcal{U}$ of usages is given by the following syntax.*

$$U ::= 0 \mid \alpha^{t_o}_{t_c}.U \mid (U_1 \,||\, U_2) \mid *U$$
$$\alpha ::= I \mid O$$

Notation 23. We give a higher precedence to prefixes ($\alpha^{t_o}_{t_c}.$ and $*$) than to $||$. So, $I^{t_o}_{t_c}.U_1 \,||\, U_2$ means $(I^{t_o}_{t_c}.U_1) \,||\, U_2$, not $I^{t_o}_{t_c}.(U_1 \,||\, U_2)$. We sometimes write $\overline{\alpha}$ for I or O. It denotes O when $\alpha = I$ and I when $\alpha = O$.

$O^{t_o}_{t_c}.U$ denotes the usage of a channel that can be first used for output, and then used according to U. Intuitively, t_o means that an output process must be executed within time t_o. If $\mathbf{t}_\infty \leq t_o$, the output need not be performed. t_c means that the output is guaranteed to succeed within time t_c after it is executed. More precisely, t_c is the time limit within which a communication partner is found and the communication starts: It takes $\mathbf{t_{min}}$ more until the communication is completed. If $\mathbf{t}_\infty \leq t_c$, then the output is not guaranteed to succeed. $I^{t_o}_{t_c}.U$ denotes the usage of a channel that must be first used for input within time t_o, and then used according to U if the input succeeds. The input is guaranteed to succeed within time t_c. The meaning of the other usage constructors is the same as that in the previous type systems (see Section 3).

Definition 24 (types). *The set of types is given by the following syntax.*

$$\tau ::= \mathit{bool} \mid [\tau_1, \ldots, \tau_n]/U$$

$[\tau_1, \ldots, \tau_n]/U$ 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.

We introduce several operations on usages and types. $\boxed{t}U$ represents the usage of a channel that is used according to U *after a delay of at most* t.

Definition 25. *A unary operation* $\boxed{t}$ *on usages is defined inductively by:*

$$\boxed{t}0 = 0 \qquad \boxed{t}\alpha^{t_o}_{t_c}.U = \alpha^{t_o+t}_{t_c}.U$$
$$\boxed{t}(U_1 \,||\, U_2) = \boxed{t}U_1 \,||\, \boxed{t}U_2 \qquad \boxed{t}(*U) = *\boxed{t}U$$

Constructors and operations on usages are extended to operations on types.

Definition 26. *Operations* $||, *, \boxed{t}$ *on types are defined by:*

$$bool \,||\, bool = bool \qquad ([\tilde{\tau}]/U_1) \,||\, ([\tilde{\tau}]/U_2) = [\tilde{\tau}]/(U_1 \,||\, U_2)$$
$$*bool = bool \qquad *[\tilde{\tau}]/U_1 = [\tilde{\tau}]/*U_1$$
$$\boxed{t}bool = bool \qquad \boxed{t}[\tilde{\tau}]/U_1 = [\tilde{\tau}]/\boxed{t}U_1$$

4.4 Reliability of Usages

As in the type systems for deadlock-freedom (see Section 3), the usage of each channel must be consistent (reliable) in the sense that each input/output capability is guaranteed by an output/input obligation. For example, consider a usage $I^{\mathbf{t}_\infty}_{t_c}.U_1 \,||\, O^{t_o}_{\mathbf{t}_\infty}.U_2$. In order for the success of input to be guaranteed, it must be the case that $t_o \leq t_c$. This consistency should be preserved during the whole computation. After communication on a channel of usage $I^{\mathbf{t}_\infty}_{t_c}.U_1 \,||\, O^{t_o}_{\mathbf{t}_\infty}.U_2$ happens, the channel is used according to $U_1 \,||\, U_2$. So, $U_1 \,||\, U_2$ must also be consistent. To state such a condition, we first define *reduction* of a usage.

Definition 27. $\cong$ *is the least congruence relation satisfying the following rules:*

$$U \cong U \,||\, 0 \qquad *0 \cong 0$$
$$U_1 \,||\, U_2 \cong U_2 \,||\, U_1 \qquad (U_1 \,||\, U_2) \,||\, U_3 \cong U_1 \,||\, (U_2 \,||\, U_3)$$
$$*U \cong *U \,||\, U \qquad *(U_1 \,||\, U_2) \cong *U_1 \,||\, *U_2$$
$$**U \cong *U$$

Definition 28 (usage reduction). *A binary relation* $\longrightarrow$ *on usages is the least relation closed under the following rules: (i)*$I^{t_o}_{t_c}.U_1 \,||\, O^{t'_o}_{t'_c}.U_2 \,||\, U_3 \longrightarrow U_1 \,||\, U_2 \,||\, U_3$*, and (ii)*$U_1 \longrightarrow U_2$ *if* $U_1 \cong U'_1$*,* $U'_1 \longrightarrow U'_2$*, and* $U'_2 \cong U_2$*.*

Now, a usage U is defined to be reliable if after any reduction steps, whenever it contains an input/output capability (i.e., it is structurally congruent to $I^{t_o}_{t_c}.U_1 \,||\, U_2$), it contains an output/input obligation of an equal or shorter time bound. We first define a predicate to judge whether a usage contains an obligation, and then give a formal definition of the reliability.

Definition 29 (obligation). *The relation* $ob_I, ob_O (\subseteq \mathcal{U})$ *between a time quantum and a usage are defined by:* $ob_\alpha(t, U)$ *if and only if* $\mathbf{t}_\infty \leq t$ *or* $\exists t_o, t_c, U_1, U_2.((U \cong \alpha^{t_o}_{t_c}.U_1 \,||\, U_2) \wedge (t_o \leq t))$.

Definition 30 (reliability). *U is* reliable*, written $rel(U)$, if $ob_\alpha(t_c, U_2)$ holds whenever $U \longrightarrow^* \cong \overline{\alpha}^{t_o}_{t_c}.U_1 \,||\, U_2$. A type τ is reliable, written $rel(\tau)$, if it is of the form $[\tilde{\tau}]/U$ and $rel(U)$ holds.*

Remark 31. The above definitions of ob_α and the reliability do not completely match the intuition on usages. Consider a usage $U = O^{\mathbf{t}}_{\mathbf{t}_\infty}.\mathbf{0} \,||\, *I^{\mathbf{t}_\infty}_{\mathbf{t}}.O^{\mathbf{t}}_{\mathbf{t}_\infty}.\mathbf{0}$ of a binary semaphore. According to the above definitions, U is reliable. Actually, however, an input operation may not succeed within time $\mathbf{t}$: Although an output operation is always executed within $\mathbf{t}$, the input may be delayed because of other input operations. If another process succeeds to input from the channel, it must wait for another period of length $\mathbf{t}$. So, U should not be reliable. (Under the fair scheduling, every input succeeds after a finite number of output operations. So, according to the intuition, $O^{\mathbf{t}}_{\mathbf{t}_\infty}.0 \,||\, *I^{\mathbf{t}_\infty}_{\mathbf{t}'}.O^{\mathbf{t}}_{\mathbf{t}_\infty}.0$ should be reliable if $\mathbf{t} \ll \mathbf{t}'$.) For the guarantee of livelock-freedom, however, the above definition of the reliability is sufficient and much simpler. We will redefine the reliability in Section 5 to guarantee time-boundedness.

4.5 Subusage and Subtyping

The subusage relation defined below allows one usage to be viewed as another usage. For example, consider the usage $I^{\mathbf{t}_\infty}_{t_c}.0 \,||\, I^{\mathbf{t}_\infty}_{t_c}.0$ of a channel that can be used twice for input, possibly in parallel. Because the input is not an obligation, it should be allowed to use the channel for input just once for now, and use once more only after the input succeeds, as expressed by $I^{\mathbf{t}_\infty}_{t_c}.I^{\mathbf{t}_\infty}_{t_c}.0$. Such a relation between usages is expressed by $I^{\mathbf{t}_\infty}_{t_c}.0 \,||\, I^{\mathbf{t}_\infty}_{t_c}.0 \le I^{\mathbf{t}_\infty}_{t_c}.I^{\mathbf{t}_\infty}_{t_c}.0$.

Definition 32. *The* subusage *relation $\le$ on usages is the least reflexive and transitive relation closed under the following rules:*

$$\frac{U \cong U'}{U \le U'} \quad (\textsc{SubU-Cong}) \qquad \frac{\mathbf{t}_\infty \le t_o}{\alpha^{t_o}_{t_c}.U \le 0} \quad (\textsc{SubU-Zero})$$

$$\frac{U_1 \le U_1' \qquad U_2 \le U_2'}{U_1 \,||\, U_2 \le U_1' \,||\, U_2'} \quad (\textsc{SubU-Par}) \qquad \frac{U \le U'}{*U \le *U'} \quad (\textsc{SubU-Rep})$$

$$\alpha^{t_o}_{t_c}.U_1 \,||\, \boxed{t_o + t_c + \mathbf{t_{min}}} U_2 \le \alpha^{t_o}_{t_c}.(U_1 \,||\, U_2) \quad (\textsc{SubU-Delay})$$

$$\frac{U \le U' \qquad t_o' \le t_o \qquad t_c \le t_c'}{\alpha^{t_o}_{t_c}.U \le \alpha^{t_o'}_{t_c'}.U'} \quad (\textsc{SubU-Act})$$

The rule (SubU-Zero) indicates that a channel with no obligation need not be used at all. The rule (SubU-Delay) allows some usage to be delayed until a communication succeeds. In the usage $\alpha^{t_o}_{t_c}.(U_1 \,||\, U_2)$, an obligation contained in U_2 is delayed until the operation α (which is I or O) is executed and it succeeds. Because the time t_o may pass before it is executed and another time period of length t_c may pass before it is enabled, the time $t_o + t_c + \mathbf{t_{min}}$ may be required before obligations in U_2 is fulfilled. It is taken into account by the operation

$\boxed{t_o + t_c + \mathbf{t_{min}}}$ in the lefthand side. The rule (SUBU-ACT) means that it is safe to estimate the time bound of an obligation to be shorter than the actual time bound, while the time bound of a capability should be estimated to be longer than the actual bound.

The subusage relation can be extended to the following subtyping relation.

Definition 33 (subtyping). *A subtyping relation $\leq$ is the least relation closed under the rules: (i)$bool \leq bool$, and (ii) $[\tilde{\tau}]/U \leq [\tilde{\tau}]/U'$ if $U \leq U'$.*

Definition 34. *$noob(\tau)$ if $\tau = bool$ or $\tau = [\tilde{\tau}]/U$ and $U \leq 0$.*

4.6 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. We use a sequence $v_1 : \tau_1, \ldots, v_n : \tau_n$ to denote the type environment Γ such that $dom(\Gamma) = \{v_1, \ldots, v_n\} \backslash \{true, false\}$ and $\Gamma(v_i) = \tau_i$ for each $v_i \in dom(\Gamma)$. In the sequence, if $v_i = true$ or $v_i = false$, τ_i must be $bool$. (So, the sequence $x : \tau, true : bool$ denotes the same type environment as $x : \tau$. The sequence $true : [\tau]/U$ is invalid.) We write $\emptyset$ for the type environment whose domain is empty. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Γ' satisfying $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for $y \in dom(\Gamma)$. $\Gamma \backslash \{x_1, \ldots, x_n\}$ denotes the type environment Γ' such that $dom(\Gamma') = dom(\Gamma) \backslash \{x_1, \ldots, x_n\}$ and $\Gamma'(x) = \Gamma(x)$ for each $x \in dom(\Gamma')$.

The operations on types are pointwise extended to those on type environments as follows.

Definition 35 (operations on type environments). *The operations $||, *, \boxed{t}$ on types are defined by:*

$$dom(\Gamma_1 \,||\, \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$$

$$(\Gamma_1 \,||\, \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \,||\, \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \backslash dom(\Gamma_1) \end{cases}$$

$$dom(*\Gamma) = dom(\Gamma) \qquad (*\Gamma)(x) = *(\Gamma(x))$$
$$dom(\boxed{t}\Gamma) = dom(\Gamma) \qquad (\boxed{t}\Gamma)(x) = \boxed{t}(\Gamma(x))$$

Definition 36. *Γ is reliable, written $rel(\Gamma)$, if $rel(\Gamma(x))$ holds for each $x \in dom(\Gamma)$.*

Definition 37. *A binary relation $\leq$ on type environments is defined by: $\Gamma_1 \leq \Gamma_2$ if and only 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) $noob(\Gamma_1(x))$ for each $x \in dom(\Gamma_1) \backslash dom(\Gamma_2)$.*

$$\emptyset \vdash \mathbf{0} \quad \text{(T-Zero)} \qquad \frac{\Gamma, x:[\tau_1,\ldots,\tau_n]/U \vdash P \qquad rel(U)}{\Gamma \vdash (\nu x)\, P} \quad \text{(T-New)}$$

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \,||\, \Gamma_2 \vdash P_1 \mid P_2} \quad \text{(T-Par)} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\boxed{\mathbf{t_{min}}}\,\Gamma \,||\, v:bool \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash P}{*\Gamma \vdash *P} \quad \text{(T-Rep)} \qquad \frac{\Gamma' \vdash P \qquad \Gamma \le \Gamma'}{\Gamma \vdash P} \quad \text{(T-Weak)}$$

$$\frac{\Gamma, x:[\tau_1,\ldots,\tau_n]/U \vdash P \qquad a = \mathbf{c} \Rightarrow t_c < \mathbf{t}_\infty}{\boxed{t_c + \mathbf{t_{min}}}(v_1:\tau_1 \,||\, \cdots \,||\, v_n:\tau_n \,||\, \Gamma) \,||\, x:[\tau_1,\ldots,\tau_n]/O^{t_o}_{t_c}.U \vdash x!^a[v_1,\ldots,v_n].\, P} \quad \text{(T-Out)}$$

$$\frac{\Gamma, x:[\tau_1,\ldots,\tau_n]/U, y_1:\tau_1,\ldots,y_n:\tau_n \vdash P \qquad a = \mathbf{c} \Rightarrow t_c < \mathbf{t}_\infty}{\boxed{t_c + \mathbf{t_{min}}}\,\Gamma, x:[\tau_1,\ldots,\tau_n]/I^{t_o}_{t_c}.U \vdash x?^a[y_1,\ldots,y_n].\, P} \quad \text{(T-In)}$$

Fig. 1. Typing Rules

4.7 Typing Rules

The typing rules are shown in Figure 1. Each rule is explained below.

(T-Par) The premises imply that P_1 uses variables as described by Γ_1, and in parallel to this, P_2 uses variables as described by Γ_2. So, the type environment of $P_1 \mid P_2$ should be represented as the combination $\Gamma_1 \,||\, \Gamma_2$. For example, if $\Gamma_1 = x:[\,]/I^{t_o}_{t_c}.0$ and $\Gamma_2 = x:[\,]/O^{t'_o}_{t'_c}.0$, then $P_1 \mid P_2$ should be well typed under $x:[\,]/(I^{t_o}_{t_c}.0 \,||\, O^{t'_o}_{t'_c}.0)$.

(T-Rep) Because $*P$ runs infinitely many copies of P in parallel and the premise implies that each P uses variables according to Γ, $*P$ uses variables according to $*\Gamma$ as a whole.

(T-New) $(\nu x)\, P$ is well typed if P is well typed and it uses x as a channel of a reliable usage.

(T-If) Since **if** v **then** P **else** Q executes either P or Q, P and Q must be well typed under the same type environment Γ. Assuming that it takes the time $\mathbf{t_{min}}$ to check whether v is *true* or *false*, we can express the total use of variables by **if** v **then** P **else** Q as $\boxed{\mathbf{t_{min}}}\Gamma \,||\, v:bool$.

(T-Weak) $\Gamma \le \Gamma'$ means that Γ represents a more liberal use of variables than Γ'. So, if P is well typed under Γ', so is under Γ.

(T-Out) The lefthand premise implies that P uses x as a channel of usage U and uses other variables according to Γ. Because $x!^a[v_1,\ldots,v_n].\, P$ uses x for output before that, the total usage of x is expressed by $O^{t_o}_{t_c}.U$. Other variables are used by P and the receiver of $[v_1,\ldots,v_n]$ according to $v_1:\tau_1 \,||\, \cdots \,||\, v_n:\tau_n \,||\, \Gamma$ only after the communication on x succeeds. Because it may take the time t_c until the communication is enabled and it takes $\mathbf{t_{min}}$ more before the

communication completes, the total use of other variables is estimated as $\boxed{t_c + \mathbf{t_{min}}}(v_1 : \tau_1 \,||\, \cdots \,||\, v_n : \tau_n \,||\, \Gamma)$. The righthand premise requires that the time bound of the capability must be finite if the input is annotated with **c**.
(T-In) This is similar to the rule (T-Out). Because P uses x according to the usage U, the total usage of x is expressed as $I^{t_o}_{t_c}.U$. Because $x?^a[y_1, \ldots, y_n].P$ executes P only after the communication on x has completed, the total use of other variables is estimated as $\boxed{t_c + \mathbf{t_{min}}}\Gamma$.

4.8 Type Soundness

Now we show that our type system is sound in the sense that any closed well-typed process is livelock-free (Corollary 42). We omit most of the proofs but present a proof sketch of the main theorem (Theorem 41) because it may be particularly interesting: While the usual type soundness refers to a safety property that a bad thing never happens, livelock-freedom is a liveness property that a good thing eventually happens. Full proofs are given in the full version of this paper [9].

Type soundness comes from the subject reduction theorem (Theorem 39), which states that well-typedness of a process is preserved by reduction, *plus* a property that some progress is always made by reduction (Lemma 40). As in the linear π-calculus, the type environment of a process may change during reduction. For example, while a process $x![\,] \,|\, x?[\,].\mathbf{0}$ is well typed under $x : [\,]/(O^{t_o}_{t_c}.0 \,||\, I^{t'_o}_{t'_c}.0)$, the reduced process $\mathbf{0}$ is well typed under $x : [\,]/0$. This change of a type environment is captured by the following relation $\Gamma \xrightarrow{l} \Delta$.

Definition 38. *A 3-place relation $\Gamma \xrightarrow{l} \Delta$ is defined to hold if one of the following conditions holds:*

1. *$l = \epsilon$ and $\Gamma = \Delta$.*
2. *$l = x$, $\Gamma = (\Gamma', x : [\tilde{\tau}]/U)$, $\Delta = (\Gamma', x : [\tilde{\tau}]/U')$, and $U \longrightarrow U'$ for some $\Gamma', \tilde{\tau}, U$, and U'.*

We write $\Gamma \longrightarrow \Delta$ when $\Gamma \xrightarrow{l} \Delta$ for some l.

Theorem 39 (subject reduction). *If $\Gamma \vdash P$ and $P \xrightarrow{l} Q$, then there exists Δ such that $\Delta \vdash Q$ and $\Gamma \xrightarrow{l} \Delta$.*

The following lemma states that after a communication succeeds, the processes that have been blocked by the communication can fulfill obligations within shorter time bounds.

Lemma 40. *If $\Gamma, x : [\tilde{\tau}]/U \vdash x!^a[\tilde{v}].P \,|\, x?^{a'}[\tilde{y}].Q$, then there exist Δ, U' such that $\Delta, x : [\tilde{\tau}]/U' \vdash P \,|\, [\tilde{y} \mapsto \tilde{v}]Q$, $\Gamma \leq \boxed{\mathbf{t_{min}}}\Delta$, and $U \longrightarrow U'$.*

The following main theorem says that any obligation with a finite time bound is eventually fulfilled (the properties A and B below), and that any capability with a finite time bound can eventually be used (the properties C and D).

Theorem 41. *Suppose that $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$ is a full fair reduction sequence. If $t < \mathbf{t}_\infty$, the following properties hold.*

- *A (Output Obligation). If (i) $\Gamma \vdash P_0$, (ii) $rel(\Gamma)$, (iii) $\Gamma = \Gamma', x:[\tilde{\tau}]/U$, and (iv) $ob_O(t, U)$, then there exists $n \geq 0$ such that $P_n \succeq (\nu\tilde{w})\,(x!^a[\tilde{v}].\,Q_1 \mid Q_2)$.*
- *B (Input Obligation). If (i) $\Gamma \vdash P_0$, (ii) $rel(\Gamma)$, (iii) $\Gamma = \Gamma', x:[\tilde{\tau}]/U$, and (iv) $ob_I(t, U)$, then there exists $n \geq 0$ such that $P_n \succeq (\nu\tilde{w})\,(x?^a[\tilde{y}].\,Q_1 \mid Q_2)$.*
- *C (Output Capability). If (i) $P_0 \succeq x!^a[\tilde{v}].\,Q \mid R$, (ii) $\Delta_1, x:[\tilde{\tau}]/O_t^{t_o}.U_1 \vdash x!^a[\tilde{v}].\,Q$, (iii) $\Delta_2 \vdash R$, and (iv) $rel((\Delta_1, x:[\tilde{\tau}]/O_t^{t_o}.U_1) \,||\, \Delta_2)$, then there exists $n \geq 0$ such that $P_n \succeq (\nu\tilde{w})\,(x!^a[\tilde{v}].\,Q \mid x?^{a'}[\tilde{y}].\,R_1 \mid R_2)$ and $(\nu\tilde{w})\,(Q \mid [\tilde{y} \mapsto \tilde{v}]R_1 \mid R_2) \succeq P_{n+1}$.*
- *D (Input Capability). If (i) $P_0 \succeq x?^a[\tilde{y}].\,Q \mid R$, (ii) $\Delta_1, x:[\tilde{\tau}]/I_t^{t_o}.U_1 \vdash x?^a[\tilde{y}].\,Q$, (iii) $\Delta_2 \vdash R$, and (iv) $rel((\Delta_1, x:[\tilde{\tau}]/I_t^{t_o}.U_1) \,||\, \Delta_2)$, then there exists $n \geq 0$ such that $P_n \succeq (\nu\tilde{w})\,(x?^a[\tilde{y}].\,Q \mid x!^{a'}[\tilde{v}].\,R_1 \mid R_2)$ and $(\nu\tilde{w})\,([\tilde{y} \mapsto \tilde{v}]Q \mid R_1 \mid R_2) \succeq P_{n+1}$.*

Proof Sketch. The proof proceeds by induction on t (notice that the relation $<$ on the subset $\{t \mid t < \mathbf{t}_\infty\}$ of time quantums is well-founded: there is no infinite decreasing sequence). We show only the properties A and C. B and D are similar.

Suppose the theorem holds for any t' such that $t' < t$.

A. By the assumption $\Gamma \vdash P_0$, there must exist P_{01} and P_{02} such that (a) $P_0 \succeq (\nu\tilde{w})\,(P_{01} \mid P_{02})$, (b) $\Gamma_1, x:[\tilde{\tau}]/(O_{t_c}^t.U_3 \,||\, U_4) \vdash P_{01}$, and (c) P_{01} is an output, input, or conditional process. Without loss of generality, we can assume that $(\nu\tilde{w})\,(P_{01} \mid P_{02}) \longrightarrow P_1$. If P_{01} is of the form $x!^a[\tilde{v}].\,R$, the result immediately holds. Otherwise, P_{01} must be **if** b **then** P'_{01} **else** P''_{01} with b being *true* or *false*, or trying to use an input or output capability on another channel with a time limit t'_c shorter than t.

In the former case, by the assumption of the fairness, P_{01} is eventually reduced to P'_{01} or P''_{01}. By the typing rules, P'_{01} and P''_{01} are well typed under $\Gamma_1, x:[\tilde{\tau}]/(O_{t''_c}^{t'}.U_3 \,||\, U_4)$ for some t' such that $t' + \mathbf{t_{min}} \leq t$ (which implies $t' < t$ as $t < \mathbf{t}_\infty$). So, the required result follows from the property A of induction hypothesis.

In the latter case, by the property C of induction hypothesis, P_{01} must be eventually reduced. By Theorem 39 and Lemma 40, the resulting process is well typed under some type environment $\Delta, x:[\tilde{\tau}]/(O_{t''_c}^{t'}.U_5 \,||\, U_6)$ such that $t' < t$. (See the full paper [9] for details.) So, the required result follows from the property A of induction hypothesis.

C. Suppose there exists no such n. Then, it must be the case that $P_i \succeq x!^a[\tilde{v}].\,Q \mid R_i$ and $R \longrightarrow R_1 \longrightarrow R_2 \longrightarrow \cdots$ is a full fair reduction sequence. We show that there exist infinitely many i such that $R_i \succeq (\nu\tilde{w})\,(x?^{a'}[\tilde{y}].\,R'_i \mid R''_i)$, which contradicts with the assumption that the reduction sequence $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$ is fair. Let j be an arbitrary natural number. Then, by subject reduction (Theorem 39), $\Delta', x:[\tilde{\tau}]/U' \vdash R_j$ and $\Delta_2 \longrightarrow^* (\Delta', x:[\tilde{\tau}]/U')$ for some Δ' and U'. By the assumption (iv) and the definition of the reliability, $ob_I(t, U')$ must hold. We also

have that $\Gamma', x:[\tilde{\tau}]/(O_t^{t_o}.U_1 \,||\, U') \vdash x!^a[\tilde{v}].Q \,|\, R_j$ and $rel(\Gamma')$ for $\Gamma' = \Delta_1 \,||\, \Delta'$. Therefore, by the property B, there exists $i \geq j$ such that $R_i \succeq (\nu\tilde{w})\,(x?^{a'}[\tilde{y}].R_i' \,|\, R_i'')$. Thus, we have shown that there exist infinitely many i such that $R_i \succeq (\nu\tilde{w})\,(x?^{a'}[\tilde{y}].R_i' \,|\, R_i'')$. □

Corollary 42 (livelock freedom). *If $\emptyset \vdash P$, then P is livelock-free.*

Proof. Suppose $\emptyset \vdash P$ and $P \longrightarrow^* P_i \succeq (\nu\tilde{w})\,(x!^{\mathbf{c}}[\tilde{v}].Q \,|\, R)$. By Theorem 39, $\tilde{w}:\tilde{\tau} \vdash x!^{\mathbf{c}}[\tilde{v}].Q \,|\, R$ and $rel(\tilde{w}:\tilde{\tau})$. Moreover, by the typing rules, it must be the case that $\Delta_1, x:[\tilde{\tau'}]/O_{t_c}^{t_o}.U_1 \vdash x!^{\mathbf{c}}[\tilde{v}].Q$, $t_c < \mathbf{t}_\infty$, $\Delta_2 \vdash R$, and $rel((\Delta_1, x:[\tilde{\tau'}]/O_{t_c}^{t_o}.U_1) \,||\, \Delta_2)$. So, the required result follows from the property C of Theorem 41. The case for input is similar. □

5 Time-Bounded Processes

In this section, we briefly show that with a minor modification, our type system can guarantee not only that certain communications *eventually* succeed, but also that some of them succeed within a certain number of (parallel) reduction steps.

5.1 Time-Boundedness

We first define the time-boundedness of a process. We extend the syntax of processes to allow a programmer to declare an upper-bound of the number of reduction steps required until an input or output operation succeeds.

$$P ::= \cdots \mid x!^n[\tilde{v}].P \mid x?^n[\tilde{y}].P$$

The annotation n of $x!^n[\tilde{v}].P$ indicates that the programmer wants this output to succeed within n *parallel* reduction steps (defined below) after it is executed. (Strictly speaking, the output process can *find* its communication partner within n reduction steps and complete the communication in the next step.)

In counting the number of reduction steps, we assume unlimited parallelism, so that communications on different channels can occur in parallel. To model such parallel reduction, we introduce parallel reduction relations $P \Longrightarrow Q$ and $P \overset{S}{\Longrightarrow} Q$ where S is a set of channels. $P \Longrightarrow Q$ means that P is reduced to Q by reducing every conditional expression and performing one communication on every channel whenever possible. $P \overset{S}{\Longrightarrow} Q$ is the same as $P \Longrightarrow Q$ except that communications on free channels occur only on those in the set S. We assume here that at most one communication can occur on each channel and that the two processes to communicate is chosen randomly. So, it takes two steps to reduce $*x?[].\mathbf{0} \,|\, x![] \,|\, x![]$ to $*x?[].\mathbf{0}$. It would be possible to change reduction rules and the type system so as to make as many communications as possible occur simultaneously on each channel and/or to reflect a certain scheduling.

Definition 43 (parallel process reduction). $\stackrel{S}{\Longrightarrow}$ *(where S is a set of variables) and* $\Longrightarrow$ *are the least relations closed under the rules given below.* a^- *is defined by:* $n^- = n-1$, $\mathbf{c}^- = \mathbf{c}$, *and* $\emptyset^- = \emptyset$.

$$x!^a[\tilde{v}].\,P \mid x?^{a'}[\tilde{z}].\,Q \stackrel{\{x\}}{\Longrightarrow} P \mid [\tilde{z} \mapsto \tilde{v}]Q \qquad \mathbf{0} \stackrel{\emptyset}{\Longrightarrow} \mathbf{0}$$

$$x!^a[\tilde{v}].\,P \stackrel{\emptyset}{\Longrightarrow} x!^{a^-}[\tilde{v}].\,P \qquad x?^a[\tilde{v}].\,P \stackrel{\emptyset}{\Longrightarrow} x?^{a^-}[\tilde{v}].\,P$$

$$\textbf{if } \mathit{true} \textbf{ then } P \textbf{ else } Q \stackrel{\emptyset}{\Longrightarrow} P \qquad \textbf{if } \mathit{false} \textbf{ then } P \textbf{ else } Q \stackrel{\emptyset}{\Longrightarrow} Q$$

$$\frac{P \stackrel{S_1}{\Longrightarrow} P' \qquad Q \stackrel{S_2}{\Longrightarrow} Q' \qquad S_1 \cap S_2 = \emptyset}{P \mid Q \stackrel{S_1 \cup S_2}{\Longrightarrow} P' \mid Q'} \qquad \frac{P \stackrel{\emptyset}{\Longrightarrow} Q}{*P \stackrel{\emptyset}{\Longrightarrow} *Q}$$

$$\frac{P \stackrel{S}{\Longrightarrow} Q \qquad x \in S \text{ if } P \stackrel{x}{\longrightarrow}}{(\nu x)\,P \stackrel{S \setminus \{x\}}{\Longrightarrow} (\nu x)\,Q} \qquad \frac{P \succeq P' \qquad P' \stackrel{S}{\Longrightarrow} Q' \qquad Q' \succeq Q}{P \stackrel{S}{\Longrightarrow} Q}$$

$$\frac{P \stackrel{S}{\Longrightarrow} Q \qquad \forall x.(x \in S \text{ if } P \stackrel{x}{\longrightarrow})}{P \Longrightarrow Q}$$

The rules in the first two lines ensure that in each parallel reduction step, every input or output process is either reduced or its time bound is decreased by 1. The rules in the third line ensure that every conditional process is reduced. The righthand premises of the rule for $(\nu x)\,P$ and the last rule make sure that a communication happens on every channel whenever possible.

A process is time-bounded if whenever the time bound of an input or output process has become 0 (i.e., it becomes $x!^0[\tilde{v}].\,P$ or $x?^0[\tilde{y}].\,P$), the input or output operation always succeeds in the next parallel reduction step:

Definition 44 (time-boundedness). *A process P is* time-bounded *if the following conditions hold whenever* $P \Longrightarrow^* P'$.

1. *If* $P' \succeq (\nu\tilde{w})\,(x!^0[\tilde{v}].\,Q \mid R)$, *then* $R \not\stackrel{x}{\longrightarrow}$ *and* $R \succeq (\nu\tilde{u})\,(x?^a[\tilde{y}].\,R_1 \mid R_2)$ *for some* $\tilde{u}, R_1, R_2$.
2. *If* $P' \succeq (\nu\tilde{w})\,(x?^0[\tilde{y}].\,Q \mid R)$, *then* $R \not\stackrel{x}{\longrightarrow}$ *and* $R \succeq (\nu\tilde{u})\,(x!^a[\tilde{v}].\,R_1 \mid R_2)$ *for some* $\tilde{u}, \tilde{v}, R_1, R_2$.

5.2 Modification to the Type System

Now we show how to modify the type system to guarantee the time-boundedness. Because the typing rules in Section 4 already take into account the delay caused by communications on other channels, we just need to refine the reliability condition to estimate the channel-wise behavior more correctly. As stated in Remark 31, a problem of Definition 30 is that it does not take race conditions into account. For example, $O^0_{\mathbf{t}_\infty}.0 \,||\, I_0^{\mathbf{t}_\infty}.O^0_{\mathbf{t}_\infty}.0 \,||\, I_0^{\mathbf{t}_\infty}.O^0_{\mathbf{t}_\infty}.0$ is reliable according to Definition 30, but only one input is guaranteed to succeed immediately: The

other input must wait for a time period of length $\mathbf{t_{min}}$ until an output is executed again. So, the correct usage should be $O^0_{\mathbf{t}_\infty}.0 \,||\, I^{\mathbf{t}_\infty}_{\mathbf{t_{min}}}.O^0_{\mathbf{t}_\infty}.0 \,||\, I^{\mathbf{t}_\infty}_{\mathbf{t_{min}}}.O^0_{\mathbf{t}_\infty}.0$.

We redefine usage reduction to take race conditions into account. For example, $O^0_{\mathbf{t}_\infty}.0 \,||\, I^{\mathbf{t}_\infty}_{\mathbf{t_{min}}}.O^0_{\mathbf{t}_\infty}.0 \,||\, I^{\mathbf{t}_\infty}_{\mathbf{t_{min}}}.O^0_{\mathbf{t}_\infty}.0$ is reduced to $O^0_{\mathbf{t}_\infty}.0 \,||\, I^{\mathbf{t}_\infty}_0.O^0_{\mathbf{t}_\infty}.0$, which can be further reduced to $O^0_{\mathbf{t}_\infty}.0$. To define a new usage reduction relation, we introduce auxiliary relations $\overset{S}{\Longrightarrow}$ where $\{\mathbf{com}, I, O\} \subseteq S$. When $\mathbf{com} \in S$, $U \overset{S}{\Longrightarrow} V$ means that one pair of an input usage and an output usage is reduced. $I \in S$ ($O \in S$, resp.) indicates that an input (output, resp.) action is ready but kept waiting (either because no output action is ready or because another input is chosen for communication).

Definition 45 (timed usage reduction). *Binary relations $\overset{S}{\Longrightarrow}$ and $\Longrightarrow$ ($\{\mathbf{com}, I, O\} \subseteq S$) on usages are the least relations closed under the following rules:*

$$\frac{}{I^{t_o}_{t_c}.U_1 \,||\, O^{t'_o}_{t'_c}.U_2 \overset{\{\mathbf{com}\}}{\Longrightarrow} U_1 \,||\, U_2} \qquad \frac{}{0 \overset{\emptyset}{\Longrightarrow} 0}$$

$$\frac{0 < t_c}{\alpha^{t_o}_{t_c}.U \overset{\{\alpha\}}{\Longrightarrow} \alpha^0_{t_c{}^-}.U} \qquad \frac{0 < t_o}{\alpha^{t_o}_{t_c}.U \overset{\emptyset}{\Longrightarrow} \alpha^{t_o{}^-}_{t_c}.U}$$

$$\frac{U_1 \overset{S_1}{\Longrightarrow} U'_1 \qquad U_2 \overset{S_2}{\Longrightarrow} U'_2 \qquad \mathbf{com} \notin S_1 \cap S_2}{U_1 \,||\, U_2 \overset{S_1 \cup S_2}{\Longrightarrow} U'_1 \,||\, U'_2} \qquad \frac{U \overset{S}{\Longrightarrow} U' \qquad \mathbf{com} \notin S}{*U \overset{S}{\Longrightarrow} *U'}$$

$$\frac{U \cong U' \qquad U' \overset{S}{\Longrightarrow} V' \qquad V' \cong V}{U \overset{S}{\Longrightarrow} V} \qquad \frac{U \overset{S}{\Longrightarrow} V \qquad \mathbf{com} \in S \text{ if } \{I, O\} \subseteq S}{U \Longrightarrow V}$$

Here, $t^- = (n-1) \cdot \mathbf{t_{min}}$ if $[\![t]\!] = [\![n \cdot \mathbf{t_{min}}]\!]$, and $t^- = t$ otherwise.

The left rule in the second line is for the case where the (input or output) action α has already been executed but does not succeed in this reduction step: in this case, the time limit of the capability is reduced by $\mathbf{t_{min}}$. The right rule in the second line is for the case where the action α has not been executed yet: in this case, the time limit of the obligation is reduced by $\mathbf{t_{min}}$. The right rule at the bottom ensures that in each reduction step, if both input and output actions are ready, some pair of an input usage and an output usage must be reduced.

Using the above parallel usage reduction, we can strengthen the reliability condition as defined below. The first condition ensures that when the time limit of an input or output capability has reached 0, the input or output operation must succeed in the next step.

Definition 46 (reliability (refined)). *A usage U is reliable if:*

1. *If $U \Longrightarrow^* \cong \alpha^{t_o}_0.U_1 \,||\, U_2$, then $U_2 \not\longrightarrow$ and there exist t_c, U_3, U_4 such that $U_2 \cong \overline{\alpha}^0_{t_c}.U_3 \,||\, U_4$; and*
2. *$U \longrightarrow^* \cong \alpha^{t_o}_{t_c}.U_1 \,||\, U_2$ implies $ob_{\overline{\alpha}}(t_c, U_2)$.*

We introduce the following typing rules for new processes.

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U \vdash P \qquad t_c \leq n \cdot \mathbf{t_{min}}}{\boxed{t_c + \mathbf{t_{min}}}(v_1 : \tau_1 \,||\, \cdots \,||\, v_n : \tau_n \,||\, \Gamma) \,||\, x : [\tau_1, \ldots, \tau_n]/O^{t_o}_{t_c}.U \vdash x!^n[v_1, \ldots, v_n].\, P} \quad \text{(T-BOut)}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash P \qquad t_c \leq n \cdot \mathbf{t_{min}}}{\boxed{t_c + \mathbf{t_{min}}}\Gamma, x : [\tau_1, \ldots, \tau_n]/I^{t_o}_{t_c}.U \vdash x?^n[y_1, \ldots, y_n].\, P} \quad \text{(T-BIn)}$$

We can prove the following time-boundedness theorem. The proof is basically similar to that of the livelock-freedom property (Corollary 42): It suffices to prove a parallel version of the subject reduction property (Theorem 39) (although it is harder).

Theorem 47. *Every closed well-typed process is time-bounded.*

6 Extensions with Dependent Types

The typed livelock-free process calculus in Section 4 is not expressive enough. For example, consider the following process P, which is a function server computing the factorial of a natural number:

$$*fact?[n, r].\, (\textbf{if } n = 0 \textbf{ then } r![1] \textbf{ else } (\nu r')\, (fact![n - 1, r'] \,|\, r'?[m].\, r![m \times n])\ )$$

$P \,|\, (\nu y)\, (fact![3, y].\ |\, y?^{\mathbf{c}}[x].\, \mathbf{0})$ cannot be judged to be livelock-free in our type system. Suppose r has type $[\mathbf{Nat}]/O^{t_o}_{t_c}.0$. Because r' is passed through the channel *fact*, r' must have the same type $[\mathbf{Nat}]/O^{t_o}_{t_c}.0$. According to the rule (T-In), in order for $r'?[m].\, r![m \times n]$ to be well typed, it must be the case that $t_o + \mathbf{t_{min}} \leq t_o$, which implies $\mathbf{t}_\infty \leq t_o$. So, it is not guaranteed that P returns a result eventually.

The problem of the above example is that the time bound of an output obligation on r depends on the other argument n. We can use dependent types to express such dependency: In the above process, the type of the channel *fact* can be expressed as $[\Sigma n : \mathbf{Nat}.[\mathbf{Nat}]/O^{2n \cdot \mathbf{t_{min}}}_0.0]/U$.

7 Related Work

Our Previous Type Systems for Deadlock-Freedom. As mentioned earlier, the type system in this paper originates from our previous type systems for deadlock-freedom [7,23,11]. We expect that we can reconstruct a type system for deadlock-freedom by changing the definition of the order between time tags (for example, by identifying $\mathbf{t}_1 + \mathbf{t}_2$ with $\mathbf{t}_2$ if $\mathbf{t}_1 \ll \mathbf{t}_2$). Nice points about the new type system are that the intuitive meaning of a channel type is clearer, and that we can get rid of complex side conditions on time tags (which were introduced to get enough expressive power) in the typing rules of the previous type systems [7,23]. We expect that we can recover much of the expressive power by using more standard concepts like dependent types and polymorphism as described in Section 6.

Other Type Systems for Analyzing Similar Properties of Concurrent Processes. There exist a few other type systems for π-calculus-like languages (where channels are first-class data) that guarantee a livelock-freedom property. As far as we know, however, they deal with more specific situations: Sangiorgi's type system for receptiveness [22] enforces that an input process is spawned immediately after a certain channel (called a receptive name) is created, and therefore guarantees that every output on that channel succeeds immediately. Puntigam and Peter's typed concurrent object calculus [20] guarantees that certain reply messages (which they call promised messages) eventually arrive.

There are a few other type systems that deal with deadlock-freedom [1,19, 26]. Please refer to our previous paper [11] for comparisons with them.

Abadi and Flanagan [4] recently proposed a typed concurrent object calculus that can prevent race conditions. Our type system can also be used to prevent race conditions. Notice that a race condition occurs only when more than one processes try to input or output on the same channel simultaneously. Such a situation can be detected by looking at the usage of each channel: If the usage of a channel can be reduced to a usage of the form $I^{t_o}_{t_c}.U_1 \,||\, I^{t'_o}_{t'_c}.U_2 \,||\, \cdots$, more than one processes may try to perform an input on the channel at the same time. Abadi and Flanagan's race detection [4] is, however, more sophisticated because dependencies between different channels (or locks) are taken into account. We might be able to extend our type system to subsume it by extending a channel usage so that it expresses the use of a group of channels, instead of that of each channel (see discussions on an extension to a deadlock-free concurrent object calculus in [11]).

Type Systems for Bounding Execution Time of Sequential Programs. There are several pierces of work that try to statically bound running-time of sequential programs [6,3]. Most closely related (especially to the extension sketched in Section 6) seems to be Crary and Weirich's work [3], which also uses dependent types. A difficulty in bounding running-time of a concurrent process is that unlike sequential programs where a function/procedure call is never blocked, a process may be blocked until a communication partner becomes ready. We have dealt with this difficulty in this paper by associating each input/output operation with two time bounds: a time bound within which the operation is executed, and another time bound within which a communication partner becomes ready.

Abstract Interpretation. An alternative way to analyze the behavior of a concurrent program would be to use abstract interpretation [2]. Actually, from a very general viewpoint, our type-based analysis of deadlock and livelock can be seen as a kind of abstract interpretation. We can read a type judgment $\Gamma \vdash P$ as "Γ is an abstraction of a concrete process P." (The relation "$\vdash$" corresponds to a pair of abstraction/concretization functions.) Indeed, we can regard a type environment as an abstract process: we have defined reductions of type environments in Section 4.8.

The subject reduction property (Theorem 39) can be interpreted as "whenever a concrete process P is reduced to another concrete process Q, an abstraction

Γ of P can also be reduced to another abstract process Δ which is an abstraction of Q." In other words, every reduction step of a concrete process is simulated by reduction of its abstract process. So, it corresponds to a consistency condition of abstract interpretation. The reliability condition (Definition 30) guarantees that an abstract process never falls into a livelock. Thus, a concrete process is also guaranteed to be livelock-free.

8 Conclusion

In this paper, we have extended our previous type systems for deadlock-freedom to guarantee livelock-freedom and time-boundedness properties. A number of practical issues remain to be solved in applying these type systems to real programming languages, such as how to combine dependent types, polymorphism, etc. to obtain a reasonable expressive power and how and to what extent to let programmers declare type information.

A key idea common to those type systems is to decompose the behavior of a whole process into that on each communication channel, which is specified by using a mini-process calculus of usages. This idea would be applicable to other analyses, such as race detection (as mentioned in Section 7) and memory management. As for an application to memory management, we have already applied a similar idea to analyze how and in which order each heap cell (instead of a communication channel) is used in functional programs [8].

Acknowledgment

We would like to thank Atsushi Igarashi, Eijiro Sumii, and Nobuko Yoshida for useful comments.

References

1. G. Boudol. Typing the use of resources in a concurrent calculus. In *Proceedings of ASIAN'97*, pages 239–253, 1997.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.
3. K. Crary and S. Weirich. Resource bound certification. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 184–198, 2000.
4. C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99*, LNCS 1664, pages 288–303. Springer-Verlag, 1999.
5. S. J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 429–438, 1993.
6. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 464–473, 1999.

7. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary summary appeared in Proceedings of LICS'97, pages 128–139.
8. N. Kobayashi. Quasi-linear types. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 29–42, 1999.
9. N. Kobayashi. A livelock-free typed process calculus. Technical report, Dept. Info. Sci., Univ. of Tokyo, 2000. To appear. Available at http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html.
10. 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. Preliminary summary appeared in Proceedings of POPL'96, pp.358-371.
11. 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. A summary will appear in Proceedings of CONCUR 2000, LNCS, Springer-Verlag.
12. 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.
13. R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100:1–77, September 1992.
15. U. Nestmann. What is a 'good' encoding of guarded choice? In *EXPRESS'97*, volume 7 of *ENTCS*. Elsevier Science Publishers, September 1997.
16. S. Peyton Jones. Concurrent Haskell. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 295–308, 1996.
17. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
18. 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)*, LNCS 907, pages 187–215. Springer-Verlag, 1995.
19. F. Puntigam. Coordination requirements expressed in types for active objects. In *Proceedings of ECOOP'97*, LNCS 1241, pages 367–388, 1997.
20. F. Puntigam and C. Peter. Changeable interfaces and promised messages for concurrent components. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 141–145, 1999.
21. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.
22. D. Sangiorgi. The name discipline of uniform receptiveness (extended abstract). In *Proceedings of ICALP'97*, LNCS 1256, pages 303–313, 1997.
23. 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.
24. V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In *CONCUR'93*, LNCS 715, pages 524–538. Springer-Verlag, 1993.
25. A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.
26. N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, LNCS 1180, pages 371–387. Springer-Verlag, 1996.