A Type System for Lock-Free Processes[1]

Naoki Kobayashi

Department of Computer Science, Graduate School of Information Science and Engineering
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo 152-8552, Japan
E-mail:kobayasi@cs.titech.ac.jp

Advanced type systems for the π-calculus have recently been proposed to guarantee deadlock-freedom in the sense that certain communications will eventually succeed *unless the whole process diverges*. Although such guarantees are useful for reasoning about the behavior of concurrent programs, there still remains the weakness that the success of a communication is not completely guaranteed due to the possibility of divergence. For example, although 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 present a type system which guarantees that certain communications will eventually succeed under fair scheduling, regardless of whether processes diverge. We also present a variant of the type system which guarantees that a communication will succeed within a given number of reduction steps.

1. INTRODUCTION

It is an important and challenging task to statically guarantee the correctness of concurrent programs. Concurrent programs are more complex than sequential programs (due to dynamic control, non-determinism, deadlock, etc.), which makes it hard for programmers to debug concurrent programs or reason about their behavior.

Unfortunately, existing concurrent/distributed programming languages and thread libraries provide only limited support for checking the correctness of concurrent programs. For example, consider the following program in CML [30, 31].

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

The function `f` takes an integer `n` as an argument, creates a fresh channel `c`, and waits to receive a value on the channel (by `recv(c)`). Since there is no sender on `c`, evaluation gets stuck at `recv(c)`.

[1]A preliminary version of this paper appeared in Proceedings of IFIP TCS2000, LNCS 1872, Springer-Verlag, pp.365–389, 2000, under the title "Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness."

A function in CML may also behave in a non-deterministic manner. Consider the following program.

```
fun g(n:int) = let val c = channel() in
                   (spawn(fn ()=>send(c,1));
                    spawn(fn ()=>send(c,2));
                    recv(c)+n)
                end
```

The function g creates a fresh channel c, spawns two processes that send 1 and 2 to the channel, and waits to receive a value on c. Then, it adds n to the received value and returns the result. So g(n) returns either n+1 or n+2 in a non-deterministic manner. In spite of these non-functional behaviors of f and g, CML assigns to them a function type $int \rightarrow int$. So, CML does not guarantee that a term of function type really behaves like a function.

To improve the situation above, a number of type systems [16, 25, 26, 32, 38] have been studied in the setting of process calculi, just as type systems for functional languages have been studied in the setting of λ-calculus. Along this line of research, we have proposed expressive type systems [14, 17, 34] for the π-calculus [21, 22, 23] to guarantee deadlock-freedom of processes. The deadlock-freedom property is useful for reasoning about behavior of concurrent programs. Suppose that a client process sends a request to a server process. The client and server processes can be written as

$$\begin{array}{rcl} Client & \stackrel{\triangle}{=} & (\nu r)\,(\overline{s}\langle req, r\rangle \mid r(rep)^{\mathbf{c}}.\,P) \\ Server & \stackrel{\triangle}{=} & *s(x,r).\,Q \end{array}$$

in a π-calculus-like language. Here, (νr) creates a fresh communication channel r for receiving a reply from the server. $\overline{s}\langle req, r\rangle$ sends a request req and the channel r to the location s (which is also a channel) of the server. In parallel to this, $r(rep)^{\mathbf{c}}.\,P$ waits to receive a reply rep from the server. The annotation $\mathbf{c}$ indicates that $r(rep)^{\mathbf{c}}.\,P$ should eventually receive a reply. The server process $*s(x,r).\,Q$ repeatedly receives a request x and a reply channel r through channel s and behaves like Q. In general, there is no guarantee that the client can receive a reply; Q may do nothing, ignoring the request from the client, or Q may be blocked forever before sending a reply. However, our previous type systems [17] can guarantee that if $Server \mid Client$ is well-typed, the client can eventually receive a reply as long as $Server$ does not diverge. In this way, type systems for deadlock-freedom can ensure that a process implementing a server really behaves as a correct server and that a channel implementing a semaphore is really used as a semaphore in the sense that a process that has acquired a semaphore will release it eventually.

Although the deadlock-freedom property above is useful for reasoning about behavior of concurrent programs, there is still a limitation: success of a communication is not completely guaranteed because a process may diverge before the communication succeeds. In the client-server model above, while $Server$ cannot ignore a request, it is allowed to infinitely delegate a request to other processes. For example, Q may be a process $\overline{s}\langle x, r\rangle$, which resends all received messages on s. Then, the client and server processes fall into a *livelock* [19, 20, 30], a situation where processes are executing forever without doing useful work. These livelocked client and server processes are, however, well-typed in our type systems for deadlock-freedom [14, 17, 34].

In this paper, we present a new type system which guarantees that certain communications will eventually succeed *regardless of whether processes diverge*, provided that scheduling is *strongly fair* [3, 7] in the sense that every process that is able to participate in a communication infinitely often can eventually participate in a communication. We call this property *lock-freedom* (modulo the fairness assumption), because not only deadlock situations but also livelock situations like the one described above are prevented: the livelocked client and server processes above are judged to be ill-typed in the new type system. We also present a variant of the type system, which guarantees that a communication will succeed within a certain amount of time. For example, one can write $r(x)^n.\, P$ to denote a process that should receive a value on channel r within n reduction steps. If the whole system of processes containing this process is well-typed, it is indeed guaranteed that a reply is received within n-steps of reductions of the whole system.

The basic idea of the new type system is the same as that of the previous type systems for deadlock-freedom [14, 17, 34]: Channel types are augmented with information about the order in which each channel is used for input or output. In the new type system, types are further augmented with information about how much time it takes for a process to become ready to input or output a value on a channel, and how much time it takes for the process to succeed to input or output a value after the process has become ready. The resulting type system is simpler than the previous ones, as discussed in Section 6.

An alternative approach to guaranteeing success of communication would be to develop a type system to guarantee termination of a process and combine it with the previous type systems for deadlock-freedom (because deadlock-freedom implies success of communications unless the whole process diverges). We do not take this approach, because in order to guarantee success of a communication, we must guarantee termination of the whole process, which is in general difficult. For example, let us consider the process $\overline{s}\langle r\rangle \mid s(x).\, (\overline{x}\langle\,\rangle \mid P)$, where P is some complex process. It is easy to see that a message is sent on r; Indeed, our type system can guarantee this property. However, if we try to derive that property by showing termination of the whole process, we may fail: When P is complex, it is difficult to analyze whether P terminates. Requiring termination of the whole process is also too restrictive: many correct concurrent programs run forever.

The rest of this paper is organized as follows. In Section 2, we introduce our target language and then explain what we mean by deadlock-freedom and lock-freedom. Section 3 introduces our new type system for lock-freedom and shows its soundness. Section 4 shows that with a minor modification, our type system can also guarantee that certain communications succeed within a certain number of reduction steps. The type system given in Section 3 is rather naive and cannot guarantee the lock-freedom of recursive programs. Section 5 discusses extensions that may be useful for increasing the expressive power of the type system. Section 6 discusses related work, and Section 7 concludes. We do not discuss algorithmic issues related to type-checking or type inference in this paper. We expect that those issues are basically similar to the case for the deadlock-free calculus [14, 17].

2. TARGET LANGUAGE

This section introduces the target language of our type system and defines deadlock-freedom and lock-freedom. The target language is a subset of the polyadic π-calculus [21]. 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 first-class channels make it difficult to guarantee deadlock/lock-freedom, we keep them because they are important in modeling modern concurrent/distributed programming languages. In fact, for example, in concurrent object-oriented programming [37], an object is dynamically created and its reference is passed through messages. Since a reference to a concurrent object corresponds to a record of communication channels [18, 27], channels should be first-class data.

2.1. Syntax

Definition 2.1 (processes). The set of processes is defined as follows:

$$\begin{array}{lcl}
P \text{ (processes)} & ::= & \mathbf{0} \mid A \mid (P \,|\, Q) \mid (\nu x)\, P \\
A \text{(atomic processes)} & ::= & \overline{x}\langle v_1,\ldots,v_n\rangle^a.\, P \mid x(y_1,\ldots,y_n)^a.\, P \\
 & & \mid \textbf{if } v \textbf{ then } P \textbf{ else } Q \mid *A \\
v \text{ (values)} & ::= & \mathit{true} \mid \mathit{false} \mid x \\
a \text{ (attributes)} & ::= & \emptyset \mid \mathbf{c}
\end{array}$$

Here, x and y_i range over a countably infinite set **Var** of variables.

Notation 2.1. 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. The set of free variables in P is denoted by $FV(P)$. We write $P \equiv_\alpha Q$ when P and Q are identical up to α-conversion (renaming of bound variables). We write $[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]P$ for the 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 of variables $x_1,\ldots,x_n$. We abbreviate $[\tilde{x} \mapsto \tilde{v}]$ and $(\nu\tilde{x})$ to $[x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]$ and $(\nu x_1)\cdots(\nu x_n)$, respectively.

We often omit an inaction $\mathbf{0}$ and write $\overline{x}\langle\tilde{y}\rangle^a$ for $\overline{x}\langle\tilde{y}\rangle^a.\,\mathbf{0}$. When attributes are not important, we omit them and just write $\overline{x}\langle\tilde{y}\rangle.\,P$ and $x(\tilde{y}).\,P$ for $\overline{x}\langle\tilde{y}\rangle^a.\,P$ and $x(\tilde{y})^a.\,P$ respectively.

We assume that prefixes ($\overline{x}\langle\tilde{y}\rangle^a.$, $x(\tilde{y})^a.$, (νx), and $*$) bind tighter than the parallel composition operator ($|$), so that $\overline{x}\langle\tilde{y}\rangle^a.\,P \,|\, Q$ means $(\overline{x}\langle\tilde{y}\rangle^a.\,P) \,|\, Q$, not $\overline{x}\langle\tilde{y}\rangle^a.\,(P \,|\, Q)$. We also assume that $|$ is right-associative, so that $P_1 \,|\, P_2 \,|\, P_3$ means $P_1 \,|\, (P_2 \,|\, P_3)$.

$\mathbf{0}$ denotes inaction. $\overline{x}\langle v_1,\ldots,v_n\rangle^a.\,P$ denotes a process that sends a tuple $\langle v_1,\ldots,v_n\rangle$ on x and then (after the tuple is received by some process) behaves like P. Each v_i is either a boolean or a variable, which denotes a channel or a boolean. An attribute a expresses a 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 $\emptyset$. $x(y_1, \ldots, y_n)^a.\,P$ denotes a process that receives a tuple $\langle v_1, \ldots, v_n \rangle$ 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. $*A$ represents infinitely many copies of the process A running in parallel. $P \,|\, Q$ denotes 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.

Remark 2.1. Our type system presented in Section 3 does not much depend on the choice of process constructors. For example, it is not difficult to extend our type system with the choice operator [21]. The restriction that the replication operator can be only applied to atomic processes is for technical convenience in defining the notion of fairness. A general replication $*P$ can be simulated by $(\nu x)\,(*x(\,).\,P \,|\, *\overline{x}\langle\,\rangle)$.

Example 2.1. The process $*sum(m, n, r).\,\overline{r}\langle m + n\rangle$ behaves as a function server computing the sum of two integers. (Here, the language is extended with integers and operations.) It receives a triple consisting of two integers and a channel, and sends the sum of the integers on the channel. A client process can be written like $(\nu y)\,(\overline{sum}\langle 1, 2, y\rangle \,|\, y(x)^{\mathbf{c}}.\,P)$. The attribute $\mathbf{c}$ of the input process specifies that the input process should eventually receive a result on channel y.

Example 2.2. 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 $((\nu x)\,(\overline{x}\langle\,\rangle \,|\, P))$. A semaphore can be acquired by extracting a value from the channel $(x(\,).\,Q)$, and released by putting a value back into the channel $(\overline{x}\langle\,\rangle)$. If we want to explicitly specify that a semaphore x can be eventually acquired, we can annotate an 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 [21]. For subtle technical reasons, we introduce a structural preorder $\preceq$ instead of a structural congruence relation. The differences from the usual structural congruence $\equiv$ are that neither $(\nu x)\,(P \,|\, Q) \preceq (\nu x)\,P \,|\, Q$ nor $*P \,|\, P \preceq *P$ holds and that $\preceq$ is not closed under output and input prefixes, replications, and conditionals. The reason for not allowing $(\nu x)\,(P \,|\, Q) \preceq (\nu x)\,P \,|\, Q$ is described in Remark 2.3. The reason for not allowing $*P \,|\, P \preceq *P$ is that in our type system given in Section 3, $*P \,|\, P$ and $*P$ are not always well-typed under the same type environment.

Definition 2.2. The *structural preorder* $\preceq$ is the least reflexive and transitive relation closed under the following rules ($P \equiv Q$ denotes $(P \preceq Q) \wedge (Q \preceq P)$):

$$\frac{P \equiv_\alpha Q}{P \equiv Q} \qquad \text{(SPCong-Alpha)}$$

$$P \,|\, \mathbf{0} \equiv P \qquad \text{(SPCong-Zero)}$$

$$P \,|\, Q \equiv Q \,|\, P \qquad \text{(SPCong-Commut)}$$

$$P \,|\, (Q \,|\, R) \equiv (P \,|\, Q) \,|\, R \qquad \text{(SPCong-Assoc)}$$

$$(\nu x)\, P \,|\, Q \preceq (\nu x)\, (P \,|\, Q) \quad \text{(if } x \text{ is not free in } Q) \qquad \text{(SPCong-New)}$$

$$*P \preceq *P \,|\, P \qquad \text{(SPCong-Rep)}$$

$$\frac{P \preceq P' \qquad Q \preceq Q'}{P \,|\, Q \preceq P' \,|\, Q'} \qquad \text{(SPCong-Par)}$$

$$\frac{P \preceq Q}{(\nu x)\, P \preceq (\nu x)\, Q} \qquad \text{(SPCong-CNew)}$$

We write $P \preceq^{-\alpha} Q$ when $P \preceq Q$ can be derived without using rule (SPCong-Alpha).

Now we define the reduction relation. Following the operational semantics of the linear π-calculus [16], we define the reduction relation as a ternary relation $P \xrightarrow{l} Q$. l describes the channel on which 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 $\mathbf{com}_x$, which means that the reduction is performed by communication on the free channel x.

Definition 2.3. The reduction relation $\xrightarrow{l}$ is the least relation closed under the following rules:

$$\overline{x}\langle v_1, \ldots, v_n\rangle^a.\, P \,|\, x(z_1, \ldots, z_n)^{a'}.\, Q \xrightarrow{\mathbf{com}_x} P \,|\, [z_1 \mapsto v_1, \ldots, z_n \mapsto v_n]Q \qquad \text{(R-Com)}$$

$$\frac{P \xrightarrow{l} Q}{P \,|\, R \xrightarrow{l} Q \,|\, R} \qquad \text{(R-Par)}$$

$$\frac{P \xrightarrow{\mathbf{com}_x} Q}{(\nu x)\, P \xrightarrow{\epsilon} (\nu x)\, Q} \qquad \text{(R-New1)}$$

$$\frac{P \xrightarrow{l} Q \qquad l \neq \mathbf{com}_x}{(\nu x)\, P \xrightarrow{l} (\nu x)\, Q} \qquad \text{(R-New2)}$$

$$\textbf{if } \mathit{true} \textbf{ then } P \textbf{ else } Q \xrightarrow{\epsilon} P \qquad \text{(R-IfT)}$$

$$\textbf{if } \mathit{false} \textbf{ then } P \textbf{ else } Q \xrightarrow{\epsilon} Q \qquad \text{(R-IfF)}$$

$$\frac{P \preceq P' \qquad P' \xrightarrow{l} Q' \qquad Q' \preceq Q}{P \xrightarrow{l} Q} \qquad \text{(R-SPCong)}$$

NOTATION 2.2. 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 Lock-Freedom

Based on the operational semantics, we formally define deadlock-freedom and lock-freedom. Basically, we regard a process as locked if one of its subprocesses is trying to communicate with some process but is 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, there should be no problem even if 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 2.2), because it means that no process tries to acquire the semaphore. Therefore, we focus on communications that are expected to succeed by a programmer, i.e., those annotated with attribute **c**. A process is considered locked if it is trying to perform input or output but is blocked forever, and if the input or output is annotated with **c**.

We first define deadlock.

DEFINITION 2.4 (deadlock, deadlock-freedom). A process P is in *deadlock* if (i) $P \preceq (\nu\tilde{y})\,(x(\tilde{z})^{\mathbf{c}}.\,Q \,|\, R)$ or $P \preceq (\nu\tilde{y})\,(\overline{x}\langle\tilde{z}\rangle^{\mathbf{c}}.\,Q \,|\, R)$ 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 2.2. In the usual terminology, deadlock often refers to a more restricted state, where processes are blocked forever because of *cyclic dependencies on communications*. As the definition above shows, in this paper, deadlock refers to a state where processes are blocked forever, regardless of cyclic dependencies.

EXAMPLE 2.3. $(\nu x)\,(x(\,)^{\mathbf{c}}.\,\mathbf{0})$ is in deadlock because the input from x is annotated with **c** but there is no output process. $(\nu x)\,(\nu y)\,(x(\,)^{\mathbf{c}}.\,\overline{y}\langle\,\rangle \,|\, y(\,).\,\overline{x}\langle\,\rangle)$ 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 $(\nu x)\,(\overline{x}\langle\,\rangle \,|\, x(\,)^{\mathbf{c}}.\,\mathbf{0})$ are not in deadlock.

Example 2.4. A process $(\nu x)\,(x(\,)^{\mathbf{c}}.\,\mathbf{0}\,|\,(\nu y)\,(\overline{y}\langle x\rangle\,|\,{*y(z)}.\,\overline{y}\langle z\rangle))$ is deadlock-free. Although the input from x never succeeds, the entire process is never blocked. In fact, our previous type systems for deadlock-freedom [17, 34] judges this process to be well-typed, hence, deadlock-free.

The last example shows a weakness of the deadlock-freedom property: Even if a process is deadlock-free, communications may not be guaranteed to succeed eventually. The lock-freedom property defined below requires that every communication annotated with **c** eventually succeeds, regardless of whether processes diverge.

Before defining the lock-freedom property, we make an assumption about scheduling. Let us consider the process $\overline{x}\langle \mathit{true}\rangle^{\mathbf{c}}\,|\,{*\overline{x}\langle \mathit{false}\rangle}\,|\,{*x(y)}\,.\,\mathbf{0}$. If we do not make any assumption about scheduling, the process $\overline{x}\langle \mathit{true}\rangle^{\mathbf{c}}$ is not guaranteed to succeed because ${*x(y)}\,.\,\mathbf{0}$ may always communicate with ${*\overline{x}\langle \mathit{false}\rangle}$. However, since ${*x(y)}\,.\,\mathbf{0}$ is always listening on channel x, it would be reasonable to expect that the process $\overline{x}\langle \mathit{true}\rangle^{\mathbf{c}}$ actually succeeds to output eventually; hence it is reasonable to consider the process lock-free. Thus, we assume that scheduling is *strongly fair* [7, 3] in the sense that every process that is enabled to participate in a communication infinitely many times can eventually participate in a communication. Strong fairness is actually implemented in programming language Pict [28, 35].

Note that weak fairness [7, 3], which says that every process that is *continuously* enabled to participate in a communication can eventually participate in a communication, is insufficient for our purpose. Let us consider the following process.

$$\overline{x}\langle\,\rangle\,|\,x(\,)^{\mathbf{c}}.\,P\,|\,{*x(\,)}.\,y(\,).\,\overline{x}\langle\,\rangle\,|\,{*\overline{y}\langle\,\rangle}$$

Here, x is a channel used as a binary semaphore (recall Example 2.2), and $x(\,)^{\mathbf{c}}.\,P$ is trying to acquire the semaphore. Since ${*x(\,)}.\,y(\,).\,\overline{x}\langle\,\rangle$ always releases the semaphore after acquiring it, it is reasonable to expect that the process $x(\,)^{\mathbf{c}}.\,P$ can eventually acquire the semaphore. However, that is not guaranteed by weak fairness: The process $x(\,)^{\mathbf{c}}.\,P$ is not able to participate in a communication continuously, because the process is reduced to

$$x(\,)^{\mathbf{c}}.\,P\,|\,y(\,).\,\overline{x}\langle\,\rangle\,|\,{*x(\,)}.\,y(\,).\,\overline{x}\langle\,\rangle\,|\,{*\overline{y}\langle\,\rangle},$$

There are subtle problems in formally stating the fairness assumption based on the reduction semantics defined in Section 2.2. In order to state that a certain process has succeeded to communicate, we must identify the process correctly. However, different processes may be confused because of the following reasons.

- Confusion of different channels. Because α-conversion is allowed in reductions, different channels may be confused. For example, consider the following reduction:

$$(\nu x)\,(x(\,).\,\mathbf{0}\,|\,R_1)\,|\,(\nu y)\,(y(\,).\,\mathbf{0}\,|\,R_2)\longrightarrow(\nu x)\,(\nu y)\,(x(\,).\,\mathbf{0}\,|\,y(\,).\,\mathbf{0}\,|\,R).$$

Because of the possibility of α-conversion, we do not know whether $x(\,).\,\mathbf{0}$ in the lefthand process corresponds to $x(\,).\,\mathbf{0}$ or $y(\,).\,\mathbf{0}$ in the righthand process.

• Confusion of structurally congruent processes. Consider the following reduction sequence:

$$\begin{array}{rl} x(\,).\,\mathbf{0} \,|\, *x(\,).\,\mathbf{0} \,|\, *\overline{x}\langle\,\rangle & \preceq \;\; x(\,).\,\mathbf{0} \,|\, (*x(\,).\,\mathbf{0} \,|\, x(\,).\,\mathbf{0}) \,|\, (*\overline{x}\langle\,\rangle \,|\, \overline{x}\langle\,\rangle) \\ & \preceq \;\; x(\,).\,\mathbf{0} \,|\, *x(\,).\,\mathbf{0} \,|\, *\overline{x}\langle\,\rangle \,|\, (x(\,).\,\mathbf{0} \,|\, *\overline{x}\langle\,\rangle) \\ & \longrightarrow \;\; x(\,).\,\mathbf{0} \,|\, *x(\,).\,\mathbf{0} \,|\, *\overline{x}\langle\,\rangle \\ & \longrightarrow \;\; x(\,).\,\mathbf{0} \,|\, *x(\,).\,\mathbf{0} \,|\, *\overline{x}\langle\,\rangle \\ & \longrightarrow \;\; \cdots \end{array}$$

It is unclear whether the leftmost process $x(\,).\,\mathbf{0}$ or a copy of $*x(\,).\,\mathbf{0}$ is reduced in each step.

To deal with the first problem above, we forbid α-conversion on top-level ν-prefixes (which stand for already created channels) in normal reduction sequences defined below (Definition 2.6). An alternative way to avoid renaming of already created channels is to replace rules (R-New1) and (R-New2) with the following rule for generating a fresh channel:

$$(\nu x)\,P \xrightarrow{\epsilon} [y/x]P \quad (y \text{ fresh})$$

For the second problem we assume that an appropriate process is chosen when there are structurally congruent processes; In the example above, we assume that there is a step in which the leftmost process is chosen. If we want to avoid the problem completely, we can tag each process to distinguish between structurally congruent processes. For example, the process $x(\,).\,\mathbf{0} \,|\, *x(\,).\,\mathbf{0} \,|\, *\overline{x}\langle\,\rangle.\,\mathbf{0}$ can be replaced by $x(\,).\,\overline{tag}\langle\,\rangle \,|\, *(\nu tag')\,x(\,).\,\overline{tag'}\langle\,\rangle \,|\, *(\nu tag'')\,\overline{x}\langle\,\rangle.\,\overline{tag''}\langle\,\rangle$. (The process $*(\nu tag')\,x(\,).\,\overline{tag'}\langle\,\rangle$ is further encoded into a valid process using the trick described in Remark 2.1.) Here, to distinguish between different occurrences of the same process, we replaced each occurrence of $\mathbf{0}$ with an output on a tag channel. Alternatively, we can introduce a new construct P^L to denote a process P tagged with L. Since processes are created dynamically, we also need a mechanism for generating fresh tags dynamically. The above solution of using channels as tags takes advantage of the fact that we already have a construct for generating fresh channels.

Definition 2.5 (normal form). A process is in normal form if it is of the form $(\nu\tilde{x})\,(A_1 \,|\, \cdots \,|\, A_n)$ and if the variables $\tilde{x}$ are different from each other and from the free variables of $(\nu\tilde{x})\,(A_1 \,|\, \cdots \,|\, A_n)$. (Here, $(\nu\tilde{x})\,(A_1 \,|\, \cdots \,|\, A_n)$ stands for $(\nu\tilde{x})\,\mathbf{0}$ if $n = 0$.) When a process $P = (\nu\tilde{x})\,(A_1 \,|\, \cdots \,|\, A_n)$ is in normal form, we write $\mathit{NewChan}(P)$ for the sequence $\tilde{x}$. When $P \preceq Q$ and Q is in normal form, we say that Q is a normal form of P.

Definition 2.6 (reduction sequence). Let I be the set $\mathbf{Nat}$ of natural numbers or a subset $\{i \in \mathbf{Nat} \mid 0 \leq i \leq n\}$ for some $n \in \mathbf{Nat}$. A set $\{P_i \mid i \in I\}$ of processes is called a *reduction sequence* if $P_{i-1} \longrightarrow P_i$ holds for every $i \in I\backslash\{0\}$. A reduction sequence $\{P_i \mid i \in I\}$ is *normal* if (i) P_i is in normal form for every $i \in I$, and (ii) $\mathit{NewChan}(P_{i-1})$ is a prefix of $\mathit{NewChan}(P_i)$ for every $i \in I\backslash\{0\}$. A reduction sequence $\{P_i \mid i \in I\}$ is *complete* if $I = \mathbf{Nat}$ or $I = \{i \mid 0 \leq i \leq n\}$ with $P_n \not\longrightarrow$,

We write $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$ for a reduction sequence $\{P_0, P_1, P_2, \ldots \mid P_i \longrightarrow P_{i+1}$ for $i = 0, 1, 2, \ldots\}$.

Remark 2.3. If we defined $(\nu x)\,(P \,|\, Q) \preceq (\nu x)\,P \,|\, Q$ to hold in Definition 2.2, α-conversion on top-level ν-prefixes may occur in a normal reduction sequence. Suppose $P \longrightarrow Q$ with $\{x, y\} \cap FV(P) = \emptyset$. Renaming on x and y are carried out in the following reduction.

$$\begin{array}{rcl}
(\nu x)\,(\nu y)\,(\overline{x}\langle \mathit{true}\rangle \,|\, \overline{y}\langle \mathit{false}\rangle \,|\, P) & \preceq & (\nu x)\,(\overline{x}\langle \mathit{true}\rangle)\,|\,(\nu y)\,(\overline{y}\langle \mathit{false}\rangle \,|\, P) \\
& \preceq & (\nu y)\,(\overline{y}\langle \mathit{true}\rangle)\,|\,(\nu x)\,(\overline{x}\langle \mathit{false}\rangle \,|\, P) \\
& \preceq & (\nu x)\,(\nu y)\,(\overline{x}\langle \mathit{false}\rangle \,|\, \overline{y}\langle \mathit{true}\rangle \,|\, P) \\
& \longrightarrow & (\nu x)\,(\nu y)\,(\overline{x}\langle \mathit{false}\rangle \,|\, \overline{y}\langle \mathit{true}\rangle \,|\, Q).
\end{array}$$

Definition 2.7 (fair reduction sequence). A normal, complete 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 $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_i} \preceq^{-\alpha} (\nu \tilde{w}_i)\,(\overline{x}\langle \tilde{v}\rangle^{a}.\,Q \,|\, x(\tilde{y})^{a_i}.\,Q_i \,|\, R_i)$, then there exists $n \geq n_0$ such that $P_n \preceq^{-\alpha} (\nu \tilde{w})\,(\overline{x}\langle \tilde{v}\rangle^{a}.\,Q \,|\, x(\tilde{y})^{a'}.\,Q' \,|\, R')$ and $(\nu \tilde{w})\,(Q \,|\, [\tilde{y} \mapsto \tilde{v}]Q' \,|\, R') \preceq P_{n+1}$.

(ii)If there exists an infinite increasing sequence $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_i} \preceq^{-\alpha} (\nu \tilde{w}_i)\,(x(\tilde{y})^{a}.\,Q \,|\, \overline{x}\langle \tilde{v}_i\rangle^{a_i}.\,Q_i \,|\, R_i)$, then there exists $n \geq n_0$ such that $P_n \preceq^{-\alpha} (\nu \tilde{w})\,(x(\tilde{y})^{a}.\,Q \,|\, \overline{x}\langle \tilde{v}\rangle^{a'}.\,Q' \,|\, R')$ and $(\nu \tilde{w})\,([\tilde{y} \mapsto \tilde{v}]Q \,|\, Q' \,|\, R') \preceq P_{n+1}$.

(iii)If $P_i \preceq^{-\alpha} (\nu \tilde{w})\,(\mathbf{if}\ v\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2 \,|\, R)$ and $v = \mathit{true}$ or false for some i, then there exists $n \geq i$ such that $P_n \preceq^{-\alpha} (\nu \tilde{w})\,(\mathbf{if}\ v\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2 \,|\, R')$, $(\nu \tilde{w})\,(Q' \,|\, R') \preceq P_{n+1}$, and $Q' = Q_1$ if $v = \mathit{true}$ and $Q' = Q_2$ otherwise.

Now we define the lock-freedom property (relative to the fairness assumption). Intuitively, a process is lock-free if in any fair reduction sequence a process trying to perform communication annotated with $\mathbf{c}$ eventually communicates with another process.

Definition 2.8 (lock-freedom). A process P_0 in normal form is *lock-free* (under fair scheduling) if the following conditions hold for any fair reduction sequence $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$:

1.If $P_i \preceq^{-\alpha} (\nu \tilde{w})\,(\overline{x}\langle \tilde{v}\rangle^{\mathbf{c}}.\,Q \,|\, R)$ for some $i \geq 0$, there exists $n \geq i$ such that $P_n \preceq^{-\alpha} (\nu \tilde{w'})\,(\overline{x}\langle \tilde{v}\rangle^{\mathbf{c}}.\,Q \,|\, x(\tilde{y})^{a}.\,R_1 \,|\, R_2)$ and $(\nu \tilde{w'})\,(Q \,|\, [\tilde{y} \mapsto \tilde{v}]R_1 \,|\, R_2) \preceq P_{n+1}$.

2.If $P_i \preceq^{-\alpha} (\nu \tilde{w})\,(x(\tilde{y})^{\mathbf{c}}.\,Q \,|\, R)$ for some $i \geq 0$, there exists $n \geq i$ such that $P_n \preceq^{-\alpha} (\nu \tilde{w'})\,(x(\tilde{y})^{\mathbf{c}}.\,Q \,|\, \overline{x}\langle \tilde{v}\rangle^{a}.\,R_1 \,|\, R_2)$ and $(\nu \tilde{w'})\,([\tilde{y} \mapsto \tilde{v}]Q \,|\, R_1 \,|\, R_2) \preceq P_{n+1}$.

A process P is defined to be *lock-free* if there is a normal form of P that is lock-free.

Note that lock-freedom is a stronger property than deadlock-freedom.

EXAMPLE 2.5. The deadlocked processes in Example 2.3 are not lock-free. The process in Example 2.4 is deadlock-free but not lock-free. The process $(\nu x)\,(\overline{x}\langle\,\rangle \,|\, x(\,)^{\mathbf{c}}.\,\mathbf{0}) \,|\, (\nu y)\,(\overline{y}\langle\,\rangle \,|\, {*y(\,)}.\,\overline{y}\langle\,\rangle)$ is lock-free: On the fairness assumption, the input from x succeeds eventually.

3. TYPE SYSTEM FOR LOCK-FREEDOM

This section introduces a type system that guarantees the lock-freedom property. We first explain general ideas of our type system (in Section 3.1). Then we define the type system and show that it is sound in the sense that every closed well-typed process is lock-free. The usage of fairness assumption in the soundness proof may be interesting for an expert.

3.1. Basic Ideas

As in our previous type systems for deadlock-freedom [14, 17, 34], we augment channel types with information about how each channel is used. In our previous type systems, the type of a channel used for exchanging integers is of the form $[\mathit{int}]/U$, where the part U, called a *usage*, describes how the channel is used for input and output. The usages are defined by the following grammar in [34]:

$$\begin{array}{lll} U \text{ (usages)} & ::= & \mathbf{0} \mid I_a.U \mid O_a.U \mid (U_1 \,|\, U_2) \mid *U \\ a \text{ (attributes)} & \subseteq & \{\mathbf{c}, \mathbf{o}\} \end{array}$$

The usage $\mathbf{0}$ describes a channel that cannot be used at all. $I_a.U$ and $O_a.U$ describe channels that can be first used for input and output respectively and then used according to U. An attribute a attached to I or O expresses whether a channel of that usage *must* be used for input or output ($\mathbf{o} \in a$ in that case) and whether the input or output is guaranteed to succeed ($\mathbf{c} \in a$ in that case). $U_1 \,|\, U_2$ describes a channel that is used according to U_1 by one process and according to U_2 by another process. A channel of usage $*U$ can be used according to U by infinitely many processes. The usage of each channel tells which communication may or may not succeed. For example, suppose that a channel has usage $I_{a_1}.\mathbf{0} \,|\, I_{a_1}.\mathbf{0} \,|\, O_{a_2}.\mathbf{0}$. Since there is only one O, we know that at least one of the two inputs will fail. On the other hand, we know that if one of the two inputs is guaranteed to be executed, the output is guaranteed to succeed. That is one of the key ideas behind our previous type systems for deadlock-freedom [14, 17, 34].

To ensure the lock-freedom property, we replace an attribute a above with more precise information: how many reduction steps it takes for a process to become ready to input or output a value on the channel, and how many steps it takes for the process to find a communication partner after it becomes ready to communicate. For example, the usage $I^{t_1}_{t_2}.\mathbf{0}$ of a channel means that some process must become ready to input a value on the channel within t_1 steps, and that once a process becomes ready to input a value on the channel, it succeeds to find an output process to communicate with within t_2 steps. For example, the usage of channel x in the process $x(\,).\,\mathbf{0} \,|\, y(\,).\,\overline{x}\langle\,\rangle \,|\, \overline{y}\langle\,\rangle$ is expressed by $I^0_1.\mathbf{0} \,|\, O^1_0.\mathbf{0}$: The part $I^0_1.\mathbf{0}$ means that a process becomes ready to input a value immediately (since $x(\,).\,\mathbf{0}$ appears at the top-level), and that the input process may have to wait for one step (since the communication on y must complete before the output process $\overline{x}\langle\,\rangle$ is executed).

To see how a lock is detected, consider a (dead)locked process $x(\,)^{\mathbf{c}}.\,\overline{y}\langle\,\rangle \,|\, y(\,).\,\overline{x}\langle\,\rangle$. Suppose that the usage of x is $I^{t_1}_{t_2}.\mathbf{0} \,|\, O^{t_3}_{t_4}.\mathbf{0}$ and that of y is $I^{t_5}_{t_6}.\mathbf{0} \,|\, O^{t_7}_{t_8}.\mathbf{0}$. Since the input process $x(\,)^{\mathbf{c}}.\,\overline{y}\langle\,\rangle$ must wait until the output process $\overline{x}\langle\,\rangle$ is executed, it must be the case that $t_3 \leq t_2$. Similarly, we have the constraint $t_7 \leq t_6$ on the usage of y. Moreover, from the sub-process $x(\,)^{\mathbf{c}}.\,\overline{y}\langle\,\rangle$, we know that y is used for output only after the input on x succeeds. So, it must be the case that $t_2 < t_7$, since t_7 should be greater than or equal to t_2 (the number of steps required for the input process on x to find a communication partner), plus another step required for the communication on x to complete. Similarly, we get the constraint $t_6 < t_3$ from the other sub-process $y(\,).\,\overline{x}\langle\,\rangle$. From these constraints, we have $t_2 < t_7 \leq t_6 < t_3 \leq t_2$, a contradiction. Thus, a finite upper-bound on the number of reduction steps cannot be assigned to the input on x, hence we can reason that the process may not be lock-free.

As another example, consider a (live)locked process $x(\,)^{\mathbf{c}}.\,\mathbf{0} \,|\, \overline{y}\langle x\rangle \,|\, {*y(z)}.\,\overline{y}\langle z\rangle$. Suppose that y is a channel used for communicating a channel of usage $O^{t_1}_{t_2}.\mathbf{0}$ and that it takes $t_3(>0)$ steps for a message sent on y to be received. The subprocess $y(z).\,\overline{y}\langle z\rangle$ receives a channel z of usage $O^{t_1}_{t_2}.\mathbf{0}$, so that it must be guaranteed that z is used for output within t_1 steps after the reception. However, since it resends z on y, it takes t_3 steps for z to be received by another process, and t_1 steps for z to be used for output by the process. So, it must be the case that $t_1 + t_3 \leq t_1$, a contradiction. Therefore, we know that the whole process may not be lock-free.

3.2. Usages and Types

Based on the ideas explained in the previous section, we define a type system that enables systematic reasoning about lock-freedom. We first define the formal syntax of usages and types.

DEFINITION 3.9 (usages). The set $\mathcal{U}$ of usages is given by the following syntax.

$$\begin{array}{l} U ::= \mathbf{0} \mid \alpha^{t_o}_{t_c}.U \mid (U_1 \,|\, U_2) \mid *U \\ \alpha ::= I \mid O \\ t_o, t_c \in \mathbf{Nat}^+ \end{array}$$

Here, $\mathbf{Nat}^+$ denotes the set $\mathbf{Nat} \cup \{\infty\}$.

As explained in the previous subsection, $\alpha^{t_o}_{t_c}.U$ means that there is an *obligation* to execute the action α within time t_o (where the *execution* means that a process becomes ready to perform the action, not that it actually succeeds in communicating with another process), and then there is a *capability* to successfully find a communication partner and start a communication within time t_c. (It takes another step to complete the communication.) We call t_o the time limit of execution of an action, and t_c the time limit of success of an action. As defined above, a time limit t is either a natural number or ∞. Intuitively, 1 denotes the time required for performing one-step reduction. (Actually, a time limit t denotes abstract length of time rather than the number of reduction steps: see Remark 3.4 below.) The time limit ∞ means that there is no time limit. For example, $O^{\infty}_{t_c}.\mathbf{0}$ means that an output may never be performed. $O^{t_o}_{\infty}.\mathbf{0}$ means that an output may never succeed.

We extend the summation $n_1 + n_2$ of natural numbers by adding the rule $\forall t \in \mathbf{Nat}^+.(\infty + t = t + \infty = \infty)$. We also extend the relation $<$ on natural numbers by adding the rule $\forall n \in \mathbf{Nat}.(n < \infty)$.

Note that the new usages express more precise information than those in the previous type systems [34, 17]. The usages $I_{\{\mathbf{c}\}}.U$ and $I_{\{\mathbf{o}\}}.U$ in the previous type systems are expressed by $I^{\infty}_{t}.U$ and $I^{t}_{\infty}.U$ respectively for some $t \in \mathbf{Nat}$.

Notation 3.3. 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 write $\overline{\alpha}$ for the co-action of α (O is the co-action of I and I is the co-action of O).

Definition 3.10 (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 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 time* t.

Definition 3.11. A unary operation $\boxed{t}$ on usages is defined inductively by:

$$\begin{array}{ll}
\boxed{t}\mathbf{0} = \mathbf{0} & \boxed{t}\alpha^{t_o}_{t_c}.U = \alpha^{t_o+t}_{t_c}.U \\
\boxed{t}(U_1 \,|\, U_2) = \boxed{t}U_1 \,|\, \boxed{t}U_2 & \boxed{t}(*U) = *\boxed{t}U
\end{array}$$

Note that $\boxed{t}\alpha^{t_o}_{t_c}.U$ is not $\alpha^{t_o+t}_{t_c+t}.U$ but $\alpha^{t_o+t}_{t_c}.U$. Usage $\boxed{t}\alpha^{t_o}_{t_c}.U$ means that a channel can be used according to $\alpha^{t_o}_{t_c}.U$ after a delay of *at most* t. So, it may take time $t_o + t$ until the action α is executed. On the other hand, since the action may be executed immediately (without waiting for time t), the action should be guaranteed to succeed within time t_c.

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

Definition 3.12. Operations $|, *, \boxed{t}$ on types are defined by:

$$\begin{array}{ll}
\mathit{bool} \,|\, \mathit{bool} = \mathit{bool} & ([\tilde{\tau}]/U_1) \,|\, ([\tilde{\tau}]/U_2) = [\tilde{\tau}]/(U_1 \,|\, U_2) \\
*\mathit{bool} = \mathit{bool} & *[\tilde{\tau}]/U_1 = [\tilde{\tau}]/*U_1 \\
\boxed{t}\mathit{bool} = \mathit{bool} & \boxed{t}[\tilde{\tau}]/U_1 = [\tilde{\tau}]/\boxed{t}U_1
\end{array}$$

3.3. Reliability of Usages

As in the type systems for deadlock-freedom [17, 34], the usage of each channel must be consistent (reliable) in the sense that the success of each input/output action is guaranteed by an execution of its co-action. For example, consider the usage $I^{\infty}_{t_c}.U_1 \,|\, O^{t_o}_{\infty}.U_2$. In order for an input process to find a communication partner within time t_c, t_o must be shorter than or equal to t_c. This consistency should be preserved during the whole computation. After a communication on a channel of usage $I_{t_c}^{\infty}.U_1 \,|\, O_{\infty}^{t_o}.U_2$ happens, the channel is used according to $U_1 \,|\, U_2$. So, $U_1 \,|\, U_2$ must be also consistent. To state such a condition, we first define *reduction* of a usage. Similarly to the reduction relation on processes defined in Section 2, the reduction relation on usages is defined via a structural relation and a reduction relation.

Definition 3.13. $\cong$ is the least congruence relation satisfying the following rules:

•Laws for $\mathbf{0}$:

$$U \cong U \,|\, \mathbf{0}$$

•Laws for $\,|\,$:

$$U_1 \,|\, U_2 \cong U_2 \,|\, U_1$$

$$(U_1 \,|\, U_2) \,|\, U_3 \cong U_1 \,|\, (U_2 \,|\, U_3)$$

•Laws for $*U$:

$$*\mathbf{0} \cong \mathbf{0}$$

$$*U \cong *U \,|\, U$$

$$*(U_1 \,|\, U_2) \cong *U_1 \,|\, *U_2$$

$$**U \cong *U$$

The last rule does not appear in the usual definition of structural congruence of processes [33] or in the definition of the structural preorder on processes in Section 2. It is better to have this rule, because we use the congruence relation on usages to define not only a reduction relation (Definition 3.14 below) but also a subusage relation (Definition 3.17). This rule allows more processes to be typed: See Remark 3.7.

Definition 3.14 (usage reduction). A binary relation $\longrightarrow$ on usages is the least relation closed under the following rules:

$$I_{t_c}^{t_o}.U_1 \,|\, O_{t'_c}^{t'_o}.U_2 \,|\, U_3 \longrightarrow U_1 \,|\, U_2 \,|\, U_3$$

$$\frac{U_1 \cong U'_1 \qquad U'_1 \longrightarrow U'_2 \qquad U'_2 \cong U_2}{U_1 \longrightarrow U_2}$$

$\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

A usage U is defined to be *reliable* if after any reduction steps, whenever it contains an input/output usage with a time limit t_c of success (i.e., it is structurally congruent to $I_{t_c}^{t_o}.U_1 \,|\, U_2$), it contains an output/input usage with an equal or shorter time limit of execution. We first define predicates ob_α and cap_α to judge whether a usage contains an obligation to execute or a capability to successfully perform an action α within a given time limit, and then give a formal definition of reliability.

Definition 3.15 (obligations and capabilities). The relations $ob_I, ob_O (\subseteq \mathbf{Nat}^+ \times \mathcal{U})$ between a time limit and a usage are defined by: $ob_\alpha(t, U)$ if and only if $t = \infty$ or $\exists t_o, t_c, U_1, U_2.((U \cong \alpha^{t_o}_{t_c}.U_1 \,|\, U_2) \wedge (t_o \leq t))$. The relations $cap_I, cap_O (\subseteq \mathbf{Nat}^+ \times \mathcal{U})$ are defined by: $cap_\alpha(t, U)$ if and only if $t = \infty$ or $\exists t_o, t_c, U_1, U_2.((U \cong \alpha^{t_o}_{t_c}.U_1 \,|\, U_2) \wedge (t_c \leq t))$.

Definition 3.16 (reliability). If $ob_\alpha(t_c, U_2)$ holds whenever $U \longrightarrow^* U'$ and $cap_{\overline{\alpha}}(t_c, U')$, U is called *reliable*, and written $rel(U)$. A type τ is *reliable*, written $rel(\tau)$, if it is of the form $[\tilde{\tau}]/U$ and $rel(U)$ holds.

Reliability $rel(U)$ is decidable: As in a type system for deadlock-freedom [17], the problem of deciding $rel(U)$ can be reduced to the reachability problem of Petri nets [8]. In practice, however, it may be necessary to approximate the problem to solve it efficiently [17].

Remark 3.4. Because of the definitions above, time limits in this section should be considered to express not the number of reduction steps but more abstract length of time. Consider a usage $U = O^t_\infty.\mathbf{0} \,|\, {*I^\infty_t}.O^t_\infty.\mathbf{0}$. This is the usage of a binary semaphore (recall Example 2.2): It means that every process can acquire the lock within time t and that every process having acquired the lock will release the lock within t. Although the usage U is reliable, actually some input action (the acquisition of the lock) may not succeed in t reduction steps: Suppose that t expresses the number of reduction steps, and that multiple processes are simultaneously trying to acquire the lock. Although the lock is always released within t reduction steps, if one process succeeds to acquire the lock, the other waiting processes must wait again until t steps of reduction are performed. We will redefine reliability in Section 4, so that a time limit exactly corresponds to an upper-bound of the number of (parallel) reduction steps.

3.4. Subusage and subtyping

The subusage relation defined below allows one usage to be viewed as another usage. Consider a usage $I^2_2.\mathbf{0}$. It means that an input must be executed within time 2, and after that, the input is guaranteed to succeed within time 2. If one has a channel of that usage, it is safe to assume that an input must be executed within shorter time, e.g., 1, and that the input is guaranteed to succeed within longer time, e.g., 3. So, $I^2_2.\mathbf{0}$ is a subusage of $I^1_3.\mathbf{0}$ and written $I^2_2.\mathbf{0} \leq I^1_3.\mathbf{0}$.

Definition 3.17. The *subusage* relation $\leq$ on usages is the least reflexive and transitive relation closed under the following rules:

$$\frac{U \cong U'}{U \leq U'} \qquad \text{(SubU-Cong)}$$

$$\alpha^{\infty}_{t_c}.U \leq \mathbf{0} \qquad \text{(SubU-Zero)}$$

$$\alpha^{t_o}_{t_c}.U_1 \,|\, \boxed{t_o + t_c + 1} U_2 \leq \alpha^{t_o}_{t_c}.(U_1 \,|\, U_2) \qquad \text{(SubU-Delay)}$$

$$\frac{U \leq U' \qquad t'_o \leq t_o \qquad t_c \leq t'_c}{\alpha^{t_o}_{t_c}.U \leq \alpha^{t'_o}_{t'_c}.U'} \qquad \text{(SubU-Act)}$$

$$\frac{U_1 \leq U'_1 \qquad U_2 \leq U'_2}{U_1 \,|\, U_2 \leq U'_1 \,|\, U'_2} \qquad \text{(SubU-Par)}$$

$$\frac{U \leq U'}{*U \leq *U'} \qquad \text{(SubU-Rep)}$$

Rule (SubU-Zero) indicates that a channel whose time limit of execution is ∞ need not be used at all. Rule (SubU-Delay) allows some usage to be delayed until a communication succeeds. For example, consider the usage $I^1_1.\mathbf{0} \,|\, O^4_1.\mathbf{0}$. The part $I^1_1.\mathbf{0}$ implies that an input is executed within time 1 and then it succeeds within time 1 (plus time 1 before the input completes). So, it is fine that an output is performed within time 1 *after* the input succeeds. Therefore, $I^1_1.\mathbf{0} \,|\, O^4_1.\mathbf{0}$ can be considered a subusage of $I^1_1.(\mathbf{0} \,|\, O^1_1.\mathbf{0})$, which says that an output should be executed only after an input succeeds. Note that the converse does not hold: $I^1_1.(\mathbf{0} \,|\, O^1_1.\mathbf{0})$ also implies that an output can be executed *only after* an input succeeds, while $I^1_1.\mathbf{0} \,|\, O^4_1.\mathbf{0}$ allows an output to be executed immediately. Rule (SubU-Act) means that it is safe to assume that a time limit of execution is shorter than the actual time limit, while the time limit of success can be assumed to be longer than the actual one.

We conjecture that the subusage relation is decidable (although we have not proved it yet). A non-trivial point is that in order to check whether $U \leq U'_1 \,|\, U'_2$ holds by using rule (SubU-Par), we have to split U appropriately. To overcome the problem, we can use a technique used in proof search in linear logic [11, 14].

Remark 3.5. In our previous type system for deadlock-freedom [17], the subusage relation was defined co-inductively by using a simulation relation. This paper uses the axiomatic definition for simplicity.

The subusage relation is extended to the following subtyping relation.

Definition 3.18 (subtyping). A subtyping relation $\leq$ is the least relation closed under the following rules:

$$\mathit{bool} \leq \mathit{bool} \qquad \text{(SubT-Bool)}$$

$$\frac{U \leq U'}{[\tilde{\tau}]/U \leq [\tilde{\tau}]/U'} \qquad \text{(SubT-Chan)}$$

REMARK 3.6. Instead of rule (SUBT-CHAN), it is possible to use the following more general rule, as in [25]:

$$\frac{\begin{array}{c} U \leq U' \\ \tilde{\tau} \leq \tilde{\tau'} \text{ if } U' \text{ contains } I \\ \tilde{\tau'} \leq \tilde{\tau} \text{ if } U' \text{ contains } O \end{array}}{[\tilde{\tau}]/U \leq [\tilde{\tau'}]/U'}$$

The resulting type system allows more processes to be typed. For simplicity, however, we did not choose this rule.

DEFINITION 3.19. A unary predicate *noob* on types is defined by: $noob(\tau)$ if and only if $\tau = bool$ or $\tau = [\tilde{\tau}]/U$ and $U \leq \mathbf{0}$.

3.5. Type Environment

A type environment is a mapping from a finite set of variables to types. We use a meta-variable Γ to denote a type environment. The domain of Γ is denoted by $dom(\Gamma)$. 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 $y \in dom(\Gamma)$. We extend the notation to $\Gamma, v\!:\!bool$. If $\tau = bool$, then $\Gamma, b\!:\!\tau$ (where $b \in \{true, false\}$) denotes the same type environment as Γ; otherwise $\Gamma, b\!:\!\tau$ is undefined. We abbreviate $\emptyset, v_1\!:\!\tau_1, \ldots, v_n\!:\!\tau_n$ to $v_1\!:\!\tau_1, \ldots, v_n\!:\!\tau_n$. We write $\Gamma\backslash S$ for the type environment Γ' such that $dom(\Gamma') = dom(\Gamma)\backslash S$ and $\Gamma'(x) = \Gamma(x)$ for each $x \in dom(\Gamma')$.

The operations and relations on usages or types are pointwise extended to those on type environments as follows.

DEFINITION 3.20 (operations on type environments). The operations $|, *, \boxed{t}$ on type environments are defined by:

$$\begin{array}{l}
dom(\Gamma_1 \,|\, \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2) \\
(\Gamma_1 \,|\, \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \,|\, \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1)\backslash dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2)\backslash dom(\Gamma_1) \end{cases} \\
dom(*\Gamma) = dom(\Gamma) \\
(*\Gamma)(x) = *(\Gamma(x)) \\
dom(\boxed{t}\,\Gamma) = dom(\Gamma) \\
(\boxed{t}\,\Gamma)(x) = \boxed{t}\,(\Gamma(x))
\end{array}$$

DEFINITION 3.21. A type environment Γ is reliable, written $rel(\Gamma)$, if $rel(\Gamma(x))$ holds for each $x \in dom(\Gamma)$.

$$\emptyset \vdash \mathbf{0} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma, x:[\tau_1,\ldots,\tau_n]/U \vdash P \qquad a = \mathbf{c} \Rightarrow t_c < \infty}{\boxed{t_c+1}(v_1:\tau_1 \mid \cdots \mid v_n:\tau_n \mid \Gamma) \mid x:[\tau_1,\ldots,\tau_n]/O^0_{t_c}.U \vdash \overline{x}\langle v_1,\ldots,v_n\rangle^a.\,P} \qquad \text{(T-Out)}$$

$$\frac{\Gamma, x:[\tau_1,\ldots,\tau_n]/U, y_1:\tau_1,\ldots,y_n:\tau_n \vdash P \qquad a = \mathbf{c} \Rightarrow t_c < \infty}{(\boxed{t_c+1}\Gamma), x:[\tau_1,\ldots,\tau_n]/I^0_{t_c}.U \vdash x(y_1,\ldots,y_n)^a.\,P} \qquad \text{(T-In)}$$

$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \mid \Gamma_2 \vdash P_1 \mid P_2} \qquad \text{(T-Par)}$$

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

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{(\boxed{1}\Gamma) \mid v:\mathit{bool} \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q} \qquad \text{(T-If)}$$

$$\frac{\Gamma \vdash P}{*\Gamma \vdash *P} \qquad \text{(T-Rep)}$$

$$\frac{\Gamma' \vdash P \qquad \Gamma \leq \Gamma'}{\Gamma \vdash P} \qquad \text{(T-Weak)}$$

FIG. 1. Typing Rules

Definition 3.22. $\mathit{ob}_x(t,\Gamma)$, $\mathit{ob}_{\overline{x}}(t,\Gamma)$, $\mathit{cap}_x(t,\Gamma)$, and $\mathit{cap}_{\overline{x}}(t,\Gamma)$ are defined by:

$$\begin{aligned}\mathit{ob}_{x^\alpha}(t,\Gamma) &\Longleftrightarrow \exists\tilde{\tau}, U.(\Gamma(x) = [\tilde{\tau}]/U \wedge \mathit{ob}_\alpha(t,U))\\ \mathit{cap}_{x^\alpha}(t,\Gamma) &\Longleftrightarrow \exists\tilde{\tau}, U.(\Gamma(x) = [\tilde{\tau}]/U \wedge \mathit{cap}_\alpha(t,U))\end{aligned}$$

Here, x^α denotes x if $\alpha = I$ and $\overline{x}$ if $\alpha = O$.

The subtyping relation is extended to a relation on type environments below. $\Gamma_1 \leq \Gamma_2$ means that Γ_1 represents a more liberal usage of free channels than Γ_2. For example,

$$(x:[\,]/I^\infty_\infty.\mathbf{0}, y:[\,]/I^\infty_\infty.\mathbf{0}) \leq x:[\,]/I^0_\infty.\mathbf{0}$$

holds. The lefthand type environment means that x and y can be used for input but that they need not be used (since the time limit for execution is ∞). Therefore, there is no problem even if x is used immediately and y is not used at all as specified by the righthand type environment.

Definition 3.23. A binary relation $\leq$ on type environments is defined by: $\Gamma_1 \leq \Gamma_2$ if and only if (i) $\mathit{dom}(\Gamma_1) \supseteq \mathit{dom}(\Gamma_2)$, (ii) $\Gamma_1(x) \leq \Gamma_2(x)$ for each $x \in \mathit{dom}(\Gamma_2)$, and (iii) $\mathit{noob}(\Gamma_1(x))$ for each $x \in \mathit{dom}(\Gamma_1)\backslash\mathit{dom}(\Gamma_2)$.

3.6. Typing Rules

A type judgment is of the form $\Gamma \vdash P$, which means that P is well typed under Γ. Here, we assume that the bound variables in P are always different from each other and the variables in $dom(\Gamma)$. are given in Figure 1. Basically, we accumulate information about the usage of a channel in type environments, and check the consistency of the accumulated information in rule (T-New). We explain each rule below.

(T-Zero). The process $\mathbf{0}$ does nothing. So, it is well-typed under the empty type environment.

(T-Out). The left premise implies that P uses x as a channel of usage U and uses other variables according to Γ. Since $\overline{x}\langle v_1,\dots,v_n\rangle^a.\,P$ uses x for output before that, the total usage of x is expressed by $O^0_{t_c}.U$. (Here, the time limit of execution of an output can be 0 since an output is executed right now.) Other variables are used by P and the receiver of $[v_1,\dots,v_n]$ according to $v_1\!:\!\tau_1 \mid \cdots \mid v_n\!:\!\tau_n \mid \Gamma$ only after the communication on x succeeds. We use $v_1\!:\!\tau_1 \mid \cdots \mid v_n\!:\!\tau_n \mid \Gamma$ instead of $\Gamma, v_1\!:\!\tau_1,\dots,v_n\!:\!\tau_n$ because v_i and v_j may refer to the same variables. It may take time t_c until the communication is enabled and it takes time 1 before the communication completes. Therefore, the total use of other variables is expressed by $\boxed{t_c+1}(v_1\!:\!\tau_1 \mid \cdots \mid v_n\!:\!\tau_n \mid \Gamma)$. The righthand premise requires that the time limit of success must be finite if the output is annotated with $\mathbf{c}$.

(T-In). This is similar to rule (T-Out). Since P uses x according to the usage U, the total usage of x is expressed by $I^0_{t_c}.U$. Also, since $x(y_1,\dots,y_n)^a.\,P$ executes P only after the communication on x has completed, the total use of other variables is expressed by $\boxed{t_c+1}\Gamma$.

(T-Par). The premises imply that P_1 uses variables as specified by Γ_1, and in parallel to this, P_2 uses variables as specified by Γ_2. So, the type environment of $P_1 \mid P_2$ should be expressed as the combination $\Gamma_1 \mid \Gamma_2$. For example, if $\Gamma_1 = x\!:\![\,]/I^{t_{o1}}_{t_{c1}}.\mathbf{0}$ and $\Gamma_2 = x\!:\![\,]/O^{t_{o2}}_{t_{c2}}.\mathbf{0}$, then $P_1 \mid P_2$ should be well typed under $x\!:\![\,]/(I^{t_{o1}}_{t_{c1}}.\mathbf{0} \mid O^{t_{o2}}_{t_{c2}}.\mathbf{0})$.

(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 time 1 to check whether v is *true* or *false*, the total use of variables in **if** v **then** P **else** Q is expressed by $\boxed{1}\Gamma \mid v\!:\!bool$.

(T-Rep). The process $*P$ runs infinitely many copies of P in parallel and the premise implies that each P uses variables as specified by Γ. Therefore, $*P$ uses variables as specified by $*\Gamma$ as a whole.

(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.

Remark 3.7. Consider a process $**\overline{x}\langle\,\rangle$. If the rule $**U \cong *U$ in Definition 3.13 were not allowed, we would not be able to derive $x\!:\![\,]/*O^{t_o}_{t_c}.\mathbf{0} \vdash **\overline{x}\langle\,\rangle$, although the process exhibits the same behavior as $*\overline{x}\langle\,\rangle$ and $x\!:\![\,]/*O^{t_o}_{t_c}.\mathbf{0} \vdash *\overline{x}\langle\,\rangle$ holds.

3.7. Type Soundness

We show that our type system is sound in the sense that any closed well-typed process is lock-free (Corollary 3.6). We need a proof technique that is different from those used in the usual proofs of type soundness. Usually, type soundness is a safety property that nothing bad happens; hence it follows immediately from a subject reduction property that well-typedness is preserved by reductions. On the other hand, soundness of our type system is a liveness property that a communication succeeds eventually under fair scheduling. We show this property by using the subject reduction theorem (Theorem 3.1) and a property that some progress is always made by reduction (Lemma 3.2).

As in the linear π-calculus [16], the type environment of a process may change during reduction. For example, while a process $\overline{x}\langle\,\rangle \,|\, x(\,).\,\mathbf{0}$ is well typed under $x\!:\![\,]/(O^{t_o}_{t_c}.\mathbf{0} \,|\, I^{t'_o}_{t'_c}.\mathbf{0})$, the reduced process $\mathbf{0}$ is well typed under $x\!:\![\,]/\mathbf{0}$. This change of a type environment is captured by the following relation $\Gamma \xrightarrow{l} \Delta$.

Definition 3.24. A ternary relation $\Gamma \xrightarrow{l} \Delta$ is defined to hold if one of the following conditions holds:

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

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

Theorem 3.1 (subject reduction). *If $\Gamma \vdash P$ and $P \xrightarrow{l} Q$, then there exists Δ such that $\Delta \vdash Q$ and $\Gamma \xrightarrow{l} \Delta$.*

Proof. See Appendix A.1.2. ■

The following lemma implies that if a process has an obligation to execute an input/output action within a certain time limit but is waiting for the success of some communication, it fulfills the obligation within a shorter time limit after the success of the communication.

Lemma 3.2. *If $\Gamma, x\!:\![\tilde{\tau}]/U \vdash \overline{x}\langle\tilde{v}\rangle^a.\,P \,|\, x(\tilde{y})^{a'}.\,Q$, then there exist Δ and U' such that $\Delta, x\!:\![\tilde{\tau}]/U' \vdash P \,|\, [\tilde{y} \mapsto \tilde{v}]Q$ and $\Gamma \leq \boxed{1}\Delta$ with $U \longrightarrow U'$.*

Proof. See Appendix A.1.2. ■

The following main theorem says that if the type environment of a process contains an obligation to execute an input or output action, the action is indeed executed eventually (property A below), and that if the type environment contains a capability to successfully complete an input or output action, the action indeed succeeds eventually (property B).

THEOREM 3.3. *Suppose that* $P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \cdots$ *is a fair reduction sequence and that* $\Gamma \vdash P_0$ *holds for some* Γ *with* $rel(\Gamma)$. *If* $t < \infty$, *the following properties hold.*

A(Obligations). If $ob_{\overline{x}}(t,\Gamma)$ ($ob_x(t,\Gamma)$, *resp.*), *then there exists* $n \geq 0$ *such that* $P_n \preceq^{-\alpha} (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle^a.\,Q_1 \,|\, Q_2)$ ($P_n \preceq^{-\alpha} (\nu\tilde{w})\,(x(\tilde{y})^a.\,Q_1 \,|\, Q_2)$, *resp.*).

B(Capabilities). Suppose that P_0 *is of the form* $P_{01} \,|\, P_{02}$. *Suppose also that* $\Gamma_1 \vdash P_{01}$ *and* $\Gamma_2 \vdash P_{02}$ *hold with* $\Gamma = \Gamma_1 \,|\, \Gamma_2$.

–If $P_{01} = \overline{x}\langle\tilde{v}\rangle^a.Q$ *and* $cap_{\overline{x}}(t,\Gamma_1)$, *then there exists* $n \geq 0$ *such that* $P_n \preceq^{-\alpha} (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle^a.\,Q \,|\, x(\tilde{y})^{a'}.\,R_1 \,|\, R_2)$ *and* $(\nu\tilde{w})\,(Q \,|\, [\tilde{y} \mapsto \tilde{v}]R_1 \,|\, R_2) \preceq P_{n+1}$.

–If $P_{01} = x(\tilde{y})^a.Q$ *and* $cap_x(t,\Gamma_1)$, *then there exists* $n \geq 0$ *such that* $P_n \preceq^{-\alpha} (\nu\tilde{w})\,(x(\tilde{y})^a.\,Q \,|\, \overline{x}\langle\tilde{v}\rangle^{a'}.\,R_1 \,|\, R_2)$ *and* $(\nu\tilde{w})\,([\tilde{y} \mapsto \tilde{v}]Q \,|\, R_1 \,|\, R_2) \preceq P_{n+1}$.

We need some auxiliary lemmas to prove the theorem. Lemma 3.4 means that if a usage U says that some action must be executed within time t and if U is a subusage of V, then V also guarantees that the action is executed with the same time limit. The action may be executed only after another action is executed (recall rule (SUBU-DELAY)); this is taken care of by the case (ii) of the following lemma..

LEMMA 3.4. *If* $ob_\alpha(t,U)$ *and* $U \leq V$, *then either (i)* $ob_\alpha(t,V)$ *holds or (ii)* $cap_{\overline{\alpha}}(t_c,U)$ *holds for some* t_c *such that* $t_c < t$.

Proof. See Appendix A.1.1. ∎

The following lemma states that reliability is preserved by usage reductions and the subusage relation. The relation $\longrightarrow^*\leq$ is the composition of the relations $\longrightarrow^*$ and $\leq$.

LEMMA 3.5. *If* $rel(U)$ *and* $U \longrightarrow^*\leq V$, *then* $rel(V)$ *also holds. If* $rel(\Gamma)$ *and* $\Gamma \longrightarrow^*\leq \Delta$, *then* $rel(\Delta)$ *also holds.*

Proof. See Appendix A.1.1. ∎

Proof (Proof of Theorem 3.3). We prove this theorem by course-of-values induction on t. Suppose that the theorem holds for any t' such that $t' < t$.

A. We show only the case for $ob_O(t,U)$. The other case is similar. By the assumption $\Gamma \vdash P$, there exist P_{01} and P_{02} such that

$$\begin{array}{l}
P_{01} \text{ is an output, an input, or a conditional process}\\
P_0 \preceq^{-\alpha} (\nu\tilde{w})\,(P_{01} \,|\, P_{02})\\
\Delta_1, \tilde{w}:\tilde{\sigma_1}, x:[\tilde{\tau}]/U_1 \vdash P_{01}\\
\Delta_2, \tilde{w}:\tilde{\sigma_2}, x:[\tilde{\tau}]/U_2 \vdash P_{02}\\
ob_O(t,U_1)\\
\Gamma \leq \Delta_1 \,|\, \Delta_2, x:[\tilde{\tau}]/(U_1 \,|\, U_2)\\
rel(\tilde{w}:\tilde{\sigma_1} \,|\, \tilde{w}:\tilde{\sigma_2}).
\end{array}$$

Without loss of generality, we can assume $(\nu\tilde{w})\,(P_{01} \,|\, P_{02}) \longrightarrow P_1$ (because otherwise, we can get a fair reduction sequence $(\nu\tilde{w})\,(P_{01} \,|\, P_{02}) \longrightarrow P'_1 \longrightarrow P'_2 \longrightarrow \cdots$ with $P_i \preceq^{-\alpha} P'_i$). Case analysis on P_{01}.

– Case where P_{01} is a conditional process **if** b **then** P'_{01} **else** P''_{01}: b must be *true* or *false*. Suppose $b = \mathit{true}$ (The case for $b = \mathit{false}$ is similar). By the assumption of fairness, there exists n such that

$$\begin{array}{l} P_n \preceq^{-\alpha} (\nu\tilde{w})\,(\nu\tilde{u})\,(P_{01} \,|\, R) \\ (\nu\tilde{w})\,(\nu\tilde{u})\,(P'_{01} \,|\, R) \preceq P_{n+1} \\ P_{02} \longrightarrow^* (\nu\tilde{u})\,R \end{array}$$

By Theorem 3.1, we have

$$\begin{array}{l} \Delta'_2, \tilde{w} : \tilde{\sigma'_2}, x : [\tilde{\tau}]/U'_2 \vdash (\nu\tilde{u})\,R \\ (\Delta_2, \tilde{w} : \tilde{\sigma_2}, x : [\tilde{\tau}]/U_2) \longrightarrow^* (\Delta'_2, \tilde{w} : \tilde{\sigma'_2}, x : [\tilde{\tau}]/U'_2). \end{array}$$

By rule (T-If), we also have

$$\begin{array}{l} \Delta'_1, \tilde{w} : \tilde{\sigma'_1}, x : [\tilde{\tau}]/U'_1 \vdash P'_{01} \\ (\Delta_1, \tilde{w} : \tilde{\sigma_1}, x : [\tilde{\tau}]/U_1) \leq \boxed{1}(\Delta'_1, \tilde{w} : \tilde{\sigma'_1}, x : [\tilde{\tau}]/U'_1) \end{array}$$

So, we have

$$\Delta'_1 \,|\, \Delta'_2, x : [\tilde{\tau}]/(U'_1 \,|\, U'_2) \vdash P_{n+1}.$$

By the condition

$$(\Delta_1, \tilde{w} : \tilde{\sigma_1}, x : [\tilde{\tau}]/U_1) \leq \boxed{1}(\Delta'_1, \tilde{w} : \tilde{\sigma'_1}, x : [\tilde{\tau}]/U'_1),$$

$ob_O(t, U_1)$ and Lemma 3.4, it must be the case that either $ob_O(t, \boxed{1}U'_1)$ holds or $cap_I(t_c, U_1)$ holds for some t_c with $t_c < t$. In the former case, it must be the case that $ob_O(t-1, U'_1)$. So, we can obtain the required result by applying property A of the induction hypothesis to the fair reduction sequence $P_{n+1} \longrightarrow P_{n+2} \longrightarrow \cdots$. In the latter case, it must be the case that $ob_O(t_c, U_1 \,|\, U_2)$. Applying property A of the induction hypothesis to the fair reduction sequence $P_0 \longrightarrow P_1 \longrightarrow \cdots$, we obtain the required result.

– Case where P_{01} is an output or input process: If $P_{01} = \overline{x}\langle\tilde{v}\rangle^a.\,R$, then the result follows immediately. Otherwise, suppose $P_{01} = \overline{y}\langle\tilde{v}\rangle^a.\,R$ with $y \neq x$. (The case where P_{01} is an input process is similar.) Then, it must be the case that

$$\begin{array}{l} (\Delta_1, \tilde{w} : \tilde{\sigma_1})(y) = [\tilde{\tau'}]/U_y \\ U_y \cong O^{t_{oy}}_{t_{cy}}.V_y \,|\, V'_y \\ t_{cy} < t \end{array}$$

Therefore, by property B of the induction hypothesis, there exists n such that

$$\begin{array}{l} P_n \preceq^{-\alpha} (\nu\tilde{w})\,(\nu\tilde{u})\,(\overline{y}\langle\tilde{v}\rangle^a.\,R \,|\, y(\tilde{z})^{a'}.\,R' \,|\, R'')(= P'_n) \\ (\nu\tilde{w})\,(\nu\tilde{u})\,(R \,|\, [\tilde{v}/\tilde{z}]R' \,|\, R'') \preceq P_{n+1} \\ P_{02} \longrightarrow^* (\nu\tilde{u})\,(y(\tilde{z})^{a'}.\,R' \,|\, R'') \end{array}$$

By Theorem 3.1, we have

$$\begin{array}{l} \Delta'_2, \tilde{w} : \tilde{\sigma'_2}, x : [\tilde{\tau}]/U'_2 \vdash (\nu\tilde{u})\,(y(\tilde{z})^{a'}.\,R' \,|\, R'') \\ (\Delta_2, \tilde{w} : \tilde{\sigma_2}, x : [\tilde{\tau}]/U_2) \longrightarrow^* (\Delta'_2, \tilde{w} : \tilde{\sigma'_2}, x : [\tilde{\tau}]/U'_2) \end{array}$$

for some $\Delta_2', \tilde{\sigma_2'}$, and U_2'. By the former condition and the typing rules, it must be the case that

$$\begin{array}{l}\Delta_3, x : [\tilde{\tau}]/U_3, \tilde{u} : \tilde{\sigma_3} \vdash y(\tilde{z})^{a'}.\, R' \\ \Delta_4, x : [\tilde{\tau}]/U_4, \tilde{u} : \tilde{\sigma_4} \vdash R'' \\ (\Delta_2', \tilde{w} : \tilde{\sigma_2'}, x : [\tilde{\tau}]/U_2') \leq (\Delta_3 \,|\, \Delta_4), x : [\tilde{\tau}]/(U_3 \,|\, U_4)\end{array}$$

So, we obtain

$$(\Delta_1, \tilde{w} : \tilde{\sigma_1}) \,|\, (\Delta_3, \tilde{u} : \tilde{\sigma_3}), x : [\tilde{\tau}]/(U_1 \,|\, U_3) \vdash P_{01} \,|\, y(\tilde{z})^{a'}.\, R'.$$

By Lemma 3.2, we have

$$\begin{array}{l}\Theta, x : [\tilde{\tau}]/U_5 \vdash R \,|\, [\tilde{z} \mapsto \tilde{v}]R' \\ U_1 \,|\, U_3 \leq \boxed{1} U_5\end{array}$$

for some Θ and U_5. By the latter condition, $ob_O(t, U_1)$, and by Lemma 3.4, it must be the case that either $ob_O(t, \boxed{1} U_5)$ holds or $cap_I(t_c, U_1 \,|\, U_3)$ holds for some t_c with $t_c < t$. In the former case, we have $ob_O(t-1, U_5)$. By Lemma 3.5, we have

$$(\Theta \backslash \{\tilde{u}, \tilde{w}\}) \,|\, (\Delta_4 \backslash \tilde{w}), x : [\tilde{\tau}]/(U_5 \,|\, U_4) \vdash P_{n+1}$$

and $rel((\Theta \backslash \{\tilde{u}, \tilde{w}\}) \,|\, (\Delta_4 \backslash \tilde{w}), x : [\tilde{\tau}]/(U_5 \,|\, U_4))$. Hence, we can obtain the required result by applying property A of the induction hypothesis to the fair reduction sequence $P_{n+1} \longrightarrow P_{n+2} \longrightarrow \cdots$.

In the latter case, we have

$$\begin{array}{l}(\Delta_1 \,|\, \Delta_3 \,|\, \Delta_4) \backslash \{\tilde{w}\}, x : [\tilde{\tau}]/(U_1 \,|\, U_3 \,|\, U_4) \vdash P_n' \\ ob_O(t_c, U_1 \,|\, U_3 \,|\, U_4).\end{array}$$

So, applying property A of the induction hypothesis to the fair reduction sequence $P_n' \longrightarrow P_{n+1} \longrightarrow \cdots$, we obtain the required result.

B. We show only the first case. The second case is similar. Suppose there exists no such n. Then, it must be the case that $P_i \preceq^{-\alpha} (\nu \tilde{u}_i)\, (\overline{x}\langle\tilde{v}\rangle^a.\, Q \,|\, R_i)$ and $R \longrightarrow (\nu \tilde{u}_1)\, R_1 \longrightarrow (\nu \tilde{u}_2)\, R_2 \longrightarrow \cdots$ is a fair reduction sequence. We show that there exist infinitely many i such that $R_i \preceq^{-\alpha} (\nu \tilde{w}_i)\, (x(\tilde{y})^{a'}.\, R_i' \,|\, R_i'')$, 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 the subject reduction theorem (Theorem 3.1), there must exist Γ_2' such that $\Gamma_2' \vdash (\nu \tilde{u_j})\, R_j$ and $\Gamma_2 \longrightarrow^* \Gamma_2'$. Let $\Gamma' = \Gamma_1 \,|\, \Gamma_2'$. Since $rel(\Gamma)$ and $\Gamma \leq\longrightarrow^* \Gamma'$ hold, $rel(\Gamma')$ follows by Lemma 3.5. By the assumption $cap_{\overline{x}}(t, \Gamma_1)$, we have $ob_x(t, \Gamma')$. We also have that $\Gamma' \vdash (\nu \tilde{u_j})\, (\overline{x}\langle\tilde{v}\rangle^a.\, Q \,|\, R_j)$. Therefore, by property A, there exists $i \geq j$ such that $R_i \preceq^{-\alpha} (\nu \tilde{w}_i)\, (x(\tilde{y})^{a'}.\, R_i' \,|\, R_i'')$. Thus, we have shown that there exist infinitely many i such that $R_i \preceq^{-\alpha} (\nu \tilde{w}_i)\, (x(\tilde{y})^{a'}.\, R_i' \,|\, R_i'')$.

■

Corollary 3.6 (lock-freedom). If $\Gamma \vdash P$ and $rel(\Gamma)$, then P is lock-free.

Proof. Suppose $\Gamma \vdash P$, $rel(\Gamma)$, and $P \longrightarrow^* P_i \preceq^{-\alpha} (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle^{\mathbf{c}}.\,Q \mid R)$. By Theorem 3.1, we have $\Gamma', \tilde{w}:\tilde{\tau} \vdash \overline{x}\langle\tilde{v}\rangle^{\mathbf{c}}.\,Q \mid R$ and $rel(\Gamma', \tilde{w}:\tilde{\tau})$ for some Γ' and $\tilde{\tau}$. Moreover, by the typing rules, it must be the case that $\Delta_1 \vdash \overline{x}\langle\tilde{v}\rangle^{\mathbf{c}}.\,Q$, $\Delta_2 \vdash R$, $cap_{\overline{x}}(t, \Delta_1)$, and $(\Gamma', \tilde{w}:\tilde{\tau}) \leq \Delta_1 \mid \Delta_2$ for some Δ_1, Δ_2, and t $(< \infty)$. By Lemma 3.5, we have $rel(\Delta_1 \mid \Delta_2)$. So, the required result follows from property B of Theorem 3.3. The case for input is similar. ■

3.8. Examples

In this subsection, we give examples to show how our type system can be used to reason about lock-freedom.

Example 3.6. A concurrent object can be modeled by multiple processes, each of which handles each method of the object [27, 18, 14]. For example, a point object is expressed by the following process:

$$
\begin{array}{l}
(\nu s:[int, int]/(O^0_\infty.\mathbf{0} \mid *I^\infty_0.O^0_\infty.\mathbf{0}))\\
\quad (\overline{s}\langle 0, 0\rangle\\
\quad\quad \mid *move(dx:int, dy:int, r:[\,]/O^1_\infty.\mathbf{0}).\,s(x, y).\,(\overline{s}\langle x + dx, y + dy\rangle \mid \overline{r}\langle\,\rangle)\\
\quad\quad \mid *read(r:[int, int]/O^1_\infty.\mathbf{0}).\,s(x, y).\,(\overline{s}\langle x, y\rangle \mid \overline{r}\langle x, y\rangle))
\end{array}
$$

Here, bound variables are annotated with types for clarity. The channel s is used to store the current location. $\overline{s}\langle 0, 0\rangle$ means that the current location is $(0, 0)$. The process above waits to receive request messages on channels *move* and *read*. For example, when a request $\overline{move}\langle dx, dy, r\rangle$ arrives, it reads the current location, updates it, and sends an acknowledgment on r.

The type of channel s implies that an output is performed immediately after s is created, and that whenever an input is performed, it succeeds eventually and after that, an output is performed immediately. The type of (two occurrences of) channel r implies that a reply is sent in time 1. So, if the process above is well-typed, a client process (like $(\nu r)\,(\overline{read}\langle r\rangle \mid r(x, y)^{\mathbf{c}}.\,\cdots)$) can receive a reply in time 1.

Let us briefly check that the process above is indeed well-typed. First, consider the sub-process $*read(r).\,s(x, y).\,(\overline{s}\langle x, y\rangle \mid \overline{r}\langle x, y\rangle)$. By using rules (T-Zero), (T-Out) and (T-Par), we obtain

$$s:[int, int]/O^0_\infty.\mathbf{0}, r:[int, int]/O^0_\infty.\mathbf{0} \vdash \overline{s}\langle x, y\rangle \mid \overline{r}\langle x, y\rangle.$$

From this, we obtain

$$s:[int, int]/I^\infty_0.O^0_\infty.\mathbf{0}, r:[int, int]/O^1_\infty.\mathbf{0} \vdash s(x, y).\,(\overline{s}\langle x, y\rangle \mid \overline{r}\langle x, y\rangle)$$

by using rule (T-In). By using rule (T-In) and (T-Rep), we derive

$$read:[[int, int]/O^1_\infty.\mathbf{0}]/*I^0_\infty.\mathbf{0}, s:[int, int]/*I^\infty_0.O^0_\infty.\mathbf{0} \vdash *read(r).\,s(x, y).\,\cdots.$$

The other part can be type-checked similarly, and the whole process is type-checked under the type environment:

$$move:[int, int, [\,]/O^1_\infty.\mathbf{0}]/*I^0_\infty.\mathbf{0}, read:[[int, int]/O^1_\infty.\mathbf{0}]/*I^0_\infty.\mathbf{0}.$$

EXAMPLE 3.7. Let *Point* be the point process in the example above. Using it, we can implement a circle process, whose center is the point above and radius is 3, as follows:

$$
\begin{array}{l}
(\nu move)\,(\nu read)\,(\nu s : [int]/(O^0_\infty.\mathbf{0} \,|\, {*}I^\infty_0.O^0_\infty.\mathbf{0}))\\
\quad (\overline{s}\langle 3\rangle \,|\, Point\\
\quad\ \,|\, {*}movec(dx : int, dy : int, r : [\,]/O^2_\infty.\mathbf{0}).\,\overline{move}\langle dx, dy, r\rangle\\
\quad\ \,|\, {*}center(r : [int, int]/O^2_\infty.\mathbf{0}).\,\overline{read}\langle r\rangle\\
\quad\ \,|\, {*}radius(r : [int]/O^1_\infty.\mathbf{0}).\,s(z).\,(\overline{s}\langle z\rangle \,|\, \overline{r}\langle z\rangle))
\end{array}
$$

This object accepts three kinds of requests *movec*, *center*, and *radius*. When a *movec* or *center* request arrives, the object forwards the request to the pointer. When a *radius* request arrives, the object reads the radius from s and returns it.

Note that the type of channel r on the third line is $[\,]/O^2_\infty.\mathbf{0}$. The time limit of execution of an output is 2, because it takes one step for the forwarded message $\overline{move}\langle dx, dy, r\rangle$ to be received by the point, and it takes another step for the object to send a reply. The process above is well typed under the following type environment:

$$
\begin{array}{l}
movec : [int, int, [\,]/O^2_\infty.\mathbf{0}]/{*}I^0_\infty.\mathbf{0}, center : [[int, int]/O^2_\infty.\mathbf{0}]/{*}I^0_\infty.\mathbf{0},\\
radius : [[int]/O^1_\infty.\mathbf{0}]/{*}I^0_\infty.\mathbf{0}
\end{array}
$$

So, we see that a client can receive a reply eventually.

Suppose that the subprocess ${*}movec(dx, dy, r).\,\overline{move}\langle dx, dy, r\rangle$ is replaced by ${*}movec(dx, dy, r).\,\overline{movec}\langle dx, dy, r\rangle$ by mistake. Then, only the type of r is not $[\,]/O^2_\infty.\mathbf{0}$ but $[\,]/O^\infty_\infty.\mathbf{0}$. So, we know that a reply may not be returned in this case.

EXAMPLE 3.8. Behavior of a dining philosopher can be expressed by the following process.

$$
P \stackrel{\triangle}{=} {*}phil(left, right).\,left(\,)^{\mathbf{c}}.\,right(\,)^{\mathbf{c}}.\,food(\,).\,(\overline{left}\langle\,\rangle \,|\, \overline{right}\langle\,\rangle \,|\, \overline{phil}\langle left, right\rangle)
$$

A philosopher is parameterized with two channels $left, right$ representing forks. It first acquires the forks (by $left(\,)^{\mathbf{c}}.\,right(\,)^{\mathbf{c}}.\,\cdots$), then eats *food*(by $food(\,).\,\cdots$), releases forks, and repeats the same behavior. The inputs on $left$ and $right$ are annotated with $\mathbf{c}$, because we want a philosopher to acquire forks eventually.

If there are two philosophers sharing forks f_1 and f_2, we can think of the following two configurations:

$$
\begin{array}{l}
Q_1 \stackrel{\triangle}{=} (\nu food)\,(\nu phil)\,(P \,|\, {*}\overline{food}\langle\,\rangle \,|\, \overline{phil}\langle f_1, f_2\rangle \,|\, \overline{phil}\langle f_2, f_1\rangle \,|\, \overline{f_1}\langle\,\rangle \,|\, \overline{f_2}\langle\,\rangle)\\
Q_2 \stackrel{\triangle}{=} (\nu food)\,(\nu phil)\,(P \,|\, {*}\overline{food}\langle\,\rangle \,|\, \overline{phil}\langle f_1, f_2\rangle \,|\, \overline{phil}\langle f_1, f_2\rangle \,|\, \overline{f_1}\langle\,\rangle \,|\, \overline{f_2}\langle\,\rangle)
\end{array}
$$

In Q_1, the two philosophers try to acquire forks f_1 and f_2 in the reverse order, while in Q_2, the philosophers try to acquire f_1 and f_2 in the same order.

To see which configuration may suffer from a lock, we can check typing. First, the process P can be typed as:

$$
phil : [[\,]/{*}I^\infty_3.O^3_\infty.\mathbf{0}, [\,]/{*}I^\infty_1.O^1_\infty.\mathbf{0}]/({*}I^0_\infty.\mathbf{0} \,|\, {*}O^\infty_0.\mathbf{0}), food : [\,]/{*}I^\infty_0.\mathbf{0} \vdash P.
$$

The part $left(\,)^{\mathbf{c}}.\,right(\,)^{\mathbf{c}}.\,\cdots$ is type-checked since the time limits of success of inputs on $left$ and $right$ are respectively 3 and 1. The time limit of execution of an output on $right$ is 1 since the process only waits on *food* before releasing the $right$ fork after acquiring it. On the other hand, the time limit of execution of an output on $left$ is 3 because the process waits on $right$ and *food* before releasing the $right$ fork after acquiring it.

Using the typing of P, Q_1 and Q_2 are typed as follows:

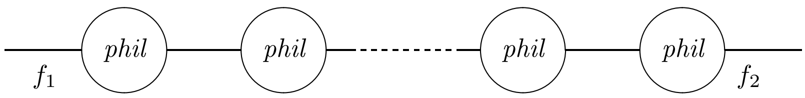
$$f_1 : [\,]/(*I_3^\infty.O_\infty^3.\mathbf{0} \mid *I_1^\infty.O_\infty^1.\mathbf{0} \mid O_\infty^0.\mathbf{0}), f_2 : [\,]/(*I_1^\infty.O_\infty^1.\mathbf{0} \mid *I_3^\infty.O_\infty^3.\mathbf{0} \mid O_\infty^0.\mathbf{0}) \vdash Q_1$$
$$f_1 : [\,]/(*I_3^\infty.O_\infty^3.\mathbf{0} \mid *I_3^\infty.O_\infty^3.\mathbf{0} \mid O_\infty^0.\mathbf{0}), f_2 : [\,]/(*I_1^\infty.O_\infty^1.\mathbf{0} \mid *I_1^\infty.O_\infty^1.\mathbf{0} \mid O_\infty^0.\mathbf{0}) \vdash Q_1$$

The usage $*I_3^\infty.O_\infty^3.\mathbf{0} \mid *I_1^\infty.O_\infty^1.\mathbf{0} \mid O_\infty^0.\mathbf{0}$ of channel f_1 in Q_1 is not reliable, because the usage is reduced to $*I_3^\infty.O_\infty^3.\mathbf{0} \mid *I_1^\infty.O_\infty^1.\mathbf{0} \mid O_\infty^3.\mathbf{0}$. Note that the part $*I_1^\infty.O_\infty^1.\mathbf{0}$ means that the fork is expected to be acquired in time 1, but the fork is only guaranteed to be released in time 3 (by the part $O_\infty^3.\mathbf{0}$). Therefore, we know that Q_1 may suffer from a lock. On the other hand, Q_2 does not suffer from a lock because the usages of f_1 and f_2 are both reliable.

The reasoning above can be extended to $2N$ philosophers easily (where N is a natural number). The following process creates a configuration consisting of $2N$ philosophers.

$$
\begin{array}{l}
(\nu food)\,(\nu f_1)\,(\nu f_2)\,(\overline{config}\langle N, f_1, f_2\rangle \mid \overline{phil}\langle f_1, f_2\rangle \mid \overline{f_1}\langle\,\rangle \mid \overline{f_2}\langle\,\rangle \mid *\overline{food}\langle\,\rangle \mid P) \mid \\
*config(n, left, right). \\
\quad (\textbf{if } n = 0 \textbf{ then } \overline{phil}\langle left, right\rangle \\
\quad\ \textbf{else } (\nu f_3)\,(\nu f_4)\,(\overline{phil}\langle left, f_3\rangle \mid \overline{phil}\langle f_4, f_3\rangle \mid \overline{f_3}\langle\,\rangle \mid \overline{f_4}\langle\,\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \overline{config}\langle n-1, f_4, right\rangle))
\end{array}
$$

Here, $\overline{config}\langle N, f_1, f_2\rangle$ creates $2N-1$ philosophers and connects them as follows:



In the process above, $2n$th philosopher acquire the lefthand fork and the righthand fork in this order, while $2n+1$th philosopher acquire the forks in the reverse order. The process above is well typed under the type environment:

$$
\begin{array}{l}
phil : [[\,]/*I_3^\infty.O_\infty^3.\mathbf{0}, [\,]/*I_1^\infty.O_\infty^1.\mathbf{0}]/(*I_\infty^0.\mathbf{0} \mid *O_0^\infty.\mathbf{0}), \\
config : [int, [\,]/*I_3^\infty.O_\infty^3.\mathbf{0}, [\,]/*I_1^\infty.O_\infty^1.\mathbf{0}]/(*I_\infty^0.\mathbf{0} \mid *O_0^\infty.\mathbf{0})
\end{array}
$$

So, we know that the process is lock-free for any natural number N.

A solution to the problem of $2N+1$ dining philosophers is to let one philosopher acquire the lefthand fork and the righthand fork in this order and let all the others acquire the forks in the reverse order. To describe this solution, we need dependent types discussed in Section 5.

4. TIME-BOUNDED PROCESSES

In this section, we 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. Parallel reduction allows several independent communications to be performed in one step.

4.1. Time-Boundedness

We first define the time-boundedness of a process. We replace attributes of input/output processes with time limits within which the input/output actions should succeed. The new syntax of processes is given by:

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid A \mid (P \,|\, Q) \mid (\nu x)\, P \\ A & ::= & \overline{x}\langle v_1, \ldots, v_n \rangle^t .\, P \mid x(y_1, \ldots, y_n)^t .\, P \mid \textbf{if } v \textbf{ then } P \textbf{ else } Q \mid {*A} \end{array}$$

The annotation t of $\overline{x}\langle \tilde{v} \rangle^t .\, P$ specifies that this output process can find a communication partner and start a communication within t *parallel* reduction steps (defined below) after it is executed. (It takes another step for the output process to complete the communication.)

Assuming unlimited parallelism in reductions of concurrent processes, we count the number of parallel reduction steps performed. Note that communications on different channels can occur in parallel in this parallel reduction model. To model such parallel reduction, we introduce a parallel reduction relation $P \Longrightarrow Q$: $P \Longrightarrow Q$ means that P is reduced to Q by reducing every conditional expression and performing one communication on every channel whenever possible. We assume here that at most one communication can occur on each channel and that the two processes that communicate each other are chosen randomly. So, it takes two steps to reduce $*x(\,).\,\mathbf{0} \,|\, \overline{x}\langle\,\rangle \,|\, \overline{x}\langle\,\rangle$ to $*x(\,).\,\mathbf{0}$. It is possible to change reduction rules and the type system in order to allow as many communications as possible to occur simultaneously on each channel and/or to reflect a certain scheduling (such as priority scheduling and the FIFO scheduling).

Remark 4.8. The reason why we do not consider the number of *sequential* reductions ($\longrightarrow$ in Section 2) is that it is not preserved by parallel composition with independent processes (not sharing any channels). Consider the process $x(\,).\,\mathbf{0} \,|\, \overline{x}\langle\,\rangle$. The input on x succeeds immediately, but if the process is executed in parallel with a diverging process $(\nu y)\,(\overline{y}\langle\,\rangle \,|\, {*y(\,).\,\overline{y}\langle\,\rangle})$, we can no longer bound the number of sequential reduction steps required for the success of the input.

To define the relation $P \Longrightarrow Q$, we use an auxiliary relation $P \stackrel{S}{\Longrightarrow} Q$ where S is a subset of $\{\mathbf{com}_x, \overline{x}, x \mid x \in \mathbf{Var}\}$. Intuitively, $P \stackrel{S}{\Longrightarrow} Q$ means:

- a communication is performed on every closed channel whenever possible,
- if $\mathbf{com}_x \in S$, then a communication is performed on the free channel x, and
- if $\overline{x} \in S$ ($x \in S$, resp.), there is an output (input, resp.) process on x that is not reduced in this step.

Definition 4.25 (parallel process reduction). $\stackrel{S}{\Longrightarrow}$ (where S is a subset of $\{\mathbf{com}_x, \overline{x}, x \mid x \in \mathbf{Var}\}$) and $\Longrightarrow$ are the least relations closed under the rules given below.

$$\overline{x}\langle v_1, \ldots, v_n \rangle^t .\, P \,|\, x(z_1, \ldots, z_n)^{t'} .\, Q \stackrel{\{\mathbf{com}_x\}}{\Longrightarrow} P \,|\, [z_1 \mapsto v_1, \ldots, z_n \mapsto v_n]Q \qquad \text{(PR-Com)}$$

$$\mathbf{0} \overset{\emptyset}{\Longrightarrow} \mathbf{0} \tag{PR-Zero}$$

$$\overline{x}\langle \tilde{v}\rangle^t.\,P \overset{\{\overline{x}\}}{\Longrightarrow} \overline{x}\langle \tilde{v}\rangle^{t-1}.\,P \tag{PR-WaitO}$$

$$x(\tilde{v})^t.\,P \overset{\{x\}}{\Longrightarrow} x(\tilde{v})^{t-1}.\,P \tag{PR-WaitI}$$

$$\frac{P \overset{S_1}{\Longrightarrow} P' \qquad Q \overset{S_2}{\Longrightarrow} Q' \qquad S_1 \cap S_2 \cap \{\mathbf{com}_x \mid x \in \mathbf{Var}\} = \emptyset}{P \,|\, Q \overset{S_1 \cup S_2}{\Longrightarrow} P' \,|\, Q'} \tag{PR-Par}$$

$$\frac{P \overset{S}{\Longrightarrow} Q \qquad \{\overline{x}, x\} \subseteq S \Rightarrow \mathbf{com}_x \in S}{(\nu x)\, P \overset{S\backslash\{x\}}{\Longrightarrow} (\nu x)\, Q} \tag{PR-New}$$

$$\frac{P \overset{S}{\Longrightarrow} Q \qquad \forall x.(\mathbf{com}_x \notin S)}{*P \overset{S}{\Longrightarrow} *Q} \tag{PR-Rep}$$

$$\mathbf{if}\ \mathit{true}\ \mathbf{then}\ P\ \mathbf{else}\ Q \overset{\emptyset}{\Longrightarrow} P \tag{PR-IfT}$$

$$\mathbf{if}\ \mathit{false}\ \mathbf{then}\ P\ \mathbf{else}\ Q \overset{\emptyset}{\Longrightarrow} Q \tag{PR-IfF}$$

$$\frac{P \preceq P' \qquad P' \overset{S}{\Longrightarrow} Q' \qquad Q' \preceq Q}{P \overset{S}{\Longrightarrow} Q} \tag{PR-SPCong}$$

$$\frac{P \overset{S}{\Longrightarrow} Q \qquad \forall x \in \mathbf{Var}.(\{\overline{x}, x\} \subseteq S \Rightarrow \mathbf{com}_x \in S)}{P \Longrightarrow Q} \tag{PR-Close}$$

Here, $\infty - 1 = \infty$, and $0 - 1$ is undefined.

Main differences between parallel reductions and sequential reductions are that in a parallel reduction step,

- Multiple communications can be performed in parallel. (Note the difference between rule (PR-Par) and rule (R-Par).)
- Time limits for processes that do not participate in any communication are decremented by one.

Similar features are found in operational semantics of timed process calculi [2]. Basically, a parallel reduction step $P \Longrightarrow Q$ corresponds to several sequential reduction steps $P \longrightarrow^* Q$, except that time annotations may change in $P \Longrightarrow Q$.

Rule (PR-WaitO) ((PR-WaitI), resp.) says that if an output (input, resp.) process is not reduced, its time limit is decremented. The rightmost premises of rules (PR-Par) and (PR-Rep) ensure that only one communication can be performed on each channel: If we remove this condition, multiple communications will be performed on the same channel in each step. The right premises of rules

(PR-New) and (PR-Close) ensure that a communication is performed on each channel whenever both an input process and an output process are ready.

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

Definition 4.26 (time-boundedness). A process P is *time-bounded* if the following conditions hold whenever $P \Longrightarrow^* P'$.

1.If $P' \preceq (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle^0.\,Q \,|\, R)$, then $R \overset{\mathbf{com}_x}{\not\longrightarrow}$ and $R \preceq (\nu\tilde{u})\,(x(\tilde{y})^{t'}.\,R_1 \,|\, R_2)$ with $x \notin \{\tilde{u}\}$ for some $t', \tilde{u}, R_1, R_2$.
2.If $P' \preceq (\nu\tilde{w})\,(x(\tilde{y})^0.\,Q \,|\, R)$, then $R \overset{\mathbf{com}_x}{\not\longrightarrow}$ and $R \preceq (\nu\tilde{u})\,(\overline{x}\langle\tilde{v}\rangle^{t'}.\,R_1 \,|\, R_2)$ with $x \notin \{\tilde{u}\}$ for some $t', \tilde{u}, \tilde{v}, R_1, R_2$.

In the first condition, $R \overset{\mathbf{com}_x}{\not\longrightarrow}$ ensures that there is no other output process, so that $\overline{x}\langle\tilde{v}\rangle^0.\,Q$ can always communicate with $x(\tilde{y})^{t'}.\,R_1$.

4.2. Modification to the Type System

Now we modify the type system in Section 3 to guarantee the time-boundedness. We just need to refine the reliability condition (Definition 3.16) to estimate the channel-wise behavior more correctly. As stated in Remark 3.4, a problem of Definition 3.16 is that it does not take race conditions into account. For example, $O^0_\infty.\mathbf{0} \,|\, I^\infty_0.O^0_\infty.\mathbf{0} \,|\, I^\infty_0.O^0_\infty.\mathbf{0}$ is reliable according to Definition 3.16, but only one input is guaranteed to succeed immediately: The other input must wait until an output action is executed again. The correct usage should be $O^0_\infty.\mathbf{0} \,|\, I^\infty_1.O^0_\infty.\mathbf{0} \,|\, I^\infty_1.O^0_\infty.\mathbf{0}$.

We redefine usage reduction to take race conditions into account. For example, $O^0_\infty.\mathbf{0} \,|\, I^\infty_1.O^0_\infty.\mathbf{0} \,|\, I^\infty_1.O^0_\infty.\mathbf{0}$ is reduced to $O^0_\infty.\mathbf{0} \,|\, I^\infty_0.O^0_\infty.\mathbf{0}$. Note that in this case the time limit of success of an input has been reduced by 1. The resultant usage is further reduced to $O^0_\infty.\mathbf{0}$. To define a new usage reduction relation $U \Longrightarrow U'$, we introduce an auxiliary relation $\overset{S}{\Longrightarrow}$ where $\{\mathbf{act}_O, \mathbf{act}_I, O, I\} \subseteq S$. When $\mathbf{act}_O \in S$ ($\mathbf{act}_I \in S$, resp.), $U \overset{S}{\Longrightarrow} V$ means that an output (input, resp.) usage in U is reduced. $O \in S$ ($I \in S$, resp.) indicates that an output (input, resp.) action is ready but does not participate in this reduction step (either because no input action is ready or because another output usage is chosen for communication).

Definition 4.27 (timed usage reduction). Binary relations $\overset{S}{\Longrightarrow}$ and $\Longrightarrow$ (where $S \subseteq \{\mathbf{act}_I, \mathbf{act}_O, I, O\}$) on usages are the least relations closed under the following rules:

$$\alpha^{t_o}_{t_c}.U \overset{\{\mathbf{act}_\alpha\}}{\Longrightarrow} U \qquad \text{(TU-Act)}$$

$$\mathbf{0} \overset{\emptyset}{\Longrightarrow} \mathbf{0} \qquad \text{(TU-Zero)}$$

$$\frac{0 < t_c}{\alpha^{t_o}_{t_c}.U \overset{\{\alpha\}}{\Longrightarrow} \alpha^{0}_{t_c-1}.U} \qquad \text{(TU-Wait)}$$

$$\frac{0 < t_o}{\alpha^{t_o}_{t_c}.U \overset{\emptyset}{\Longrightarrow} \alpha^{t_o-1}_{t_c}.U} \qquad \text{(TU-Skip)}$$

$$\frac{U_1 \stackrel{S_1}{\Longrightarrow} U_1' \qquad U_2 \stackrel{S_2}{\Longrightarrow} U_2' \qquad S_1 \cap S_2 \cap \{\mathbf{act}_I, \mathbf{act}_O\} = \emptyset}{U_1 \,|\, U_2 \stackrel{S_1 \cup S_2}{\Longrightarrow} U_1' \,|\, U_2'} \quad \text{(TU-Par)}$$

$$\frac{U \stackrel{S}{\Longrightarrow} U' \qquad S \cap \{\mathbf{act}_I, \mathbf{act}_O\} = \emptyset}{*U \stackrel{S}{\Longrightarrow} *U'} \quad \text{(TU-Rep)}$$

$$\frac{U \cong U' \qquad U' \stackrel{S}{\Longrightarrow} V' \qquad V' \cong V}{U \stackrel{S}{\Longrightarrow} V} \quad \text{(TU-Cong)}$$

$$\frac{\begin{array}{c} U \stackrel{S}{\Longrightarrow} V \qquad \{I, O\} \subseteq S \Rightarrow \{\mathbf{act}_I, \mathbf{act}_O\} \subseteq S \\ \mathbf{act}_\alpha \in S \Rightarrow \mathbf{act}_{\overline{\alpha}} \in S \end{array}}{U \Longrightarrow V} \quad \text{(TU-Red)}$$

Rules (TU-ComI) and (TU-ComO) respectively model the situations where input and output actions succeed. On the other hand, rule (TU-Wait) models the situation where the (input or output) action α has been executed but has not succeeded. In this case, the time limit of success of the action is decremented by 1. Rule (TU-Skip) is for the case where the action α has not been executed yet: in this case, the time limit of execution of the action is reduced by 1. The rightmost premises of rules (TU-Par) and (TU-Rep) ensure that only one pair of an input usage and an output usage can be reduced. Rule (TU-Red) 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.

Remark 4.9. The above usage reduction relation allows only one pair of I and O to be reduced in one step. This is because we defined parallel reduction of processes so that only one communication can occur on each channel. If we allow multiple communications to be simultaneously performed on the same channel, we should allow multiple pairs of I and O to be reduced in one step, by removing the side conditions of (TU-Par) and (TU-Rep), replacing labels with multisets, and replacing the third condition of (TU-Red) with the condition that S must contain the same number of $\mathbf{act}_I$ and $\mathbf{act}_O$.

Using the above parallel usage reduction, we can strengthen the reliability condition as defined below. The 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 4.28 (reliability (refined)). A usage U is reliable, written $rel_T(U)$, if $ob_{\overline{\alpha}}(0, U_2)$ and $U_2 \not\longrightarrow$ hold whenever $U \Longrightarrow^* \cong \alpha_0^{t_o}.U_1 \,|\, U_2$.

Predicate rel_T is extended to those on types and type environments in a manner similar to rel.

The new set of typing rules is obtained by replacing rules (T-Out), (T-In), and (T-New) in Figure 1 with the following rules.

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U \vdash P \qquad t_c \leq t}{x : [\tau_1, \ldots, \tau_n]/O^0_{t_c}.U \mid \boxed{t_c + 1}(v_1 : \tau_1 \mid \cdots \mid v_n : \tau_n \mid \Gamma) \vdash \overline{x}\langle v_1, \ldots, v_n\rangle^t.\, P} \quad \text{(T-BOut)}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash P \qquad t_c \leq t}{\boxed{t_c + 1}\Gamma, x : [\tau_1, \ldots, \tau_n]/I^0_{t_c}.U \vdash x(y_1, \ldots, y_n)^t.\, P} \quad \text{(T-BIn)}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U \vdash P \qquad \mathit{rel}_T(U)}{\Gamma \vdash (\nu x)\, P} \quad \text{(T-BNew)}$$

Rules (T-BOut) and (T-BIn) are the same as (T-Out) and (T-In) except that the time limit t given by a programmer is compared with the actual time limit t_c guaranteed by the type system. We write $\Gamma \vdash_T P$ if $\Gamma \vdash P$ is derivable using the new typing rules.

4.3. Type Soundness

We can prove the following time-boundedness theorem.

Theorem 4.1. *If $\Gamma \vdash_T P$ and $\mathit{rel}_T(\Gamma)$, then P is time-bounded.*

In particular, every closed well-typed process is time-bounded. The above theorem follows from the following properties:

- The well-typedness of a process is preserved by (parallel) reductions.
- Any well-typed process does not immediately suffer from a time-out. (By "time-out," we mean that the time limit of success of some action has reached, but the action is not enabled.)

As in the case for sequential reductions (recall Theorem 3.1), the type environment of a process changes during parallel reductions. To express the change of a type environment, we first define a parallel reduction relation $\Gamma \stackrel{S}{\Longrightarrow} \Delta$.

Definition 4.29. Let S be a subset of $\{\mathbf{com}_x, x, \overline{x} \mid x \in \mathbf{Var}\}$. $S|_x$ is the set $\{\mathbf{act}_I, \mathbf{act}_O \mid \mathbf{com}_x \in S\} \cup \{I \mid x \in S\} \cup \{O \mid \overline{x} \in S\}$.

Definition 4.30. A ternary relation $\Gamma \stackrel{S}{\Longrightarrow} \Delta$ is defined to hold if the following conditions hold

1.$\mathit{dom}(\Gamma) = \mathit{dom}(\Delta)$.

2.For each x in $\mathit{dom}(\Gamma)$, one of the following conditions hold:

(i)$\Gamma(x) = \Delta(x) = \mathit{bool}$, or

(ii)There exist $\tilde{\tau}$, U, U' such that $\Gamma(x) = [\tilde{\tau}]/U$, $\Delta(x) = [\tilde{\tau}]/U'$, and $U \stackrel{S|_x}{\Longrightarrow} U'$.

We write $\Gamma \Longrightarrow \Delta$ when $\Gamma \stackrel{S}{\Longrightarrow} \Delta$ for some S.

Using the above relation, the first property discussed above is stated as the following theorem.

THEOREM 4.2 (parallel subject reduction). *If* $\Gamma \vdash_T P$, $rel_T(\Gamma)$, *and* $P \stackrel{S}{\Longrightarrow} Q$, *then there exists* Δ *such that* $\Gamma \stackrel{S}{\Longrightarrow} \Delta$ *and* $\Delta \vdash_T Q$.

Proof. See Appendix B.1. ∎

Proof (Proof of Theorem 4.1). Suppose $P \Longrightarrow^* \preceq (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle^0.\,Q \mid R)$. By Theorem 4.2, we have $\Gamma' \vdash_T (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle^0.\,Q \mid R)$ and $rel_T(\Gamma')$ for some Γ'. By the typing rules, it must be the case that

$$\begin{array}{l} \Delta_1, x : [\tilde{\tau}]/O_0^{t_o}.U_1 \vdash_T \overline{x}\langle\tilde{v}\rangle^0.\,Q \\ \Delta_2, x : [\tilde{\tau}]/U_2 \vdash_T R \\ rel_T(O_0^{t_o}.U_1 \mid U_2) \end{array}$$

By the last condition, it must be the case that $U_2 \cong I_{t_c}^0.U_3 \mid U_4$ and $U_2 \not\longrightarrow$. By the first condition and typing rules, it must be the case that $R \preceq (\nu\tilde{u})\,(x(\tilde{y}).\,R_1 \mid R_2)$. The other required condition $R \stackrel{x}{\not\longrightarrow}$ follows from the condition $U_2 \not\longrightarrow$ and the subject reduction theorem (Theorem 3.1).

The case for input is similar. ∎

5. EXTENSIONS

In this section, we give some examples that show limitations of our type system in Section 3 and discuss how to extend the type system to overcome the limitations.

5.1. Dependent Types

Let us consider the following process P, which works as a function server computing the factorial of a natural number:

$$*fact(n, r).\,(\textbf{if } n = 0 \textbf{ then } \overline{r}\langle 1\rangle \textbf{ else } (\nu r')\,(\overline{fact}\langle n-1, r'\rangle \mid r'(m).\,\overline{r}\langle m \times n\rangle))$$

The parallel composition of P and a client process $(\nu y)\,(\overline{fact}\langle 3, y\rangle \mid y(x)^{\mathbf{c}}.\,\mathbf{0})$ cannot be judged to be lock-free in our type system. Suppose that the type of $fact$ is of the form $[\mathbf{Nat}, [\mathbf{Nat}]/O_{t_c}^{t_o}.\mathbf{0}]/U$. Then, $\overline{fact}\langle n-1, r'\rangle$ and $r'(m).\,\overline{r}\langle m \times n\rangle$ are typed as:

$$\begin{array}{r} fact : [\mathbf{Nat}, [\mathbf{Nat}]/O_{t_c}^{t_o}.\mathbf{0}]/I_0^0.\mathbf{0}, r' : [\mathbf{Nat}]/O_{t_c}^{t_o+1}.\mathbf{0}, n : \mathbf{Nat} \;\vdash\; \overline{fact}\langle n-1, r'\rangle \\ r' : [\mathbf{Nat}]/I_{t'_c}^0.\mathbf{0}, r : [\mathbf{Nat}]/O_{t_c}^{t'_c+1}.\mathbf{0} \;\vdash\; r'(m).\,\overline{r}\langle m \times n\rangle \end{array}$$

In order for the usage of r' to be reliable, it must be the case that $t_o + 1 \le t'_c$. Since r must have type $[\mathbf{Nat}]/O_{t_c}^{t_o}.\mathbf{0}$ in the process $\textbf{if } n = 0 \textbf{ then } \overline{r}\langle 1\rangle \textbf{ else } \cdots$, it must be the case that $t'_c + 1 + 1 \le t_o$. From the two conditions, we obtain $t_o + 3 \le t_o$, which implies $t_o = \infty$. So, it is not guaranteed that P returns a result eventually.

The problem of the example above is that the time required for r to be used for output depends on the other argument n. To express such dependency, we can use dependent types [24]. In general, a dependent type $\Sigma x : \tau_1.\tau_2$ describes a pair $\langle v_1, v_2\rangle$ such that v_1 has type τ_1 and v_2 has type $[x \mapsto v_1]\tau_2$. In the process above, the type of channel $fact$ can be expressed by $[\Sigma n : \mathbf{Nat}.[\mathbf{Nat}]/O_\infty^{3n}.\mathbf{0}]/U$. It means that $fact$ is used to communicate pairs of a natural number n and a channel x, and

x is used for output within time $3n$. In order for type-checking to work, explicit type annotations by programmers would be necessary. It would also be necessary to impose some restriction on arithmetic expressions that can appear in dependent types, as in DML, an extension of ML with dependent types [36].

5.2. Polymorphism

Polymorphism in the π-calculus [26] is also useful to improve the expressive power of our type system. For example, let us consider the following process.

$$*id(x,r).\,\overline{r}\langle x\rangle \mid (\nu r')\,(\overline{id}\langle y,r'\rangle \mid r'(u).\,\overline{u}\langle\,\rangle) \mid (\nu r'')\,(\overline{id}\langle z,r'\rangle \mid r''(u).\,u(\,).\,\mathbf{0})$$

The process $*id(x,r).\,\overline{r}\langle x\rangle$ works as an identity function: It receives a pair of a value v and a reply channel r, and returns v to r. The process $(\nu r')\,(\overline{id}\langle y,r'\rangle \mid r'(u).\,\overline{u}\langle\,\rangle)$ calls the identify function, and then uses the returned channel y for output, while $(\nu r'')\,(\overline{id}\langle z,r'\rangle \mid r''(u).\,u(\,).\,\mathbf{0})$ calls the identify function and then uses the returned channel for input. Therefore, the type assigned to x, y, and z is of the form $[\,]/(O^{\infty}_{t_1}.\mathbf{0} \mid I^{\infty}_{t_2}.\mathbf{0})$, from which we cannot tell whether y and z are used for input or output.

If we introduce polymorphism as in the polymorphic π-calculus [26], we can rewrite the process above as:

$$\begin{array}{l} *id(\rho; x:\rho, r:[\rho]/O^{0}_{\infty}.\mathbf{0}).\,\overline{r}\langle x\rangle \\ \mid (\nu r')\,(\overline{id}\langle [\,]/O^{0}_{t_1}.\mathbf{0}; y, r'\rangle \mid r'(u).\,\overline{u}\langle\,\rangle) \mid (\nu r'')\,(\overline{id}\langle [\,]/I^{0}_{t_1}.\mathbf{0}; z, r'\rangle \mid r''(u).\,u(\,).\,\mathbf{0}) \end{array}$$

Here, channel id carries an additional parameter ρ, which denotes the type of x. With this modification, the types of y and z become $[\,]/O^{0}_{t_1}.\mathbf{0}$ and $[\,]/I^{0}_{t_1}.\mathbf{0}$, which imply that y and z are used for output and input, respectively.

5.3. Dependencies between Different Channels

Another shortcoming of our type system is that it loses some information about dependencies between different channels. Let us consider the following process.

$$*ping(\,).\,\overline{pong}\langle\,\rangle \mid \overline{ping}\langle\,\rangle.\,pong(\,)^{\mathbf{c}}.\,\mathbf{0}$$

The process $*ping(\,).\,\overline{pong}\langle\,\rangle$ listens to receive a message on channel $ping$ and sends an acknowledgment on channel $pong$. The process $\overline{ping}\langle\,\rangle.\,pong(\,)^{\mathbf{c}}.\,\mathbf{0}$ sends a message on $ping$ and waits to receive an acknowledgment on $pong$. Our type system in Section 3 cannot guarantee that an acknowledgment is received. Let the type of $ping$ be $[\,]/(*I^{t_1}_{t_2}.\mathbf{0} \mid O^{t_3}_{t_4}.\mathbf{0})$ and $pong$ be $[\,]/(*O^{t_5}_{t_6}.\mathbf{0} \mid I^{t_7}_{t_8}.\mathbf{0})$. In order for the input on $pong$ to be guaranteed to succeed, it must be the case that $t_8 < \infty$, which implies that $t_5 < \infty$. In order for $t_5 < \infty$ to hold, it must be the case that $t_2 < \infty$ (since $t_2 < t_5$ must hold in order for $ping(\,).\,\overline{pong}\langle\,\rangle$ to be well typed). This cannot hold, however, because the usage $*I^{t_1}_{t_2}.\mathbf{0} \mid O^{t_3}_{t_4}.\mathbf{0}$ of $ping$ must be reliable and it is reduced to $*I^{t_1}_{t_2}.\mathbf{0}$.

The problem above arises because our type system does not keep information that the obligation to send a message on $pong$ arises only after a message is received on $ping$. To overcome this problem, we must extend types and type environments to express the usage of multiple channels together. Using an idea of our recent generic type system for the π-calculus [13], we can express a combined usage of channels *ping* and *pong* as $*ping^0_\infty.\overline{pong}^0_\infty.\mathbf{0} \,|\, \overline{ping}^\infty_0.pong^\infty_0.\mathbf{0}$. Here, I and O are replaced with $ping, pong$, $\overline{ping}$, and $\overline{pong}$ to express which channel is used for input and output. The usage implies that a process is always waiting to receive a message on *ping*, and sends a message on *pong* immediately after a message arrives on *ping*, and that a process can successfully send a message on *ping* and then receive a message on *pong*. In order to check validity (reliability) of the usage, we can reduce it as follows.

$$\begin{aligned} *ping^0_\infty.\overline{pong}^0_\infty.\mathbf{0} \,|\, \overline{ping}^\infty_0.pong^\infty_0.\mathbf{0} &\longrightarrow *ping^0_\infty.\overline{pong}^0_\infty.\mathbf{0} \,|\, \overline{pong}^0_\infty.\mathbf{0} \,|\, pong^\infty_0.\mathbf{0} \\ &\longrightarrow *ping^0_\infty.\overline{pong}^0_\infty.\mathbf{0}. \end{aligned}$$

Each capability (a usage of the form $\alpha^t_0.U$) is matched by a corresponding obligation (a usage of the form $\overline{\alpha}^0_t.U'$). Hence, we see that $*ping^0_\infty.\overline{pong}^0_\infty.\mathbf{0} \,|\, \overline{ping}^\infty_0.pong^\infty_0.\mathbf{0}$ is valid and that the input on *pong* is guaranteed to succeed.

6. RELATED WORK

Our Previous Type Systems for Deadlock-Freedom. The type system in this paper has evolved from our previous type systems for deadlock-freedom [14, 17, 34] by extending the usages of channels with the notion of time limits. We think that a similar type system for deadlock-freedom can be obtained from the type system in Section 3, by replacing the rules for input and output with the following rules (where $t_c < \Gamma$ means that all the time limits for the success of actions in Γ should be greater than t_c):

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U \vdash P \qquad a = \mathbf{c} \Rightarrow t_c < \infty \qquad t_c < (v_1 : \tau_1 \,|\, \cdots \,|\, v_n : \tau_n \,|\, \Gamma)}{v_1 : \tau_1 \,|\, \cdots \,|\, v_n : \tau_n \,|\, \Gamma \,|\, x : [\tau_1, \ldots, \tau_n]/O^0_{t_c}.U \vdash \overline{x}\langle v_1, \ldots, v_n\rangle^a.\, P} \quad \text{(T-OUT')}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]/U, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash P \qquad a = \mathbf{c} \Rightarrow t_c < \infty \qquad t_c < \Gamma}{\Gamma, x : [\tau_1, \ldots, \tau_n]/I^0_{t_c}.U \vdash x(y_1, \ldots, y_n)^a.\, P} \quad \text{(T-IN')}$$

Nice points about the new type system are that time tags [14, 34] and usages are integrated and that we can get rid of complex side conditions on the time tags (which were introduced to get enough expressive power) in the typing rules of the previous type systems [14, 34]. 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 5.

Other Type Systems to Analyze Similar Properties of Concurrent Processes. To the author's knowledge, among previous type systems for languages of π-calculus family, only Sangiorgi's type system for receptiveness [32] and Yoshida et al.'s recent type system for strong normalization [39] can guarantee the lock-freedom property. Sangiorgi's type system deals with more specific situations than our type system: It ensures 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. Yoshida et al's type system guarantees the termination of every well-typed process, which is a stronger property than lock-freedom. As discussed in Section 1, this approach seems too restrictive for our purpose.

There are other type systems that deal with some deadlock-freedom property [1, 29, 38]. Please refer to our previous paper [17] for comparisons with them.

Recently, we have extended the idea of augmenting a type with information about usage of each channel and developed a generic type system for the π-calculus [13], which can be used to verify various properties of processes, like deadlock-freedom [14, 17, 34] and race-freedom [9, 10]. The generic type system incorporates the extension outlined in Section 5.3, to keep track of dependencies between different channels. Unfortunately, however, the generic type system is not general enough to subsume the type system described in this paper. It is left for future work to integrate the type system of this paper and the generic type system.

Abadi and Flanagan [10] developed type systems to guarantee race-freedom for a concurrent language with reference cells and lock primitives. They sketch an extension of the type system to avoid deadlocks (without a proof of correctness), by allowing a programmer to specify a partial order on locks (binary semaphores). The partial order roughly corresponds to the order induced by time limits in this paper and the tag ordering used in our previous type systems for deadlock-freedom [14, 17, 34]. However, their type system does not prevent livelocks.

Type Systems for Bounding Execution Time of Sequential Programs. There are several pieces of work that try to statically bound running-time of sequential programs [6, 12]. A major difficulty in bounding the 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 limits: a time limit within which a process executes the operation, and another time limit within which the process can successfully complete the operation.

Abstract Interpretation. An alternative way to analyze the behavior of a concurrent program would be to use abstract interpretation [4, 5]. Actually, from a very general viewpoint, our type-based analysis of locks 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 3.7.

The subject reduction property (Theorem 3.1) 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. A concrete process is guaranteed to be lock-free, because the reliability condition (Definition 3.16) guarantees that an abstract process never falls into a lock,

7. CONCLUSION

In this paper, we have extended our previous type systems for deadlock-freedom to guarantee the lock-freedom and time-boundedness properties. The type system for lock-freedom given in Section 3 guarantees that certain communications succeed eventually on the assumption of strong fairness. Our contributions about this type system include formal definitions of fairness and lock-freedom in the π-calculus. We have also devised a novel technique to prove the soundness of our type system. The type system given in Section 4 guarantees that certain communications succeed within a given number of parallel reduction steps. We have defined parallel reductions in the π-calculus, and proved the soundness of the type system in a novel manner, using a property that well-typedness is preserved by parallel reductions.

In applying our type systems to real programming languages, we must develop a type-checking or type inference algorithm. We can probably use an algorithm similar to the type inference algorithm for type systems for deadlock-freedom [17]. The details are left for future work. A number of practical issues also remain to be solved. For example, we must address how to combine dependent types, polymorphism, etc., and how and to what extent to let programmers supply 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 and memory management. The former application has been exploited in our recent generic type system [13]. As for the latter application (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 [15].

Acknowledgment

We would like to especially thank Takayasu Ito and anonymous referees for numerous useful suggestions and comments. We would also like to thank Atsushi Igarashi, Benjamin Pierce, Eijiro Sumii, and Nobuko Yoshida for discussions and comments. This research is partially supported by Grant-in-Aid (12133202) for Scientific Research, Ministry of Education, Culture, Sports, Science and Technology.

REFERENCES

1. Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Proceedings of ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, 1997.

2. Liang Chen. Timed processes: Models, axioms and decidability. PhD Thesis, University of Edinburgh, 1992.

3. Berardo Costa and Colin Stirling. Weak and strong fairness in CCS. *Information and Computation*, 73(3):207–244, 1987.

4. Patrick Cousot. Types as abstract interpretations. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 316–331, 1997.

5. Patrick Cousot and Radhia 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.

6. Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 184–198, 2000.

7. E. Allen Emerson. Temporal and modal logic. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B*, chapter 16, pages 995–1072. The MIT press/Elsevier, 1990.

8. J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994.

9. Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.

10. Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proceedings of ESOP 1999*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, 1999.

11. Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

12. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 464–473, 1999.

13. Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 128–141, January 2001.

14. Naoki 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.

15. Naoki Kobayashi. Quasi-linear types. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 29–42, 1999.

16. Naoki Kobayashi, Benjamin C. Pierce, and David 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.

17. Naoki Kobayashi, Shin Saito, and Eijiro 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 has appeared in Proceedings of CONCUR 2000, Springer LNCS1877, pp.489-503, 2000.

18. Naoki Kobayashi and Akinori Yonezawa. Towards foundations for concurrent object-oriented programming – types and language design. *Theory and Practice of Object Systems*, 1(4):243–268, 1995.

19. E. V. Krishnamurthy. *Parallel Processing: Principles and Practice*. Addison-Wesley, 1989.

20. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

21. Robin 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.

22. Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

23. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100:1–77, September 1992.

24. John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

25. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

26. Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the Association for Computing Machinery (JACM)*, 47(5):531–584, 2000.

27. Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, 1995.

28. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

29. Franz Puntigam. Coordination requirements expressed in types for active objects. In *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388, 1997.

30. John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

31. John 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.

32. Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1-2):457–493, 1999.

33. Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

34. Eijiro Sumii and Naoki 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.

35. David T. Turner. The polymorphic pi-calculus: Theory and implementation. PhD Thesis, University of Edinburgh, 1996.

36. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 214–227, 1999.

37. Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.

38. Nobuko Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–387. Springer-Verlag, 1996. Full version as LFCS report, ECS-LFCS-96-350, University of Edinburgh.

39. Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalization in the π-calculus. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 2001.

APPENDIX A

A.1. PROOFS OF LEMMAS AND THEOREMS IN SECTION 3

A.1.1. Proofs of Lemmas 3.4 and 3.5

We first prove auxiliary lemmas. The following lemma means that if a usage has an obligation or capability to perform some action, its subusage also has a corresponding obligation or capability to perform the same action.

LEMMA A.1. *Suppose $U \leq V$. Then the following conditions hold:*

A.If $V \cong \alpha^{t_o}_{t_c}.V_1 \,|\, V_2$, then there exist U_1, U_2, t'_o, t'_c such that $U \cong \alpha^{t'_o}_{t'_c}.U_1 \,|\, U_2$, $U_1 \,|\, U_2 \leq V_1 \,|\, V_2$, $t_o \leq t'_o$, and $t'_c \leq t_c$.

B.If $V \cong I^{t_{o1}}_{t_{c1}}.V_1 \,|\, O^{t_{o2}}_{t_{c2}}.V_2 \,|\, V_3$, then there exist $U_1, U_2, U_3, t'_{o1}, t'_{c1}, t'_{o2}, t'_{c2}$ such that $U \cong I^{t'_{o1}}_{t'_{c1}}.U_1 \,|\, O^{t'_{o2}}_{t'_{c2}}.U_2 \,|\, U_3$, $U_1 \,|\, U_2 \,|\, U_3 \leq V_1 \,|\, V_2 \,|\, V_3$, $t_{o1} \leq t'_{o1}$, $t_{o2} \leq t'_{o2}$, $t'_{c1} \leq t_{c1}$, and $t'_{c2} \leq t_{c2}$.

Proof. The proof proceeds by induction on derivation of $\leq$, with case analysis on the last rule.

- Case for (SubU-Cong): Trivial (Let $U_1 = V_1$, $U_2 = V_2$, and $U_3 = V_3$).
- Case for (SubU-Zero): This cannot happen.

• Case for (SubU-Delay): Case B is vacuously true. So, we need to show only A. Suppose $V \cong \alpha^{t_o}_{t_c}.V_1 \,|\, V_2$. It must be the case that $V_1 \cong V_{11} \,|\, V_{12}$, $V_2 \cong \mathbf{0}$, and $U = \alpha^{t_o}_{t_c}.V_{11} \,|\, \boxed{t_o + t_c + 1} V_{12}$. So, the required result holds for $t'_o = t_o$, $t'_c = t_c$, $U_1 = V_{11}$, and $U_2 = \boxed{t_o + t_c + 1} V_{12}$.

• Case for (SubU-Act): Case B is vacuously true. So, we need to show only A. Suppose $V \cong \alpha^{t_o}_{t_c}.V_1 \,|\, V_2$. It must be the case that

$$\begin{array}{lll} U = \alpha^{t'_o}_{t'_c}.U' & V = \alpha^{t_o}_{t_c}.V' & U' \leq V' \\ V_1 \cong V' & V_2 \cong \mathbf{0} & \\ t_o \leq t'_o & t'_c \leq t_c & \end{array}$$

The required result holds for $U_1 = U'$ and $U_2 = \mathbf{0}$.

• Case for (SubU-Par): In this case, $V = V_1 \,|\, V_2$, $U = U_1 \,|\, U_2$, $U_1 \leq V_1$, and $U_2 \leq V_2$. We show only B: The proof of A is similar and simpler. Suppose $V \cong I^{t_{o1}}_{t_{c1}}.V_4 \,|\, O^{t_{o2}}_{t_{c2}}.V_5 \,|\, V_6$. Then one of the following conditions hold:

(a) $V_i \cong I^{t_{o1}}_{t_{c1}}.V_4 \,|\, O^{t_{o2}}_{t_{c2}}.V_5 \,|\, V'_6$ and $V'_6 \,|\, V_{3-i} \cong V_6$ for $i = 1$ or 2.

(b) $V_i \cong I^{t_{o1}}_{t_{c1}}.V_4 \,|\, V'_6$, $V_{3-i} \cong O^{t_{o2}}_{t_{c2}}.V_5 \,|\, V''_6$, and $V'_6 \,|\, V''_6 \cong V_6$ for $i = 1$ or 2.

Since the case (a) is trivial by induction hypothesis, we show only the case (b). Without loss of generality, we can assume that $i = 1$. By induction hypothesis, we have

$$\begin{array}{ll} U_1 \cong I^{t'_{o1}}_{t'_{c1}}.U_4 \,|\, U'_6 & U_2 \cong O^{t'_{o2}}_{t'_{c2}}.U_5 \,|\, U''_6 \\ U_4 \,|\, U'_6 \leq V_4 \,|\, V'_6 & U_5 \,|\, U''_6 \leq V_5 \,|\, V''_6 \\ t_{o1} \leq t'_{o1} & t_{o2} \leq t'_{o2} \\ t'_{c1} \leq t_{c1} & t'_{c2} \leq t_{c2}. \end{array}$$

Let $U_6 = U'_6 \,|\, U''_6$. Then, we have $U \cong I^{t'_{o1}}_{t'_{c1}}.U_4 \,|\, O^{t'_{o2}}_{t'_{c2}}.U_5 \,|\, U_6$ and $U_4 \,|\, U_5 \,|\, U_6 \leq V_4 \,|\, V_5 \,|\, V_6$ as required.

• Case for (SubU-Rep): In this case, $V = *V'$, $U = *U'$, and $U' \leq V'$. We show only B. The proof of A is similar. Suppose $V \cong I^{t_{o1}}_{t_{c1}}.V_1 \,|\, O^{t_{o2}}_{t_{c2}}.V_2 \,|\, V_3$. Then, it must be the case that $V' \cong I^{t_{o1}}_{t_{c1}}.V_1 \,|\, O^{t_{o2}}_{t_{c2}}.V_2 \,|\, V'_3$ and $V'_3 \,|\, *V' \cong V_3$. By induction hypothesis, $U' \cong I^{t'_{o1}}_{t'_{c1}}.U_1 \,|\, O^{t'_{o2}}_{t'_{c2}}.U_2 \,|\, U'_3$, $U_1 \,|\, U_2 \,|\, U'_3 \leq V_1 \,|\, V_2 \,|\, V'_3$, $t_{o1} \leq t'_{o1}$, $t_{o2} \leq t'_{o2}$, $t'_{c1} \leq t_{c1}$, and $t'_{c2} \leq t_{c2}$. Let $U_3 = U'_3 \,|\, *U'$. Then, we have $U \cong I^{t'_{o1}}_{t'_{c1}}.U_1 \,|\, O^{t'_{o2}}_{t'_{c2}}.U_2 \,|\, U_3$ and $U_1 \,|\, U_2 \,|\, U_3 \leq (U_1 \,|\, U_2 \,|\, U'_3) \,|\, *U' \leq (V_1 \,|\, V_2 \,|\, V'_3) \,|\, *V' \leq V_1 \,|\, V_2 \,|\, V_3$ as required.

• Case for the rule for transitivity: The result follows immediately by using induction hypothesis twice.

■

The following lemma means that if a usage can be reduced, its subusage also has a corresponding reduction.

Lemma A.2. *If $U \leq V$ and $V \longrightarrow V'$, then there exists U' such that $U \longrightarrow U'$ and $U' \leq V'$. If $\Gamma \leq \Gamma'$ and $\Gamma' \xrightarrow{l} \Delta'$, then there exists Δ such that $\Delta \leq \Delta'$ and $\Gamma \xrightarrow{l} \Delta$.*

Proof. We show the former property by induction on derivation of $V \longrightarrow V'$. The latter property is an immediate corollary. We have two cases to analyze.

• Case where $V = I^{t_o}_{t_c}.V_1 \,|\, O^{t'_o}_{t'_c}.V_2 \,|\, V_3$ and $V' = V_1 \,|\, V_2 \,|\, V_3$: By Lemma A.1, it must be the case that

$$U \cong I^{t_{o1}}_{t_{c1}}.U_1 \,|\, O^{t'_{o1}}_{t'_{c1}}.U_2 \,|\, U_3$$
$$U_1 \,|\, U_2 \,|\, U_3 \leq V_1 \,|\, V_2 \,|\, V_3$$

So, we have the required result for $U' = U_1 \,|\, U_2 \,|\, U_3$.

• Case where $V \cong V_1 \longrightarrow V'_1 \cong V'$: By $U \leq V \leq V_1$, $V_1 \longrightarrow V'_1$ and induction hypothesis, there exists U' such that $U \longrightarrow U' \leq V'_1 \leq V'$.

■

Proof of Lemma 3.4. Since the case where $t = \infty$ is trivial, suppose $t < \infty$. The proof proceeds by induction on derivation of $U \leq V$, with case analysis on the last rule. We show only the case for rule (SubU-Delay) and the rule for transitivity. The other cases are similar or trivial.

• Case for (SubU-Delay): It must be the case that $U = \alpha'^{t_o}_{t_c}.U_1 \,|\, \boxed{t_o + t_c + 1} U_2$ and $V = \alpha'^{t_o}_{t_c}.(U_1 \,|\, U_2)$. $ob_\alpha(t, U)$ implies either $ob_\alpha(t, \alpha'^{t_o}_{t_c}.U_1)$ or $ob_\alpha(t, \boxed{t_o + t_c + 1} U_2)$. In the former case, $\alpha = \alpha'$ and $t_o \leq t$, which implies $ob_\alpha(t, V)$. In the latter case, $t_o + t_c + 1 \leq t$. So, if $\alpha = \alpha'$, then $ob_\alpha(t, V)$ holds. If $\alpha' = \overline{\alpha}$, then $cap_{\overline{\alpha}}(t_c, U)$ and $t_c < t$ hold as required.

• Case for the rule for transitivity: It must be the case that $U \leq U' \leq V$. By applying induction hypothesis to $U \leq U'$, we have either $ob_\alpha(t, U')$ or $cap_{\overline{\alpha}}(t_c, U)$ for some t_c with $t_c < t$. In the former case, by applying induction hypothesis again to $U' \leq V$, we have either $ob_\alpha(t, V)$ or $cap_{\overline{\alpha}}(t_c, U')$ for some t_c with $t_c < t$. If $cap_{\overline{\alpha}}(t_c, U')$ holds, $cap_{\overline{\alpha}}(t_c, U)$ follows from Lemma A.1, as required. ■

The following lemma means that for reliable usages, $ob_\alpha(t, U)$ is closed under the subusage relation.

Lemma A.3. *If $rel(U)$, $U \leq V$, and $ob_\alpha(t, U)$, then $ob_\alpha(t, V)$ holds.*

Proof. If $t = \infty$, then $ob_\alpha(t, V)$ always holds. We show the case for $t < \infty$ by induction on t. Suppose that the lemma holds for any t' such that $t' < t$. By Lemma 3.4, either $ob_\alpha(t, V)$ holds or $cap_{\overline{\alpha}}(t_c, U)$ holds for some $t_c(< t)$. In the latter case, by the assumption $rel(U)$, it must be the case that $ob_\alpha(t_c, U_2)$, which also implies $ob_\alpha(t_c, U)$. So, by induction hypothesis, we have $ob_\alpha(t_c, V)$, which implies $ob_\alpha(t, V)$. ■

Proof of Lemma 3.5. Suppose $U \longrightarrow^* \leq V$, $rel(U)$, and $V \longrightarrow^* V'$ with $cap_\alpha(t_c, V')$. It suffices to show $ob_{\overline{\alpha}}(t_c, V')$. By Lemma A.2, there exists U' such that $U \longrightarrow^* U'$ with $U' \leq V'$. By Lemma A.1, we have $cap_\alpha(t'_c, U')$ for some $t'_c(\leq t_c)$. So, by the assumption $rel(U)$, it must be the case that $ob_{\overline{\alpha}}(t'_c, U')$. By using Lemma A.3, we obtain $ob_{\overline{\alpha}}(t_c, V')$. ■

A.1.2. Proofs of the Subject Reduction Properties

Lemma A.4 (substitution lemma). If $\Gamma, x:\tau \vdash P$ and $\Gamma \,|\, v:\tau$ is well defined, then $\Gamma \,|\, v:\tau \vdash [x \mapsto v]P$ holds.

Proof.
We prove this by induction on the structure of P. We show only the case where P is an input process: The other cases are similar or trivial. Suppose $P = w(\tilde{z})^a.\, Q$. The only non-trivial cases are where $w = x$ and v appears in Q and where $w = v$ and x appears in Q: the other cases are trivial by induction hypothesis. Suppose $w = x$ and v appears in Q. By the typing rules, we have

$$\begin{array}{l}
\Gamma_1, x:[\tilde{\tau}]/U_1, v:[\tilde{\tau}]/U_2, \tilde{z}:\tilde{\tau} \vdash Q \\
\Gamma \leq \boxed{t_c+1}(\Gamma_1, v:[\tilde{\tau}]/U_2) \\
\tau \leq [\tilde{\tau}]/I^0_{t_c}.U_1 \\
a = \mathbf{c} \Rightarrow t_c < \infty
\end{array}$$

By induction hypothesis, we have $\Gamma_1, v:[\tilde{\tau}]/(U_1 \,|\, U_2), \tilde{z}:\tilde{\tau} \vdash [x \mapsto v]Q$. By using (T-In), we obtain:

$$\boxed{t_c+1}\Gamma_1, v:[\tilde{\tau}]/I^0_{t_c}.(U_1 \,|\, U_2) \vdash v(\tilde{z})^a.\,([x \mapsto v]Q)(= [x \mapsto v]P).$$

Since $\boxed{t_c+1}U_2 \,|\, I^0_{t_c}.U_1 \leq I^0_{t_c}.(U_1 \,|\, U_2)$ holds, we have

$$\begin{array}{rcl}
\Gamma \,|\, v:\tau & \leq & \boxed{t_c+1}(\Gamma_1, v:[\tilde{\tau}]/U_2) \,|\, v:[\tilde{\tau}]/I^0_{t_c}.U_1 \\
& \leq & \boxed{t_c+1}\Gamma_1, v:[\tilde{\tau}]/I^0_{t_c}.(U_1 \,|\, U_2).
\end{array}$$

We have therefore $\Gamma \,|\, v:\tau \vdash [x \mapsto v]P$ as required.

The case where $w = v$ is similar. ■

Lemma A.5. *If $\Gamma, x:\tau \vdash P$ and x is not free in P, then $noob(\tau)$ and $\Gamma \vdash P$ hold.*

Proof. Trivial from the fact that $x:\tau$ can be introduced only by rule (T-Weak). ■

Proof (Proof of Lemma 3.2). By the typing rules, we have

$$\begin{array}{l}
\Gamma_1, x:[\tilde{\tau}]/U_1 \vdash P \\
\Gamma_2, x:[\tilde{\tau}]/U_2, y_1:\tau_1, \ldots, y_n:\tau_n \vdash Q \\
\Gamma \leq \boxed{t_c+1}(\Gamma_1 \,|\, v_1:\tau_1 \,|\, \cdots \,|\, v_n:\tau_n) \,|\, \boxed{t'_c+1}\Gamma_2 \\
U \leq O^0_{t_c}.U_1 \,|\, I^0_{t'_c}.U_2
\end{array}$$

By the substitution lemma (Lemma A.4), we have

$$\Gamma_2 \,|\, v_1:\tau_1 \,|\, \cdots \,|\, v_n:\tau_n, x:[\tilde{\tau}]/U_2 \vdash [\tilde{y} \mapsto \tilde{v}]Q.$$

So, by using (T-Par), we obtain

$$\Gamma_1 \,|\, \Gamma_2 \,|\, v_1:\tau_1 \,|\, \cdots \,|\, v_n:\tau_n, x:[\tilde{\tau}]/(U_1 \,|\, U_2) \vdash P \,|\, [\tilde{y} \mapsto \tilde{v}]Q.$$

By Lemma A.2 and the condition $U \leq O^{t_o}_{t_c}.U_1 \,|\, I^{t'_o}_{t'_c}.U_2$, there exists U' such that $U \longrightarrow U'$ and $U' \leq U_1 \,|\, U_2$. Let $\Delta = \Gamma_1 \,|\, \Gamma_2 \,|\, v_1 : \tau_1 \,|\, \cdots \,|\, v_n : \tau_n$. Then we have $\Delta, x : [\tilde{\tau}]/U' \vdash P \,|\, [\tilde{y} \mapsto \tilde{v}]Q$ and $\Gamma \leq \boxed{t_c + 1}(\Gamma_1 \,|\, v_1 : \tau_1 \,|\, \cdots \,|\, v_n : \tau_n) \,|\, \boxed{t'_c + 1}\Gamma_2 \leq \boxed{1}\Delta$ as required. ∎

Lemma A.6. *If $\Gamma \vdash P$ and $P \preceq Q$, then $\Gamma \vdash Q$.*

Proof. The proof proceeds by induction on derivation of $P \preceq Q$ with case analysis on the last rule used. We show only the cases for (SPCong-New) and (SPCong-Rep). The other cases are trivial,

• Case for (SPCong-New) (in both directions): Suppose $\Gamma \vdash (\nu x)\,(P_1 \,|\, P_2)$ and x is not free in P_2. By the typing rules, we have

$$\begin{array}{l}\Gamma_1, x : [\tilde{\tau}]/U_1 \vdash P_1\\ \Gamma_2, x : [\tilde{\tau}]/U_2 \vdash P_2\\ U \leq U_1 \,|\, U_2\\ \mathit{rel}(U)\\ \Gamma \leq \Gamma_1 \,|\, \Gamma_2\end{array}$$

By Lemma A.5, we have $\Gamma_2 \vdash P_2$ and $U_2 \leq \mathbf{0}$. So, by using (T-Weak), we get $\Gamma_1, x : [\tilde{\tau}]/(U_1 \,|\, U_2) \vdash P_1$. By using (T-New), (T-Par), and (T-Weak), we obtain $\Gamma \vdash (\nu x)\, P_1 \,|\, P_2$ as required.

On the other hand, suppose $\Gamma \vdash (\nu x)\, P_1 \,|\, P_2$. By the typing rules, we have

$$\begin{array}{l}\Gamma_1, x : [\tilde{\tau}]/U \vdash P_1\\ \Gamma_2 \vdash P_2\\ \mathit{rel}(U)\\ \Gamma \leq \Gamma_1 \,|\, \Gamma_2\end{array}$$

By Lemma A.5, we have $\Gamma_2 \backslash \{x\} \vdash P_2$ and $\Gamma_2 \leq \Gamma_2 \backslash \{x\}$. So, we have $\Gamma_1 \,|\, (\Gamma_2 \backslash \{x\}) \vdash (\nu x)\,(P_1 \,|\, P_2)$. Since $\Gamma \leq \Gamma_1 \,|\, \Gamma_2 \leq \Gamma_1 \,|\, (\Gamma_2 \backslash \{x\})$ holds, we obtain $\Gamma \vdash (\nu x)\,(P_1 \,|\, P_2)$ by using (T-Weak).

• Case for (SPCong-Rep): It must be the case that $P = *R$ and $Q = *R \,|\, R$. By the typing rules, it must be the case that $\Delta \vdash R$ and $\Gamma \leq *\Delta$ for some Δ. By using (T-Par) and (T-Rep), we obtain $*\Delta \,|\, \Delta \vdash Q$. By the fact $*U \cong *U \,|\, U$, we have $\Gamma \leq *\Delta \leq *\Delta \,|\, \Delta$. So, by using (T-Weak), we obtain $\Gamma \vdash *R \,|\, R$ as required.

∎

Proof of Theorem 3.1. The proof proceeds by induction on derivation of $P \xrightarrow{l} Q$, with case analysis on the last rule used.

• Case (R-Com): This follows immediately from Lemma 3.2 and rule (T-Weak).

• Case (R-Par): It must be the case that $P = P_1 \,|\, P_2$, $Q = Q_1 \,|\, P_2$, and $P_1 \xrightarrow{l} Q_1$. By the typing rules and $\Gamma \vdash P$, there must exist Γ_1 and Γ_2 such that $\Gamma_1 \vdash P_1$, $\Gamma_2 \vdash P_2$, and $\Gamma \leq \Gamma_1 \,|\, \Gamma_2$. By induction hypothesis, we have Δ_1 such that $\Delta_1 \vdash Q_1$ and $\Gamma_1 \xrightarrow{l} \Delta_1$, which also implies $\Gamma_1 \,|\, \Gamma_2 \xrightarrow{l} \Delta_1 \,|\, \Gamma_2$. By Lemma A.2, there exists

Δ such that $\Gamma \xrightarrow{l} \Delta$ and $\Delta \leq \Delta_1 \,|\, \Gamma_2$. By using (T-PAR) and (T-WEAK), we obtain $\Delta \vdash Q$ as required.

• Cases for (R-NEW1) and (R-NEW2): Trivial by induction hypothesis and Lemma 3.5.

• Case for (R-IFT): It must be the case that $P = \mathbf{if}\ \mathit{true}\ \mathbf{then}\ Q\ \mathbf{else}\ R$ and $l = \epsilon$. By the typing rules, $\Gamma \leq \boxed{1}\Delta$ and $\Delta \vdash Q$. By using (T-WEAK), we have $\Gamma \vdash Q$ as required.

• Case for (R-IFF): Similar to the case for (R-IFT).

• Case for (R-SPCONG): Trivial by induction hypothesis and Lemma A.6. ■

APPENDIX B

B.1. PROOF OF PARALLEL SUBJECT REDUCTION THEOREM (THEOREM 4.2)

We prove the following, more general property than Theorem 4.2:

LEMMA B.1. *If $\Gamma \vdash_T P$ and $P \overset{S}{\Longrightarrow} Q$, then either of the following conditions holds:*

1. *There exists Δ such that $\Gamma \overset{S}{\Longrightarrow} \Delta$ and $\Delta \vdash_T Q$.*
2. *There exists $x \in \mathit{dom}(\Gamma)$ such that $x^\alpha \in S$ and $\mathit{cap}_{x^\alpha}(0, \Gamma)$.*

The lemma means that if a well-typed process is reduced, either the resulting process is well-typed under a reduced type environment (the first case above) or a time-out has occurred on a free channel. The lemma implies Theorem 4.1 because the second case cannot happen if $\mathit{rel}_T(\Gamma)$ holds. We prove Lemma B.1 after introducing several lemmas.

LEMMA B.2. *For any $U \in \mathcal{U}$ and $t \in \mathbf{Nat}^+$ such that $0 < t$, $\boxed{t}U \overset{\emptyset}{\Longrightarrow} \boxed{t-1}U$ holds. For any type environment Γ and $t \in \mathbf{Nat}^+$ such that $0 < t$, $\boxed{t}\Gamma \overset{\emptyset}{\Longrightarrow} \boxed{t-1}\Gamma$ holds.*

Proof. The former property follows by straightforward induction on the structure of U' (use (TU-ZERO) and (TU-SKIP) for base cases). The latter is an immediate corollary. ■

LEMMA B.3. *If $U \leq V$ and $V \overset{S}{\Longrightarrow} V'$, then one of the following conditions holds:*

1. *There exists U' such that $U \overset{S}{\Longrightarrow} U'$ and $U' \leq V'$.*
2. *$\alpha \in S$ and $\mathit{cap}_\alpha(0, U)$*

Proof. The proof proceeds by induction on derivation of $U \leq V$ with case analysis on the last rule. Without loss of generality, we can assume that (SUBU-REP) is applied only when the righthand usage of the premise is $\mathbf{0}$ or of the form $\alpha^{t_o}_{t_c}.U$. Suppose that the second condition of the lemma does not hold.

• Case for (SubU-Cong): It must be the case that $U \cong V$. So, $U' = V'$ satisfies the required condition.

• Case for (SubU-Zero): It must be the case that $U = \alpha^{t_o}_{t_c}.U_1$, $V = \mathbf{0}$, and $V' \cong \mathbf{0}$. So, $U' = U$ satisfies the required condition.

• Case for (SubU-Par): It must be the case that:

$$\begin{array}{ll} U = U_1 \,|\, U_2 & V = V_1 \,|\, V_2 \\ U_1 \leq V_1 & U_2 \leq V_2. \end{array}$$

By the assumption $V \overset{S}{\Longrightarrow} V'$, there exist V_1', V_2', S_1, and S_2 such that:

$$\begin{array}{ll} V' \cong V_1' \,|\, V_2' & \\ V_1 \overset{S_1}{\Longrightarrow} V_1' & V_2 \overset{S_2}{\Longrightarrow} V_2' \\ S = S_1 \cup S_2 & S_1 \cap S_2 \cap \{\mathbf{com}_I, \mathbf{com}_O\} = \emptyset. \end{array}$$

(This can be proved by induction on derivation of $V \overset{S}{\Longrightarrow} V'$.)

By induction hypothesis, there exist U_1' and U_2' such that:

$$\begin{array}{ll} U_1 \overset{S_1}{\Longrightarrow} U_1' & U_2 \overset{S_2}{\Longrightarrow} U_2' \\ U_1' \leq V_1' & U_2' \leq V_2'. \end{array}$$

So, $U' = U_1' \,|\, U_2'$ satisfies the required condition.

• Case for (SubU-Rep): It must be the case that:

$$\begin{array}{lll} U = *U_1 & V = *V_1 & U_1 \leq V_1 \\ V_1 \text{ is either } \mathbf{0} \text{ or } \alpha^{t_o}_{t_c}.V_2 & & \end{array}$$

If $V_1 = \mathbf{0}$, then it must be the case that $S = \emptyset$, $V' \cong \mathbf{0}$, and $V_1 \overset{S}{\Longrightarrow} \mathbf{0}$. So, by induction hypothesis, there exists U_1' such that $U_1 \overset{S}{\Longrightarrow} U_1'$ and $U_1' \leq V_1$. $U = *U_1$ satisfies the required condition.

If $V_1 = \alpha^{t_o}_{t_c}.V_2$, then because there are three ways to reduce V_1 (by using (U-ComX), (U-Wait) or (U-Skip)), there exist V_1', V_2', V_3' such that

$$\begin{array}{l} V' \cong V_1' \,|\, V_2' \,|\, V_3' \\ V_1' \in \{\mathbf{0}, V_2\} \\ V_2' \in \{\mathbf{0}, \alpha^{t_o-1}_{t_c}.V_2, *\alpha^{t_o-1}_{t_c}.V_2\} \\ V_3' \in \{\mathbf{0}, \alpha^{0}_{t_c-1}.V_2, *\alpha^{0}_{t_c-1}.V_2\} \\ V_2' = *\alpha^{t_o-1}_{t_c}.V_2 \text{ or } V_3' = *\alpha^{0}_{t_c-1}.V_2 \end{array}$$

Since the other cases are similar and simpler, we show only the case for $V_1' = V_2$, $V_2' = *\alpha^{t_o-1}_{t_c}.V_2$, and $V_3' = *\alpha^{0}_{t_c-1}.V_2$. In this case, $S = \{\mathbf{com}_\alpha, \alpha\}$. By applying induction hypothesis to

$$\begin{array}{l} U_1 \leq V_1 \overset{\{\mathbf{com}_\alpha\}}{\Longrightarrow} V_2 \\ U_1 \leq V_1 \overset{\emptyset}{\Longrightarrow} \alpha^{t_o-1}_{t_c}.V_2 \\ U_1 \leq V_1 \overset{\{\alpha\}}{\Longrightarrow} \alpha^{0}_{t_c-1}.V_2, \end{array}$$

we obtain U_1', U_2', U_3' such that:

$$\begin{array}{ll} U_1 \stackrel{\{\mathbf{com}_\alpha\}}{\Longrightarrow} U_1' & U_1' \leq V_2 \\ U_1 \stackrel{\emptyset}{\Longrightarrow} U_2' & U_2' \leq \alpha_{t_c}^{t_o-1}.V_2 \\ U_1 \stackrel{\{\alpha\}}{\Longrightarrow} U_3' & U_3' \leq \alpha_{t_c-1}^{0}.V_2. \end{array}$$

Let $U' = U_1' \,|\, {*U_2'} \,|\, {*U_3'}$. Then, we have $U = {*U_1} \cong U_1 \,|\, {*U_1} \,|\, {*U_1} \stackrel{S}{\Longrightarrow} U'$ and $U' \leq V'$ as required.

- Case for (SubU-Delay): It must be the case that

$$U = \alpha_{t_c}^{t_o}.U_1 \,|\, \boxed{t_o + t_c + 1} U_2 \qquad V = \alpha_{t_c}^{t_o}.(U_1 \,|\, U_2)$$

By Lemma B.2, we have $\boxed{t_o + t_c + 1} U_2 \stackrel{\emptyset}{\Longrightarrow} \boxed{t_o + t_c} U_2$. By the assumption $V \stackrel{S}{\Longrightarrow} V'$, one of the following conditions must hold:

1. $V' \cong U_1 \,|\, U_2$ and $S = \{\mathbf{com}_\alpha\}$.
2. $V' \cong \alpha_{t_c-1}^{0}.(U_1 \,|\, U_2)$, $S = \{\alpha\}$, and $t_c \neq 0$.
3. $V' \cong \alpha_{t_c}^{t_o-1}.(U_1 \,|\, U_2)$ and $S = \{\alpha\}$.

For each case, define U' by:

1. $U' = U_1 \,|\, \boxed{t_o + t_c} U_2$.
2. $U' = \alpha_{t_c-1}^{0}.U_1 \,|\, \boxed{t_o + t_c} U_2$.
3. $U' = \alpha_{t_c}^{t_o-1}.U_1 \,|\, \boxed{t_o + t_c} U_2$.

Then, $U \stackrel{S}{\Longrightarrow} U'$ holds. $U' \leq V'$ can be proved as follows:

$$\begin{array}{rcl} U_1 \,|\, \boxed{t_o + t_c} U_2 & \leq & U_1 \,|\, U_2 \\ \alpha_{t_c-1}^{0}.U_1 \,|\, \boxed{t_o + t_c} U_2 & \leq & \alpha_{t_c-1}^{0}.U_1 \,|\, \boxed{0 + (t_c - 1) + 1} U_2 \\ & \leq & \alpha_{t_c-1}^{0}.(U_1 \,|\, U_2) \\ \alpha_{t_c}^{t_o-1}.U_1 \,|\, \boxed{t_o + t_c} U_2 & \leq & \alpha_{t_c}^{t_o-1}.U_1 \,|\, \boxed{(t_o - 1) + t_c + 1} U_2 \\ & \leq & \alpha_{t_c}^{t_o-1}.(U_1 \,|\, U_2) \end{array}$$

- Case for (SubU-Act): It must be the case that

$$\begin{array}{lll} U = \alpha_{t_c}^{t_o}.U_1 & V = \alpha_{t_c'}^{t_o'}.V_1 & \\ U_1 \leq V_1 & t_o' \leq t_o & t_c \leq t_c'. \end{array}$$

By the assumption $V \stackrel{S}{\Longrightarrow} V'$, one of the following conditions must hold:

1. $V' \cong V_1$ and $S = \{\mathbf{com}_\alpha\}$.
2. $V' \cong \alpha_{t_c'-1}^{0}.V_1$, $S = \{\alpha\}$, and $t_c \neq 0$.
3. $V' \cong \alpha_{t_c'}^{t_o'-1}.V_1$ and $S = \{\alpha\}$.

The required conditions hold if we define U' by:

1. $U' = U_1$
2. $U' = \alpha^{0}_{t_c-1}.U_1$
3. $U' = \alpha^{t_o-1}_{t_c}.U_1$

for each case.

- Case for the rule for transitivity: Trivial by induction hypothesis.

■

LEMMA B.4. *If* $\Gamma \leq \Delta$ *and* $\Delta \overset{S}{\Longrightarrow} \Delta'$*, then one of the following conditions holds:*

1. *There exists* Γ' *such that* $\Gamma \overset{S}{\Longrightarrow} \Gamma'$ *and* $\Gamma' \leq \Delta'$*.*
2. *There exists* $x \in dom(\Gamma)$ *such that* $x^\alpha \in S$ *and* $cap_{x^\alpha}(0, \Gamma)$*.*

Proof. This follows immediately from Lemma B.3. ■

We are now ready to prove Lemma B.1, from which Theorem 4.2 follows.

Proof (Proof of Lemma B.1). The proof proceeds by induction on derivation of $P \overset{S}{\Longrightarrow} Q$, with case analysis on the last rule used. Suppose that the second condition of the theorem does not hold.

- Case for (PR-COM): It must be the case that:

$$\begin{array}{l} P = \overline{x}\langle\tilde{v}\rangle^{t}.\,R_1 \,|\, x(\tilde{y})^{t'}.\,R_2 \\ Q = R_1 \,|\, [\tilde{y} \mapsto \tilde{v}]R_2 \\ S = \{\mathbf{com}_x\} \\ \Gamma \leq \Gamma', x : [\tilde{\tau}]/(O^{t_o}_{t_c}.U_1 \,|\, I^{t'_o}_{t'_c}.U_2) \end{array}$$

By Lemma B.4, we can assume without loss of generality that:

$$\Gamma = \Gamma', x : [\tilde{\tau}]/(O^{t_o}_{t_c}.U_1 \,|\, I^{t'_o}_{t'_c}.U_2)$$

By Lemma 3.2, there exists Δ' such that

$$\begin{array}{l} \Gamma' \leq \boxed{1}\Delta' \\ \Delta', x : [\tilde{\tau}]/(U_1 \,|\, U_2) \vdash_T Q \end{array}$$

So, by Lemmas B.4 and B.2, there exists Δ'' such that $\Gamma' \overset{\emptyset}{\Longrightarrow} \Delta''$ and $\Delta'' \leq \Delta'$. Let $\Delta = \Delta'', x : [\tilde{\tau}]/(U_1 \,|\, U_2)$. Then we have $\Delta \vdash_T Q$ and $\Gamma \overset{\{\mathbf{com}_x\}}{\Longrightarrow} \Delta$ as required.

- Case for (PR-ZERO): Trivial.
- Case for (PR-WAITO): It must be the case that

$$\begin{array}{l} P = \overline{x}\langle\tilde{v}\rangle^{t}.\,R \\ Q = \overline{x}\langle\tilde{v}\rangle^{t-1}.\,R \\ S = \{\overline{x}\} \\ \Gamma', x : [\tilde{\tau}]/U \vdash_T R \\ \Gamma \leq x : [\tilde{\tau}]/O^{t_o}_{t_c}.U, \boxed{t_c + 1}(\tilde{v} : \tilde{\tau} \,|\, \Gamma') \\ t_c \leq t \end{array}$$

By Lemma B.4, we can assume without loss of generality that:

$$\Gamma = x : [\tilde{\tau}]/O^{t_o}_{t_c}.U, \boxed{t_c + 1}(\tilde{v} : \tilde{\tau} \mid \Gamma')$$

and $0 < t_c$. Let $\Delta = x : [\tilde{\tau}]/O^{0}_{t_c-1}.U, \boxed{t_c}(\tilde{v} : \tilde{\tau} \mid \Gamma')$. Then by Lemma B.2, we have $\Gamma \overset{S}{\Longrightarrow} \Delta$. By the condition $t_c \leq t$, we have $t_c - 1 \leq t - 1$. So, by applying rule (BT-Out) to $\Gamma', x : [\tilde{\tau}]/U \vdash_T R$, we obtain $\Delta \vdash_T Q$, as required.

- Case for (PR-WaitI): Similar to the case above.
- Case for (PR-Par): It must be the case that

$$\begin{array}{ll} P = P_1 \mid P_2 & Q = Q_1 \mid Q_2 \\ P_1 \overset{S_1}{\Longrightarrow} P_2 & Q_1 \overset{S_2}{\Longrightarrow} Q_2 \\ S = S_1 \cup S_2 & S_1 \cap S_2 \cap \{\mathbf{com}_x \mid x \in \mathbf{Var}\} = \emptyset \\ \Gamma_i \vdash_T P_i & \Gamma \leq \Gamma_1 \mid \Gamma_2 \end{array}$$

By induction hypothesis, there exist Δ_1, Δ_2 such that $\Delta_i \vdash_T Q_i$ and $\Gamma_i \overset{S_i}{\Longrightarrow} \Delta_i$. (Notice that the second condition of the theorem cannot hold.) So, we have $(\Gamma_1 \mid \Gamma_2) \overset{S}{\Longrightarrow} (\Delta_1 \mid \Delta_2)$ by rule (TU-Par). By Lemma B.4, we have $\Delta \vdash_T Q$ and $\Gamma \overset{S}{\Longrightarrow} \Delta$ for some Δ.

- Case for (PR-Rep): It must be the case that

$$\begin{array}{ll} P = *P' & Q = *Q' \\ P' \overset{S}{\Longrightarrow} Q' & S \cap \{\mathbf{com}_x \mid x \in \mathbf{Var}\} = \emptyset \\ \Gamma' \vdash_T P' & \Gamma \leq *\Gamma' \end{array}$$

By induction hypothesis, there exists Δ' such that $\Gamma' \overset{S}{\Longrightarrow} \Delta'$ and $\Delta' \vdash_T Q'$. By rule (TU-Rep), we have $*\Gamma' \overset{S}{\Longrightarrow} *\Delta'$. So, we obtain the required result by using Lemma B.4.

- Case for (PR-IfT): It must be the case that

$$\begin{array}{ll} P = \mathbf{if}\ \mathit{true}\ \mathbf{then}\ Q\ \mathbf{else}\ R & S = \emptyset \\ \Gamma' \vdash_T Q & \Gamma \leq \boxed{1}\Gamma' \end{array}$$

By Lemmas B.4 and B.2, there exists Δ such that $\Delta \leq \Gamma'$ and $\Gamma \overset{\emptyset}{\Longrightarrow} \Delta$. By using (T-Weak), we obtain $\Delta \vdash_T Q$ as required.

- Case for (PR-IfF): Similar to the case above.
- Case for (PR-New): It must be the case that

$$\begin{array}{ll} P = (\nu x)\, P' & Q = (\nu x)\, Q' \\ P \overset{S'}{\longrightarrow} Q & S = S' \backslash \{\mathbf{com}_x, x, \overline{x}\} \\ \Gamma, x : [\tilde{\tau}]/U \vdash_T P' & \mathit{rel}_T(U) \end{array}$$

By induction hypothesis, one of the following conditions holds:

1. $\Delta, x : [\tilde{\tau}]/U' \vdash_T Q'$ and $(\Gamma, x : [\tilde{\tau}]/U) \overset{S'}{\Longrightarrow} (\Delta, x : [\tilde{\tau}]/U')$ for some Δ, U'.

2. For some $y \in dom(\Gamma)$, $y^\alpha \in S$, $\Gamma(y) = [\tilde{\tau_y}]/V$, and $V \cong \alpha_0^{t_o}.V_1 \,|\, V_2$.
3. $x^\alpha \in S$ and $U \cong \alpha_0^{t_o}.U_1 \,|\, U_2$.

In the second case, the required condition follows immediately. Next, we show that the third case cannot happen. Suppose that the third is the case. By the condition $rel_T(U)$, it must be the case that $U_2 \cong \overline{\alpha}_{t_c}^0.U_3 \,|\, U_4$ for some t_c, U_3 and U_4. Suppose $\alpha = I$ (the other case is similar). It must be the case that $P' \preceq (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle.\,P_1 \,|\, P_2)$, which implies $\mathbf{com}_x \in S$ or $\overline{x} \in S$. By the side condition of (PR-New), we have $\mathbf{com}_x \in S$ in both cases. So, it must be the case that $P' \preceq (\nu\tilde{w})\,(\overline{x}\langle\tilde{v}\rangle.\,P_1 \,|\, x(\tilde{y}).\,P_3 \,|\, x(\tilde{y}).\,P_4 \,|\, P_5)$. By the typing rules, there must exist U_5 and U_6 such that $U_2 \cong O_{t_c}^0.U_3 \,|\, I_{t_c'}^{t_o'}.U_5 \,|\, U_6$, which contradicts with the necessary condition $U_2 \not\longrightarrow$ of $rel_T(U)$.

In the first case, the required result ($\Delta \vdash_T Q$ and $\Gamma \overset{S}{\Longrightarrow} \Delta$) follows if we show $rel_T(U')$. By the condition $(\Gamma, x : [\tilde{\tau}]/U) \overset{S'}{\Longrightarrow} (\Delta, x : [\tilde{\tau}]/U')$, we have $U \overset{S'|_x}{\Longrightarrow} U'$. By the side condition of (PR-New), $\{x, \overline{x}\} \subseteq S'$ implies $\mathbf{com}_x \in S'$. So, $U \Longrightarrow U'$. By the definition of $rel_T(U)$, we have $rel_T(U')$.

- Case for (PR-SPCong): Trivial by Lemma A.6.

■

Proof of Theorem 4.2. This follows immediately from Lemma 4.2. Note that, by the same argument as the case for (PR-New) in the proof of Lemma 4.2, the second case of Lemma 4.2 cannot happen if $rel_T(\Gamma)$ holds. ■