A Partially Deadlock-Free Typed Process Calculus

NAOKI KOBAYASHI
University of Tokyo

We propose a novel static type system for a process calculus, which ensures both partial deadlock-freedom and partial confluence. The key novel ideas are (1) introduction of the order of channel use as type information, and (2) classification of communication channels into reliable and unreliable channels based on their usage and a guarantee of the usage by the type system. We can ensure that communication on reliable channels never causes deadlock and also that certain reliable channels never introduce nondeterminism. After presenting the type system and formal proofs of its correctness, we show encodings of the λ-calculus and typical concurrent objects in the deadlock-free fragment of the calculus and demonstrate how type information can be used for reasoning about program behavior.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages

Additional Key Words and Phrases: Concurrency, deadlock-freedom, type theory

1. INTRODUCTION

1.1 Backgrounds and Problems

Various concurrent programming languages [Kobayashi 1996a; Peyton Jones 1996; Pierce and Turner 1997; Reppy 1991; Shapiro 1989; Yonezawa and Tokoro 1987] have recently been proposed and are attracting a great deal of attention because of their usefulness in describing symbolic/numeric computations for parallel or distributed environments and inherently concurrent applications such as GUIs. Many of them share some common primitives for concurrent computation—creations of first-class communication channels (which we here call *channels* for short), communication over those channels, and concurrent execution of processes—which have recently been studied extensively through process calculi [Honda and Yoshida 1995; Kobayashi et al. 1996; Milner et al. 1992; Pierce and Sangiorgi 1993; Sangiorgi 1992]. They have been found to be expressive enough to provide higher-level mechanisms such as concurrent objects [Kobayashi and Yonezawa 1995; Pierce and Turner 1995; Walker 1995].

However, such concurrency primitives are too low-level, and they suffer from

An extended and revised version of the paper presented at 12th IEEE Symposium on Logic in Computer Science (LICS'97).
Author's address: Department of Information Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan; email:koba@is.s.u-tokyo.ac.jp.

serious problems: deadlock and nondeterminism. They often make it difficult to debug programs and to think about program behavior. For example, if we extend a functional programming language with concurrency primitives [Reppy 1991], a term of a function type may not behave like a function: it may return nondeterministic results, or may be blocked forever without returning the result. Deadlock and nondeterminism also make static decision of control flows difficult, disabling efficient implementation of concurrency primitives.

1.2 Our Approach

The goal of this article is to overcome the above problems by identifying deadlock-free and/or deterministic parts of concurrent programs with a static type system. The important observation behind this solution is that not all parts of concurrent programs suffer from deadlock and nondeterminism: most parts of actual programs should be deadlock-free and/or deterministic. The key ideas in our approach (especially for ensuring partial deadlock-freedom) are (1) to introduce the order of channel use as type information, and (2) to classify channels according to their usage and to enforce the usage by a type system.

To illustrate these ideas, we consider the following asynchronous process calculus (which is a subset of the calculus we consider in this article):

$$
\begin{array}{rrl}
P ::= & P_1 \mid P_2 & \text{(executes } P_1 \text{ and } P_2 \text{ concurrently)} \\
& x![\tilde{y}] & \text{(sends } \tilde{y} \text{ along the channel } x) \\
& x?[\tilde{z}].\, P & \text{(receives values } \tilde{v} \text{ along } x \text{ and behaves like } [\tilde{v}/\tilde{z}]P) \\
& x?^*[\tilde{z}].\, P & \text{(repeatedly receives values } \tilde{v} \text{ along } x \\
& & \quad \text{and spawns the process } [\tilde{v}/\tilde{z}]P) \\
& (\nu x)\, P & \text{(creates a channel } x \text{ and executes } P) \\
& \mathbf{0} & \text{(becomes silent)}
\end{array}
$$

Here $\tilde{y}$ abbreviates a sequence $y_1, \ldots, y_n$. As in the π-calculus [Milner 1993], channels are first-class values in the sense that they can be dynamically created and passed as data along other channels. Note that $x?^*[\tilde{z}].\, P$ behaves like a (recursive) process definition $x![\tilde{z}] = P$ since the process $x?^*[\tilde{z}].\, P \mid x![\tilde{v}]$ is reduced to $x?^*[\tilde{z}].\, P \mid [\tilde{v}/\tilde{z}]P$.

Let us write $\updownarrow[\tilde{T}]$ for the type of channels used for sending/receiving a sequence of values of types $\tilde{T}$. In earlier type systems for process calculi or concurrent languages [Gay 1993; Reppy 1991; Vasconcelos and Honda 1993], $x?[\,].\, y![\,]$ is typed as

$$x : \updownarrow[\,], y : \updownarrow[\,] \vdash x?[\,].\, y![\,].$$

The first key idea of the present work is to attach a time tag to each instance of a channel type constructor and keep ordering between them. So in our type system, the above type judgment is replaced by

$$x : \updownarrow[\,]^{t_x}, y : \updownarrow[\,]^{t_y}, \{(t_x, t_y)\} \vdash x?[\,].\, y![\,],$$

where the relation $\{(t_x, t_y)\}$ expresses the fact that a channel x may be used before y (a more correct intuition is: "communication on y may be delayed until that on x is completed"). On the other hand, a process $y?[\,].\, x![\,]$ is typed as

$$x : \updownarrow[\,]^{t_x}, y : \updownarrow[\,]^{t_y}, \{(t_y, t_x)\} \vdash y?[\,].\, x![\,],$$

where $\{(t_y, t_x)\}$ expresses the fact that a channel y may be used before x (or "communication on x may be delayed until that on y is completed"). Thus, if we make the parallel composition $x?[\,].\,y![\,] \mid y?[\,].\,x![\,]$, we detect a cyclic ordering $\{(t_x, t_y), (t_y, t_x)\}$ which indicates that the process may fall into deadlock because of bad use of the channels x and y. Let us consider the case where channels are passed as first-class values. $x : \updownarrow[\updownarrow[\,]^{s_y}, \updownarrow[\,]^{s_z}]^{t_x}$ with the relation $\{(s_y, s_z)\}$ means that x is a channel to send/receive two channels y, z such that (1) a receiver from x cannot use y, z in an order unspecified by $\{(s_y, s_z)\}$ (i.e., it is not allowed to delay communication on y until that on z is completed), and (2) channels sent along x must be those that can be used in an order specified by $\{(s_y, s_z)\}$. So the processes $x?[y, z].\,(y?[\,].\,z![\,])$, $x?[y, z].\,(y?[\,].\,\mathbf{0} \mid z![\,])$ and $x![u, v]$ are well-typed under $x : \updownarrow[\updownarrow[\,]^{s_y}, \updownarrow[\,]^{s_z}]^{t_x}$ with $\{(s_y, s_z)\}$, but neither $x?[y, z].\,(z?[\,].\,y![\,])$ nor $x![u, v] \mid v?[\,].\,u![\,]$ is. ($x![u, v] \mid v?[\,].\,u![\,]$ is not allowed because a receiver of $x![u, v]$ cannot use u before v as $v?[\,].\,u![\,]$ uses v before u.) In general, a type judgment is written in the form $x_1 : T_1, \ldots, x_n : T_n, \prec \vdash P$, where $T_1, \ldots, T_n$ are types annotated with time tags and $\prec$ is a binary relation on time tags. To avoid deadlock, we require that the transitive closure of $\prec$ should be a strict partial order.

The second key idea of the present work is to classify channels. Note that introduction of ordering information is itself not able to ensure freedom from deadlock completely. For example, although the process $(\nu x)\,(x?[\,].\,y?[\,].\,\mathbf{0}) \mid y![\,]$ does not have any cyclic dependency on channel use, it is blocked forever. In order to reject such a process, we need to require, in addition to the lack of cyclic dependency, that every process communicating over a certain channel can eventually find its communication partner. We consider three kinds of such channels (which we call *reliable channels*) and ensure the correct usage of each channel by using a static type system. For other kinds of channels (*unreliable channels*), we give up ensuring deadlock-freedom.

By combining the above ideas, we can ensure that use of reliable channels does not cause deadlock. That is, (1) if only reliable channels are used in a process, deadlock never occurs, and (2) deadlock may occur if a process also uses unreliable channels, but only if there is bad use of the unreliable channels (when a process deadlocks, there is some subprocess trying to communicate over an unreliable channel). As will be demonstrated later, the lambda-calculus and typical concurrent objects can be encoded by using only reliable channels. From this observation and brief profiling of several concurrent programs for window systems and numerical computation, we expect that large parts of many concurrent programs can be written with reliable channels and that our type system helps programmers to pinpoint the source of deadlock in such programs.

1.3 Main Results

The main results of this work are (1) the formulation of typing rules to guarantee the channel usage and order of channel use mentioned above, and (2) formal proofs of the above claim that reliable channels never cause deadlock in a well-typed process.

In order to show the expressive power of reliable channels and demonstrate how type information can be used for reasoning on the program behavior, we show that every well-typed term of call-by-value, call-by-name, and call-by-need simply typed λ-calculi is encoded into a well-typed process of our process calculus without using

unreliable channels, and we use the theorem on partial deadlock-freedom for inferring that the evaluation of those processes never gets stuck. Moreover, in the cases of the call-by-value and call-by-need lambda-calculi, type information is sufficient for us to infer that the result of the evaluation is unique (up to some typed process congruence [Kobayashi et al. 1996]). These kinds of reasoning have previously been possible only by looking at the code in detail. We also show (informally) that typical concurrent objects are also implemented by using reliable channels.

In addition, it would be worth mentioning that there exists a polynomial time type check algorithm which, given $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ and P, decides whether or not there exists $\prec$ such that $\Gamma, \prec \vdash P$ (this implies that a programmer need not annotate a program with ordering information).

1.4 Other Applications

In addition to the static detection of deadlock, there are several interesting potential applications outside the scope of this article. Because the ordering $\prec$ in a judgment $x_1 : T_1, \ldots, x_n : T_n, \prec \vdash P$ expresses the order in which P uses channels for communicating with external processes, our type system gives a better specification of processes than do previous type systems, which typically only specify which type of values are transmitted via each channel. Therefore, our type system would be useful for improving module systems of concurrent programming languages although a lot of work is still left to be done for this purpose (for example, we need to make it much easier for programmers to write and read types). It would also be useful for efficient implementation of concurrency primitives: for example, we expect it can be used for eliminating run-time check of the channel status. The latter issue is discussed in Section 9 in more detail.

1.5 Advantages of Our Approach

1.5.1 *Why Not Just Introduce High-Level Primitives?* An alternative, more obvious approach to avoiding the problems of deadlock and nondeterminism would be to provide higher-level constructs as primitives, so that programmers can be assured that a program is deadlock-free and/or deterministic as long as those constructs are used. However, we think that it is hard to provide a set of high-level constructs that is satisfactory for any application programs in any parallel/distributed environments; even if it is possible, the set of primitives would become quite large: as a result, other analyses and implementation would become very complicated, and addition or modification of high-level constructs would be difficult. Consider concurrent objects, for example: we need various kinds of concurrent objects—from simple ones to complex ones supporting transactions, intraobject concurrency, etc. In parallel/distributed environments, many primitives (for parallel evaluation, transmission of data, etc.) seem necessary even for functional programs, unless one expects that it is possible to realize automatic, optimal parallelization and data distribution.

In contrast, although we are not opposed to adding some basic constructs for functions and concurrent objects as primitives, our approach in this article is to let programmers define high-level constructs as derived forms, and still to provide a way for reasoning about the behavior of those constructs in terms of a static type system. Such an approach has been exploited in other recent work that tries

to design and implement a concurrent programming language based on solid theoretical foundations of process calculi [Igarashi and Kobayashi 1997; Kobayashi 1996a; Kobayashi and Yonezawa 1995; Oyama et al. 1997; Pierce and Sangiorgi 1997; Pierce and Turner 1997; Turner 1996]. Advantages of the approach should be clear: other analyses and implementation are simplified, and programmers can safely define and use application-specific high-level constructs. From encodings of various evaluation strategies for the λ-calculus and concurrent objects given later, we think that our type system works well at least for a class of concurrent programs in which concurrency primitives are used mainly for controlling parallelism and data distribution, and also for simple reactive programs. More studies will be necessary before we can conclude that our approach is reasonable also for complex reactive programs.

1.5.2 *Why Type Systems?* Our type system presented here may look rather complex. So one might wonder if the same thing can be done more easily by using other methods, such as abstract interpretation. However, using a type system seems preferable in many points. First, designing a suitable abstract domain is not easier than designing our type system: for the purpose of analyzing deadlock, one cannot simply map multiple communication channels to the same abstract channel, as discussed in Section 8. So, an abstract process would be more or less what we represent by a type environment and what abstract interpretation does would be similar to what we do in type checking. Second, type systems seem more suitable for compositional reasoning: the reason is that types can express the assumption about the behavior of the environment as well as the behavior of a process itself, so that two processes can be composed as long as they satisfy each other's assumption about the environment. Third, with type systems, correctness and complexity of the analysis can be discussed in a systematic and standard manner: in our type system, for example, correctness can be ensured by the subject reduction property and properties that immediately follow from typing rules. Furthermore, a type system provides more than a means for automatic analysis; it can also allow a programmer to specify the intended behavior of a program.

1.6 Structure of the Article

The rest of this article is structured as follows. Section 2 explains basic notations. Section 3 gives our type system and Section 4 shows that a well-typed process satisfies a partial deadlock-freedom property. Section 5 informally presents a polynomial-time type check algorithm for our type system. In order to show the expressive power of our type system and demonstrate how type information can be used for reasoning about program behavior, Section 6 shows encodings of functions and concurrent objects into the deadlock-free fragment of our process calculus. Section 7 discusses other remaining issues and Section 8 discusses related work. Section 9 concludes this article.

2. NOTATIONAL PRELIMINARIES

This section introduces the syntax of types, expressions, and type judgments. If types are ignored, our process calculus can be considered a subset of the polyadic π-calculus [Milner 1993]. We dropped synchronous outputs, summations (choices),

and matchings from the π-calculus, and we restricted replications to be only performed to input prefixes. In the course of introducing the notations, we also sharpen the intuitions given in Section 1.

2.1 Types

For simplicity in this article, we consider channel types and base types as primary type constructors. $p^m[T_1, \ldots, T_n]^t$ (abbreviated to $p^m[\tilde{T}]^t$) represents the type of channels which carry a sequence of values of types $T_1, \ldots, T_n$. Extra annotations p, called a *polarity*, m, called a *multiplicity* or *mode*, and t, called a *time tag*, are explained as follows.

2.1.1 *Multiplicities.* A multiplicity m ranges over $\{1, *, M, \omega\}$. It represents how channels can be used:

—If $m = 1$, types are called *linear channel types* and channels *linear channels* [Kobayashi et al. 1996]. A linear channel can be used just once for communication. More precisely, it can be used once for an output (send) operation and once for an input (receive) operation.

—If $m = *$, types are called *replicated input channel types* and channels *replicated input channels* or simply *replicated channels*. A replicated input channel x can be used just once in a replicated input prefix $(x?^*[\tilde{y}].\,)$, and an arbitrary number of times in output expressions $(x![\tilde{y}])$, but cannot be used in any input prefix $(x?[\tilde{y}].\,)$. It is typically used in a process of the form $(\nu x)\,(x?^*[\tilde{y}].\,P \mid Q)$, which creates a replicated input channel x and executes two processes $x?^*[\tilde{y}].\,P$ and Q. Here x cannot be used for any input operations in P or Q. So it can be considered a process that executes Q under the process definition $x![y] = P$.

—If $m = M$, types are called *mutex channel types* and channels *mutex channels*. A single message must be put into a mutex channel after its creation. Afterward, the channel can be used an arbitrary number of times for input, as long as a process that has extracted a value from it eventually puts a value into it. A mutex channel can, therefore, be considered a binary semaphore.

—If $m = \omega$, types are called *unreliable channel types* and channels *unreliable channels*. We have no particular restriction on the use of unreliable channels, except that they cannot be used in replicated input prefixes.

A channel is also called *reliable* if it is a linear, replicated input, or mutex channel. The reason why we call them "reliable" is that, with our type system, input/output operations on linear channels, output operations on replicated input channels, and input operations on mutex channels can eventually proceed, unless the whole program falls into an infinite loop or a deadlock caused by bad use of unreliable channels.

2.1.2 *Polarities.* A polarity p, which ranges over subsets of the set $\{I, O\}$ of input/output capabilities, controls whether a channel is used for input or output. We write $\updownarrow$ for $\{I, O\}$, ? for $\{I\}$, ! for $\{O\}$, and | for the empty set $\emptyset$. For example, a type binding $x : \updownarrow^{\omega}[int]^t$ means that x can be used both for input and output an arbitrary number of times, and $x : ?^1[int]^t$ means that x can be used just once for input. The polarity of a mutex channel type $p^M[T]^t$ has a slightly special meaning:

If $p = ?$, a channel can be used for receiving a value of type T, and after that, the process has the obligation to send some value back to the channel.

The above usage of channels would be better understood if we consider that types represent both *capabilities* and *obligations*. For example, if a process holds a linear channel of type $?^1[\tilde{T}]$, then it has the capability to receive a value once from the channel, and at the same time, the obligation to do so. If a process holds a mutex channel of type $\updownarrow^M[\tilde{T}]$, then it has the capability to send/receive a value along the channel (the send operation is allowed only once), but it only has an obligation to send a value: it need not receive a value from the channel. Similarly, if a process holds a replicated input channel of type $\updownarrow^*[\tilde{T}]$, it has the capability to send/receive a value along the channel, but the obligation is just to (be ready to) receive a value (via replicated input prefix). In the case of an unreliable channel, a process has capabilities but no obligation.

2.1.3 *Time Tags.* A time tag $s(, t, u, \ldots)$ intuitively describes *when* channels should be used. As indicated in Section 1, we will later associate time tags with a binary relation $\prec$. $t_1 \prec t_2$ means "a process may be blocked on the channel tagged with t_1 *before* it fulfills an *obligation* to communicate over the channel tagged with t_2." So if a process has a channel x of type $?^1[int]^{t_1}$ and y of type $?^1[int]^{t_2}$ such that $t_1 \prec t_2$, the process can behave like $x?[m].\, y?[n].\, \mathbf{0}$ but cannot behave like $y?[n].\, x?[m].\, \mathbf{0}$. As another example, suppose t_x is the time tag of a mutex channel x and t_y is the time tag of another channel y, then

—If y is a linear channel, $t_x \prec t_y$ implies that a process may try to acquire a lock x (i.e., wait for a value on x) before communicating over y. For example, the process $x?[\,].\,(y![\,] \mid x![\,])$ is allowed only if there is an ordering $t_x \prec t_y$.

—$t_y \prec t_x$ implies that a process may try to communicate over y before releasing a lock x (i.e., sending a value along x). For example, if y is also a mutex channel, a process can behave like $x?[\,].\, y?[\,].\,(x![\,] \mid y![\,])$ only if $t_y \prec t_x$ holds, since the process tries to communicate over the channel y before fulfilling the obligation to send a value to the channel x. Note that, in this case, the ordering $t_x \prec t_y$ is unnecessary because receiving a value from the mutex channel y is not an obligation.

Since we cannot keep an arbitrary ordering on an infinite set of time tags, many actual channels may be mapped to the same tag in our type system. For example, consider the type $\updownarrow^\omega[!^1[\,]^{s_y}, !^1[\,]^{s_z}]^\star$: s_y and s_z correspond to any channels sent along a channel of that type.

The set of time tags, written $\mathbf{T}_\star$, includes a special tag $\star$. $\mathbf{T}$ denotes $\mathbf{T}_\star \backslash \{\star\}$. We always annotate replicated input channel types and unreliable channel types with the special tag $\star$, in order to ignore when channels of those types are used.

We could keep orderings on the use of replicated input channels in the same manner as we do on the use of linear and mutex channels. We, however, do not take that approach in this article because it would damage the expressive power of the deadlock-free fragment of the calculus.[1] Instead, we take a more trivial

[1]This is for a rather subtle technical reason coming from the side condition of rule (T-NEW-1) in Section 3. It might be possible to keep ordering on replicated input channels by relaxing the condition.

approach to deadlock-freedom: we ensure (by typing rules and the definition of $\prec^\sharp$ given later) that the obligation on a replicated input channel must be fulfilled immediately after its creation; in other words, a replicated input channel must be used for input (in a replicated input prefix) immediately after its creation.

In order to simplify later arguments, we assume that the set $\mathbf{T}$ can be divided into two disjoint sets $\mathbf{T_O}$ and $\mathbf{T_I}$: we will use an element of $\mathbf{T_O}$ for the outermost time tag of a type and use an element of $\mathbf{T_I}$ for the other time tags.

2.1.4 *Types.* Now we are ready to formally define types. We use the word *bare type* for a type whose outermost time tag is dropped.

Definition 2.1.4.1. The sets of bare types and types are given by the following syntax, with the additional well-formedness conditions described below:

$$\begin{array}{lll} \rho \text{ (bare types)} & ::= & b \text{ (base types)} \mid p^m[\tilde{T}] \\ T, U \text{ (types)} & ::= & \rho^t \end{array}$$

Here $p^m[\tilde{T}]$ abbreviates $p^m[T_1, \ldots, T_n]$. The additional conditions are given as follows:

(1) t in ρ^t is $\star$ if and only if ρ is a base type b, an unreliable channel type $p^\omega[\tilde{T}]$, or a replicated input channel type $p^*[\tilde{T}]$.

(2) If a type contains a bare type of the form: $p^M[\rho_1^{t_1}, \ldots, \rho_n^{t_n}]$, then for each $i \in \{1, \ldots, n\}$, one of the following conditions holds:
—ρ_i is a base type, an unreliable channel type, or a replicated input channel type;
—$\rho_i = q^M[\tilde{T}]$ and $O \notin q$; or
—$\rho_i = q^1[\tilde{T}]$, and q is neither ! nor ?.
(i.e., output-only/input-only linear channels and output ends of mutex channels cannot be transmitted along mutex channels).

We assume that the set of base types contains "*bool*."

The second additional condition means that a channel containing some obligation cannot be put into a mutex channel. The reason for this is that there is no guarantee by our type system that the value in a mutex channel will eventually be extracted: since a mutex channel is used as a reference cell extended with a mutual exclusion mechanism, we want to ensure that a receiver is not blocked forever, but we do allow a sender to be blocked forever. For example, let x be a mutex channel to send/receive the output end of a linear channel, and let y be a linear channel; if we removed the restriction, $x![y] \mid y?[\,].\,\mathbf{0}$ would be well typed, but are in deadlock (if there is no receiver on x in parallel to this process).

Notation 2.1.4.2. When the time tag is not important (especially when it is $\star$), we sometimes just write ρ for ρ^t. We also often write $\tilde{\rho}^{\tilde{s}}$ for $\rho_1^{s_1}, \ldots, \rho_n^{s_n}$.

Terminology and operations on types are introduced below. Because we must be very sensitive to how often each channel is used, it is convenient to classify types into *unlimited types* and nonunlimited types, according to whether or not a value of the type can appear in an unlimited number of places. Whether or not a type is unlimited depends on its polarity and multiplicity: for example, the output end of

a replicated input channel can be used an arbitrary number of times, but the input end cannot.

Definition 2.1.4.3. A type T is *unlimited* if (1) T is a base type or an unreliable channel type, (2) $T = p^*[\tilde{T}]^\star$ and $I \notin p$, (3) $T = p^M[\tilde{T}]^t$ and $O \notin p$, or (4) $T = |^1[\tilde{T}]^t$.

Consider a process $P \mid Q$. If P uses x as a channel of type $!^1[int]$ (i.e., P uses x just once for output) and Q uses x as a channel of type $?^1[int]$, then x is totally used once for output and once for input in $P \mid Q$, thus x should have a type $\updownarrow^1[int]$ in $P \mid Q$. The operator $+$ defined below captures this: the type of a variable in a process is calculated by combining the types of the variable in its subprocesses with $+$. In the example above, $!^1[int] + ?^1[int]$ can be defined as $\updownarrow^1[int]$.

Definition 2.1.4.4. The *combination* of two bare types ρ_1 and ρ_2, written $\rho_1 + \rho_2$, is defined by

$$\begin{aligned} b + b &= b \\ p^m[\tilde{T}] + q^m[\tilde{T}] &= (p \cup q)^m[\tilde{T}] \text{ if } (p \cap q)^m[\tilde{T}] \text{ is an unlimited type.} \end{aligned}$$

For the other cases, $\rho_1 + \rho_2$ is undefined. The combination of two types, written $\rho_1^{t_1} + \rho_2^{t_2}$, is defined as $(\rho_1 + \rho_2)^{t_1}$ only if $\rho_1 + \rho_2$ is well defined and $t_1 = t_2$.

Definition 2.1.4.5. Let T be a type. The type $Rem(T)$ is defined by

$$Rem(T) = \begin{cases} |^1[\tilde{S}]^t & \text{if } T = p^1[\tilde{S}]^t \\ T & \text{otherwise.} \end{cases}$$

2.2 Process Expressions

Here we add some annotations and conditional expressions to the syntax used in Section 1. The process **0** in Section 1 will be defined as a derived form below.

Definition 2.2.1. The set of process expressions is given by the following syntax:

$$\begin{aligned} P ::= \quad & P_1 \mid P_2 && \text{(concurrent composition)} \\ & x![\tilde{y}] && \text{(output expressions)} \\ & x?[\tilde{z}].\, P && \text{(input expressions)} \\ & x?^*[\tilde{z}].\, P && \text{(replicated input expressions)} \\ & (\nu x : T)\, P && \text{(channel creation)} \\ & \mathit{if}\ x\ \mathit{then}\ P_1\ \mathit{else}\ P_2 && \text{(conditional expression)} \end{aligned}$$

where $\tilde{y}$ abbreviates a sequence $y_1, \ldots, y_n$ (n may be 0). In an expression $(\nu x : \rho^s)\, P$, s must be an element of $\mathbf{T_O} \cup \{\star\}$, and all the time tags in ρ must be elements of $\mathbf{T_I} \cup \{\star\}$.

As usual, we assume that $\tilde{z}$ are bound by prefixes $x?[\tilde{z}]$ and $x?^*[\tilde{z}]$ and that x is bound by $(\nu x : T)$. We identify expressions up to the renaming of bound variables (i.e., α-conversion): for example, we do not distinguish between $(\nu x : T)\, x![\,]$ and $(\nu y : T)\, y![\,]$. We give prefixes $(x?[\tilde{z}], x?^*[\tilde{z}], (\nu x : T))$ a higher precedence than concurrent composition $(\mid)$, so that $x?[\,].(\nu y : T)\, x![\,] \mid y![\,]$ means $(x?[\,].((\nu y : T)\, x![\,])) \mid y![\,]$. We often write $(\nu \tilde{x}{:}\tilde{T})\, P$ for $(\nu x_1 : T_1) \cdots (\nu x_n : T_n)\, P$ (n may be 0), and **0** for $(\nu x : \updownarrow^M[\,]^t)\, (x![\,])$ where t is a fresh time tag.

We write $New(P)$ for the set of time tags t of ν-prefixes $(\nu x : \rho^t)$ in P. It is formally defined by

$$\begin{aligned} New(P_1 \mid P_2) &= New(P_1) \cup New(P_2) \\ New(x![\tilde{y}]) &= \emptyset \\ New(x?[\tilde{y}].\, P) &= New(P) \\ New(x?^*[\tilde{y}].\, P) &= New(P) \\ New((\nu x : \rho^s)\, P) &= (\{s\} \backslash \{\star\}) \cup New(P) \\ New(\mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2) &= New(P_1) \cup New(P_2). \end{aligned}$$

Examples

In order to help understand the syntax, we give examples below of process expressions and show how they are reduced. A formal definition of the operational semantics, which is based on the standard reduction relation [Milner 1993], is deferred until Section 4.

Example 2.2.2. The process $x![y_1, y_2] \mid x?[z_1, z_2].\, z_1![z_2] \mid y_1?[w].\, w![\ ]$ can be reduced to $y_1![y_2] \mid y_1?[w].\, w![\ ]$ by communication on x. Then it is reduced to $y_2![\ ]$ by communication on y_1.

Example 2.2.3. The process $x![y_1, y_2] \mid x?^*[z_1, z_2].\, z_1![z_2]$ can be reduced to the process $y_1![y_2] \mid x?^*[z_1, z_2].\, z_1![z_2]$.

Example 2.2.4. The process

$$\begin{aligned} P = fact?^*[n, r].\, (&\mathit{if}\ n = 0\ \mathit{then}\ r![1] \\ &\mathit{else}\ (\nu r' : \updownarrow^1[int]^t)\, (fact![n-1, r'] \mid r'?[k].\, r![k \times n])) \end{aligned}$$

can be viewed as a definition of the factorial function. (For simplicity, we assume that we have $\times, =$ on integers as primitives in this example.) It receives an integer n and a linear channel r along the replicated input channel $fact$, and sends the factorial of n to r. For example, $P \mid fact![2, y]$ is reduced as follows:

$$\begin{aligned} P \mid fact![2, y] &\longrightarrow P \mid (\nu r')\, (fact![1, r'] \mid r'?[k].\, y![k \times 2]) \\ &\longrightarrow (\nu r')\, (P \mid (\nu r'')\, (fact![0, r''] \mid r''?[l].\, r'![l \times 1]) \\ &\qquad\qquad \mid r'?[k].\, y![k \times 2]) \\ &\longrightarrow (\nu r')\, (\nu r'')\, (P \mid r''![1] \mid r''?[l].\, r'![l \times 1] \mid r'?[k].\, y![k \times 2]) \\ &\longrightarrow (\nu r')\, (\nu r'')\, (P \mid r'![1] \mid r'?[k].\, y![k \times 2]) \\ &\longrightarrow (\nu r')\, (\nu r'')\, (P \mid y![2]) \end{aligned}$$

Example 2.2.5. The following process, which represents a counter object, shows typical use of a mutex channel.

$$\begin{aligned} &(\nu st : \updownarrow^M[int]^t) \\ &\quad (st![1] \\ &\quad \mid inc?^*[r].\, st?[n].\, (r![\ ] \mid st![n+1]) \\ &\quad \mid read?^*[r].\, st?[n].\, (r![n] \mid st![n])). \end{aligned}$$

The mutex channel st is used for keeping the internal state of the object, so the process $st![1]$ expresses that the current counter value is 1. The processes $inc?^*[r].\, \cdots$ and $read?^*[r].\, \cdots$ can be viewed as definitions of methods for incrementing and

reading the current value respectively: if a message $inc![r]$ arrives, the process $inc?^*[r].\cdots$ receives it, extracts the current value from the mutex channel st, sends a null tuple to the reply channel r, and puts the incremented value into st. The assumption of st being a mutex channel avoids easy mistakes such as forgetting to write back a new state and duplicating a state.

2.3 Type Judgment Form

We introduce type environments, several operations on them, and the form of type judgments.

2.3.1 *Type Environments.* A type environment is a mapping from a finite set of variables to types. If *true* (*false*, resp.) is in the domain of a type environment Γ, then $\Gamma(true)$ ($\Gamma(false)$, resp.) must be *bool*. We also assume that if $\Gamma(x) = \rho^s$, then s is an element of $\mathbf{T_O} \cup \{\star\}$ and all the time tags in ρ are elements of $\mathbf{T_I} \cup \{\star\}$.

Notation 2.3.1.1. We often use metavariables Γ, Δ for type environments. The domain of Γ is denoted by $dom(\Gamma)$. We write $x_1 : T_1, \ldots, x_n : T_n$ (or $\tilde{x} : \tilde{T}$ in an abbreviated form) for the type environment Γ such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$, $\Gamma(x_i) = T_i$ for each $i \in \{1, \ldots, n\}$. When $dom(\Gamma) \cap \{\tilde{x}\} = \emptyset$, we write $\Gamma, \tilde{x} : \tilde{T}$ for the type environment Δ such that $dom(\Delta) = dom(\Gamma) \cup \{\tilde{x}\}$, $\Delta(\tilde{x}) = \tilde{T}$ and $\Delta(y) = \Gamma(y)$ for $y \in dom(\Gamma)$. We also write $\emptyset$ for the type environment whose domain is empty.

We say Γ is *unlimited* if $\Gamma(x)$ is unlimited for each $x \in dom(\Gamma)$. The operation $+$ is extended to type environments as follows:

Definition 2.3.1.2. Let Γ_1 and Γ_2 be type environments. We define $\Gamma_1 + \Gamma_2$ by

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

Trivially, $+$ is associative and commutative. We sometimes write $\Sigma\{\Gamma_1, \ldots, \Gamma_n\}$ for $\Gamma_1 + \cdots + \Gamma_n$.

Example 2.3.1.3. $(x : ?^1[int]^t, y : !^1[int]^s) + (x : !^1[int]^t, y : ?^1[int]^s) = x : \updownarrow^1[int]^t, y : \updownarrow^1[int]^s$.

Example 2.3.1.4. $(x : !^*[int]^\star, y : ?^M[int]^s) + (x : \updownarrow^*[int]^\star, y : \updownarrow^M[int]^s) = x : \updownarrow^*[int]^\star, y : \updownarrow^M[int]^s$.

2.3.2 *Tag Ordering.* A *tag ordering* $\prec \subseteq \mathbf{T} \times \mathbf{T}$ is a binary relation on time tags such that its transitive closure, written $\prec^+$, is a strict partial order (i.e., a relation $\mathcal{R}$ that satisfies transitivity and irreflexivity: $\forall t.\neg(t\mathcal{R}t)$). Note that we do not require $\prec$ itself to be a strict partial order.[2] We often regard a tag ordering

[2] This follows intuitively because we want to distinguish between *direct* dependency (e.g., the dependency between x and y in $x?[].y![]$) and *indirect* dependency (e.g., the one between x and y in $x?[].z![] \mid z?[].y![]$) of channel usage. This distinction is especially important in the rule (T-NEW-1) in Section 3.

$\prec$ ($\subseteq \mathbf{T} \times \mathbf{T}$) as the binary relation $\mathcal{R}$ on the set $\mathbf{T}_\star$ defined by: $t_1 \mathcal{R} t_2 \Leftrightarrow (t_1 = \star \vee t_2 = \star \vee (t_1 \in \mathbf{T} \wedge t_2 \in \mathbf{T} \wedge t_1 \prec t_2))$.

Let $\prec$ be a tag ordering. We extend $\prec$ to a relation between time tags and type environments. Intuitively, $t \prec^\sharp \Gamma$ means that a process may be blocked on the channel tagged with t before fulfilling the obligations specified by Γ. Therefore, $t \prec^\sharp x : T$ always holds if T represents no obligation. In particular, $t \prec^\sharp x : T$ always holds if T is an unlimited type since it cannot represent any obligation (because a value of the unlimited type may not be used at all).

Definition 2.3.2.1. Let t be a time tag, Γ be a type environment, and $\prec$ be a tag ordering. We define $t \prec^\sharp \Gamma$ by

$$
\begin{array}{lll}
t \prec^\sharp \emptyset & & \\
t \prec^\sharp x : b^\star & & \\
t \prec^\sharp x : p^m[\tilde{T}]^s & \text{iff} & (t \prec s \wedge m \neq *) \\
 & & \vee (p^m[\tilde{T}]^s \text{ is unlimited}) \\
 & & \vee (m = 1 \wedge p = \updownarrow) \\
t \prec^\sharp \Gamma, x : T & \text{if} & (t \prec^\sharp \Gamma) \wedge (t \prec^\sharp x : T).
\end{array}
$$

Careful readers may think that we should have required $t \prec s$ for $t \prec^\sharp \updownarrow^1[\tilde{T}]^s$ to hold. We do not require it, however, because deadlock-freedom is not affected by a process being blocked on a channel specified by t before performing both send and receive operations on a linear channel. This can be understood by considering that $\updownarrow^1[\tilde{T}]$ currently represents no obligation and that the obligation to receive (send, resp.) a value along a channel of type $\updownarrow^1[\tilde{T}]^s$ arises only when the capability to send (receive, resp.) a value is used. Note also that, if a type T contains the obligation to receive a value from a replicated input channel, $t \prec^\sharp x : T$ never holds regardless of the relation $\prec$. This means that a process must never try to communicate over other channels before using the input end of a replicated input channel.

We introduce several notations on tag orderings.

Definition 2.3.2.2. Let $\prec \subseteq \mathbf{T} \times \mathbf{T}$ and $S_1, S_2 \subseteq \mathbf{T}_\star$. We define $S_1 \not\prec\!\!\not\succ S_2$ by

$$\forall s \in S_1. \forall t \in S_2. (s = \star \vee t = \star \vee \{(s,t),(t,s)\} \cap \prec = \emptyset).$$

When $\prec$ is subscripted with $i, j, k, \ldots$, we write $\not\prec_i\!\!\not\succ$, $\not\prec_j\!\!\not\succ$, $\not\prec_k\!\!\not\succ$, etc.

Definition 2.3.2.3. Let $\mathcal{R} (\subseteq \mathbf{T} \times \mathbf{T})$ be a relation on time tags and $S (\subseteq \mathbf{T}_\star)$ be a set of time tags. We define $\mathcal{R} \downarrow_S, \mathcal{R} \uparrow_S, \mathcal{R} \Downarrow_S$ by

$$
\begin{array}{rcl}
\mathcal{R} \downarrow_S & = & \mathcal{R} \cap (\mathbf{T} \backslash S \times \mathbf{T} \backslash S) (= \mathcal{R} \backslash ((S \times \mathbf{T}) \cup (\mathbf{T} \times S))) \\
\mathcal{R} \uparrow_S & = & \mathcal{R} \cap ((S \times \mathbf{T}) \cup (\mathbf{T} \times S)) \\
\mathcal{R} \Downarrow_S & = & \mathcal{R} \downarrow_S \cup \{(s,t) \mid s, t \notin S \\
 & & \quad \text{and } s \mathcal{R} u_1 \mathcal{R} \cdots \mathcal{R} u_n \mathcal{R} t \text{ for some } u_1, \ldots, u_n \in S \backslash \{\star\} (n \geq 0)\}.
\end{array}
$$

Definition 2.3.2.4. Let $\mathcal{R} \subseteq \mathbf{T} \times \mathbf{T}$ be a relation on time tags, and let $[\tilde{t}/\tilde{s}] = [t_1/s_1, \ldots, t_n/s_n]$ be a renaming on time tags. We define the relation $[\tilde{t}/\tilde{s}]\mathcal{R}$ by

$$[\tilde{t}/\tilde{s}]\mathcal{R} = \{([\tilde{t}/\tilde{s}]u, [\tilde{t}/\tilde{s}]u') \mid (u, u') \in \mathcal{R}\}.$$

Definition 2.3.2.5. Let $\prec$ be a tag ordering. We define $\prec_{s_1,\ldots,s_n \approx t_1,\ldots,t_n}$ as the least relation $\prec'$ such that $\prec \subseteq \prec'$ and $\forall u \in \mathbf{T} \forall i \in \{1,\ldots,n\}.((s_i \prec' u \Leftrightarrow t_i \prec' u) \wedge (u \prec' s_i \Leftrightarrow u \prec' t_i))$.

2.3.3 *Type Judgment Form.* A type judgment is of the form $\Gamma, \prec \vdash P$, where Γ is a type environment and $\prec$ is a tag ordering. We require that each element of $\mathbf{T_O}$ has at most one occurrence in a type judgment. Therefore, $x_1 : \rho^{t_1}, \ldots, x_n : \rho^{t_n}, \prec \vdash P$ is a type judgment only if $t_1, \ldots, t_n$ and the elements of $New(P)$ are distinct from each other. (Note that this is analogous to the usual assumption that all the bound variables in P and the variables in Γ are distinct from each other.)

Intuitively, $\Gamma, \prec \vdash P$ means that (1) the process P uses only the capabilities specified by Γ, (2) P fulfills all the obligations specified by Γ, and (3) P obeys the ordering specified by $\prec$ in fulfilling the obligations. An ordering $\prec$ specifies an *upper-bound* of the ordering of channel use in P, rather than the exact ordering. For example, if ρ_x, ρ_y are linear channel types and $x : \rho_x^{t_x}, y : \rho_y^{t_y}, \{(t_x, t_y)\} \vdash P$ holds, then communication on y *may* (not *must*) be blocked by communication on x (by $x?[n].\, y![n]$, for instance) in P: P may use x and y concurrently as $x?[n].\, \mathbf{0} \mid y![n]$ does. Note also that there may be an ordering between different type bindings. For example, $x : ?^{\omega}[?^1[int]^t]^{\star}, y : !^1[int]^s, \{(s,t)\} \vdash P$ implies that P cannot wait for a value on a channel received via x before it sends an integer along y.

3. TYPING RULES

We give typing rules below and mainly explain the conditions on tag orderings. The conditions on type environments are basically the same as those for the linear channel type system and are explained there in detail [Kobayashi et al. 1996].

(T-PAR) is one of the key rules. The rule should be read "if P_1, P_2 are respectively well typed under Γ_1, Γ_2 and a tag ordering $\prec$, and if $\Gamma_1 + \Gamma_2$ is well defined, then $P_1 \mid P_2$ is well typed under $\Gamma_1 + \Gamma_2$ and $\prec$." Note that the ordering $\prec$ must be shared between P_1 and P_2: it ensures that there is no conflict on the order of channel use between P_1 and P_2. For example, consider $P_1 = x?[].\, y![], P_2 = y?[].\, x![]$. P_1 and P_2 are respectively well typed under type environments $\Gamma_1 = x : ?^1[\,]^{t_x}, y : !^1[\,]^{t_y}$ and $\Gamma_2 = x : !^1[\,]^{t_x}, y : ?^1[\,]^{t_y}$, and $\Gamma_1 + \Gamma_2 = x : \updownarrow^1[\,]^{t_x}, y : \updownarrow^1[\,]^{t_y}$ is well defined. However, $P_1 \mid P_2$ is not well typed under $\Gamma_1 + \Gamma_2$ because P_1 and P_2 cannot be typed under the same ordering $\prec$: while P_1 requires $t_x \prec t_y$, P_2 requires the reverse ordering $t_y \prec t_x$.

$$\frac{\Gamma_1, \prec \vdash P_1 \qquad \Gamma_2, \prec \vdash P_2}{\Gamma_1 + \Gamma_2, \prec \vdash P_1 \mid P_2} \qquad \text{(T-PAR)}$$

If $\tilde{y}$ is sent on x by the expression $x![\tilde{y}]$, the components of $\tilde{y}$ can be used only after they are extracted from x. So, the type environment must allow the fulfillment of the obligations specified by $\Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}$ to be delayed until the communication on x is completed. It is ensured by the condition $s \prec^{\sharp} \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}$ in the rule (T-OUT). The condition $([\tilde{t}/\tilde{u}] \prec) \subseteq \prec$ means that if there is an ordering on $\tilde{u}$ in $\prec$, then there must also be the corresponding ordering on $\tilde{t}$, so that a receiver can safely use $\tilde{y}$ in an order specified by the ordering on $\tilde{u}$.

$$\frac{\begin{array}{c}\Gamma \textit{ unlimited} \\ s \prec^\sharp \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} \\ ([\tilde{t}/\tilde{u}]\prec) \subseteq \prec\end{array}}{\Gamma + x : (!^m[\tilde{\rho}^{\tilde{u}}]^s) + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}, \prec \vdash x![\tilde{y}]} \qquad \text{(T-OUT)}$$

In the rule for input expressions below, the condition $[\tilde{u}/\tilde{s}]\prec \subseteq \prec$ ensures that the components of $\tilde{z}$ must not be used in P in an order unspecified by the ordering on $\tilde{u}$. Since the body of an input expression can be executed only after communication on x is completed, we need the condition $t \prec^\sharp \Gamma$ to allow the fulfillment of the obligations specified by Γ to be delayed. The ordering on $\tilde{s}$ can be deleted, because all the dependencies on the use of the bound variables $\tilde{z}$ are already taken into account by the ordering on $\tilde{u}$. An additional condition $\{\tilde{s}\} \not\prec\not> \{\tilde{u}\}$ is necessary for a subtle technical reason.

$$\frac{\begin{array}{c}\Gamma, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P \\ t \prec^\sharp \Gamma \\ ([\tilde{u}/\tilde{s}]\prec) \subseteq \prec \qquad \{\tilde{s}\} \not\prec\not> \{\tilde{u}\} \\ m = 1 \text{ or } \omega\end{array}}{\Gamma + x : ?^m[\tilde{\rho}^{\tilde{u}}]^t, \prec\downarrow_{\{\tilde{s}\}} \vdash x?[\tilde{z}].\, P} \qquad \text{(T-IN)}$$

The rule (T-MIN) for input prefixes on mutex channels is identical to (T-IN), except for the condition $O \in p$, which ensures that a value is put into x in the body of the input expression:

$$\frac{\begin{array}{c}\Gamma, x : p^M[\tilde{\rho}^{\tilde{u}}]^t, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P \qquad O \in p \\ t \prec^\sharp \Gamma \\ ([\tilde{u}/\tilde{s}]\prec) \subseteq \prec \qquad \{\tilde{s}\} \not\prec\not> \{\tilde{u}\}\end{array}}{\Gamma, x : ?^M[\tilde{\rho}^{\tilde{u}}]^t, \prec\downarrow_{\{\tilde{s}\}} \vdash x?[\tilde{z}].\, P} \qquad \text{(T-MIN)}$$

The rule (T-RIN) for replicated input expressions is also similar to (T-IN). The type environment Γ below must be unlimited because the body of a replicated input expression may be executed more than once. (The condition $u \prec^\sharp \Gamma$ is not required here because it is implied by the condition that Γ is unlimited.)

$$\frac{\begin{array}{cc}\Gamma, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P & \Gamma \text{ unlimited} \\ ([\tilde{u}/\tilde{s}]\prec) \subseteq \prec & \{\tilde{s}\} \not\prec\not> \{\tilde{u}\}\end{array}}{\Gamma + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star, \prec\downarrow_{\{\tilde{s}\}} \vdash x?^*[\tilde{z}].\, P} \qquad \text{(T-RIN)}$$

We have two rules for ν-expressions. The condition $\{s\} \not\prec\not> \mathbf{T_I}$ in the rule (T-NEW-1) means intuitively that there is only local dependency on the use of the created channel. If this is the case, the ordering on s can be removed from the type judgment. Otherwise, (T-NEW-2) must be applied and the ordering on s must be preserved.

$$\frac{\begin{array}{c}\Gamma, x : p^m[\tilde{T}]^s, \prec \vdash P \qquad p \text{ is either } \updownarrow \text{ or } | \\ \{s\} \not\prec\not> \mathbf{T_I}\end{array}}{\Gamma, \prec \Downarrow_{\{s\}} \vdash (\nu x : p^m[\tilde{T}]^s)\, P} \qquad \text{(T-NEW-1)}$$

$$\frac{\Gamma, x : p^m[\tilde{T}]^s, \prec \vdash P \qquad p \text{ is either } \updownarrow \text{ or } |}{\Gamma, \prec \vdash (\nu x : p^m[\tilde{T}]^s)\, P} \qquad \text{(T-NEW-2)}$$

In a conditional expression, both a type environment and a tag ordering are shared between the then-part and the else-part.

$$\frac{\Gamma, \prec \vdash P_1 \qquad \Gamma, \prec \vdash P_2}{\Gamma + x : \mathit{bool}^\star, \prec \vdash \mathit{if}\ x\ \mathit{then}\ P_1\ \mathit{else}\ P_2} \qquad \text{(T-IF)}$$

Examples

We give several examples of valid/invalid type judgments below. We write $P \longrightarrow Q$ when P is reduced to Q in one step, and $P \Longrightarrow Q$ when P is reduced to Q in zero or more steps. They are the same as the standard reduction relation for the π-calculus [Milner 1993], and formal definitions are given in Section 4. The time tags for *int* and *bool* (which are always $\star$) are omitted in the rest of this article.

Example 3.1. Let $\prec = \{(t_1, t_2)\}$. Then,

$$x : \updownarrow^1[\,]^{t_1}, y : \updownarrow^1[\,]^{t_2}, \prec \vdash x?[\,].\, y![\,] \mid x![\,] \mid y?[\,].\, \mathbf{0},$$

but

$$x : \updownarrow^1[\,]^{t_1}, y : \updownarrow^1[\,]^{t_2}, \prec \not\vdash x?[\,].\, y![\,] \mid y?[\,].\, x![\,].$$

Example 3.2. Let us consider a process $P = x?[y, z].\, y?[n].\, z![n] \mid x![v, w]$, and let $\prec = \{(t_x, t_v), (t_v, t_w), (t_x, t_w), (u_y, u_z)\}$. Then, we have

$$y : ?^1[\mathit{int}]^{s_y}, z : !^1[\mathit{int}]^{s_z}, \prec' \vdash y?[n].\, z![n]$$

for $\prec' = \{(s_y, s_z)\} \cup \prec$. By (T-IN), we obtain

$$x : ?^1[?^1[\mathit{int}]^{u_y}, !^1[\mathit{int}]^{u_z}]^{t_x}, \prec \vdash x?[y, z].\, y?[n].\, z![n].$$

On the other hand, $x![v, w]$ is typed as

$$x : !^1[?^1[\mathit{int}]^{u_y}, !^1[\mathit{int}]^{u_z}]^{t_x}, v : ?^1[\mathit{int}]^{t_v}, w : !^1[\mathit{int}]^{t_w}, \prec \vdash x![v, w].$$

Thus, we obtain

$$x : \updownarrow^1[?^1[\mathit{int}]^{u_y}, !^1[\mathit{int}]^{u_z}]^{t_x}, v : ?^1[\mathit{int}]^{t_v}, w : !^1[\mathit{int}]^{t_w}, \prec \vdash P.$$

The condition $t_x \prec t_v \prec t_w$ implies that x is first used for communication, v is used next, and then w is used. Note that $\prec$ can be inferred from the given type environment and process: the subexpression $x?[y, z].\, y?[n].\, z![n]$ requires $u_y \prec u_z$, and the subexpression $x![v, w]$ requires $t_x \prec t_v, t_w$ and $[t_v/u_y, t_w/u_z]\ \prec \subseteq \prec$, by which we obtain $t_x \prec t_v \prec t_w$. Such reconstruction of a tag ordering is discussed later in Section 5.

Example 3.3. Let $P = x?[y].\, y?[n].\, z![n] \mid x![v]$, and $\Gamma = x : \updownarrow^1[?^1[\mathit{int}]^{u_y}]^{t_x}, z : !^1[\mathit{int}]^{t_z}, v : ?^1[\mathit{int}]^{t_v}$. Then, $\Gamma, \prec \vdash P$ for $\prec = \{(t_x, t_v), (u_y, t_z), (t_v, t_z), (t_x, t_z)\}$.

Example 3.4. Let $P = x?[].\, y?[].\, (x![] \mid y![] \mid z![])$ and $\Gamma = x : ?^M[]^{t_x},\ y : ?^M[]^{t_y},\ z : !^1[]^{t_z}$. Then, $\Gamma, \prec \vdash P$ for $\prec = \{(t_x, t_z), (t_y, t_z), (t_y, t_x)\}$.

Example 3.5. Let us consider the following process, which corresponds to a recursive function definition.

$$P = f?^*[b,r].\,(\mathit{if}\ b\ \mathit{then}\ r![\,]\ \mathit{else}\ (\nu r' : \updownarrow^1[\,]^{t_r})\,(f![b,r'] \mid r'?[\,].\,r![\,])).$$

By applying (T-NEW-1) to

$$f : !^*[\mathit{bool}, !^1[\,]^u]^\star, b : \mathit{bool}, r : !^1[\,]^s, r' : \updownarrow^1[\,]^{t_r}, \{(t_r, s)\} \vdash f![b,r'] \mid r'?[\,].\,r![\,],$$

we obtain

$$f : !^*[\mathit{bool}, !^1[\,]^u]^\star, b : \mathit{bool}, r : !^1[\,]^s, \emptyset \vdash (\nu r' : \updownarrow^1[\,]^{t_r})\,(f![b,r'] \mid r'?[\,].\,r![\,]).$$

By (T-IF) and (T-RIN), we further obtain $f : \updownarrow^*[\mathit{bool}, !^1[\,]^u]^\star, \emptyset \vdash P$.

The following two examples indicate that the side condition $\{s\} \not\asymp \mathbf{T_I}$ in (T-NEW-1) cannot be dropped.

Example 3.6. Let P be the following process:

$$z![\,] \mid w?^*[x,y].\,(z?[\,].\,x?[\,].\,(x![\,] \mid z![\,]) \mid y?[\,].\,z?[\,].\,(y![\,] \mid z![\,]))$$

Then,

$$w : ?^*[?^M[\,]^{u_1}, ?^M[\,]^{u_2}]^\star, z : \updownarrow^M[\,]^t, \{(u_1,t),(t,u_2)\} \vdash P.$$

If we could ignore the side condition of (T-NEW-1), we would have

$$w : ?^*[?^M[\,]^{u_1}, ?^M[\,]^{u_2}]^\star, \{(u_1,u_2)\} \vdash (\nu z : \updownarrow^M[\,]^t)\,P,$$

from which we could obtain

$$\begin{aligned} &w : ?^*[?^M[\,]^{u_1}, ?^M[\,]^{u_2}]^\star, \{(u_1,u_2)\} \vdash \\ &\quad (\nu v_1)\,(\nu v_2)\,(\nu v_3)\,((\nu z)\,P \mid w![v_1,v_2] \mid w![v_2,v_3] \mid v_1![\,] \mid v_2![\,] \mid v_3![\,]). \end{aligned}$$

However, it may deadlock as follows:

$$\begin{aligned} &(\nu v_1)\,(\nu v_2)\,(\nu v_3)\,((\nu z)\,P \mid w![v_1,v_2] \mid w![v_2,v_3] \mid v_1![\,] \mid v_2![\,] \mid v_3![\,]) \\ \Longrightarrow\ &(\nu v_1)\,(\nu v_2)\,(\nu v_3)\,(\nu z)\,(z?[\,].\,v_2?[\,].\,(v_2![\,] \mid z![\,]) \mid v_2?[\,].\,z?[\,].\,(v_2![\,] \mid z![\,]) \\ &\qquad \mid z![\,] \mid v_1![\,] \mid v_2![\,] \mid v_3![\,] \mid \cdots) \\ \Longrightarrow\ &(\nu v_1)\,(\nu v_2)\,(\nu v_3)\,(\nu z) \\ &\qquad (v_2?[\,].\,(v_2![\,] \mid z![\,]) \mid z?[\,].\,(v_2![\,] \mid z![\,]) \mid v_1![\,] \mid v_3![\,] \mid \cdots) \end{aligned}$$

Example 3.7. Let

$$\begin{aligned} P &= m?[y].\,y?[\,].\,x![\,] \mid n![x] \\ \Gamma &= m : ?^1[?^1[\,]^{u_y}]^{t_m}, n : !^1[?^1[\,]^{u_x}]^{t_n} \\ \prec &= \{(u_y,t_x),(t_m,t_x),(t_n,t_x)\}. \end{aligned}$$

Then, $\Gamma, x : \updownarrow^1[\,]^{t_x}, \prec \vdash P$. We can apply (T-NEW-2) to obtain $\Gamma, \prec \vdash (\nu x : \updownarrow^1[\,]^{t_x})\,P$, but we cannot apply (T-NEW-1) to obtain $\Gamma, \emptyset \vdash (\nu x : \updownarrow^1[\,]^{t_x})\,P$. Note that if we ignored the side condition of (T-NEW-1), we would have

$$m : \updownarrow^1[?^1[\,]^{u_y}]^{t_m}, n : \updownarrow^1[?^1[\,]^{u_x}]^{t_n}, \{(t_n,t_m)\} \vdash (\nu x : \updownarrow^1[\,]^{t_x})\,P \mid n?[z].\,m![z].$$

But $(\nu x : \updownarrow^1[\,]^{t_x})\,P \mid n?[z].\,m![z]$ falls into deadlock:

$$\begin{aligned} (\nu x)\,P \mid n?[z].\,m![z] &\longrightarrow (\nu x)\,(m?[y].y?[\,].x![\,] \mid m![x]) \\ &\longrightarrow (\nu x)\,(x?[\,].x![\,]). \end{aligned}$$

Technical Properties

We state several key lemmas that will be used for proving type soundness: Lemmas 3.8–3.12 are used for strengthening and weakening the typing assumption (the left-hand side) of a type judgment. Lemma 3.13 corresponds to the substitution lemma in the usual type systems. Readers who are not interested in proofs can safely skip the rest of this section.

LEMMA 3.8. *If* $\Gamma, x:T, \prec \vdash P$ *and* x *is not free in* P*, then* T *is unlimited and* $\Gamma, \prec \vdash P$.

PROOF. By the typing rules, $x:T$ must be introduced in (T-OUT). Therefore, T must be unlimited and we can obtain $\Gamma, \prec \vdash P$ from a derivation of $\Gamma, x:T, \prec \vdash P$ by dropping all the bindings on x. □

LEMMA 3.9. *Let* Γ *be a type environment and* T *be an unlimited type such that* $\Gamma + x:T, \prec \vdash P$ *is well formed. Then* $\Gamma, \prec \vdash P$ *implies* $\Gamma + x:T, \prec \vdash P$.

PROOF. This is proved by straightforward induction on typing derivations. (A derivation of $\Gamma + x:T, \prec \vdash P$ can be obtained from that of $\Gamma, \prec \vdash P$ by replacing each type environment Γ' in the derivation with $\Gamma' + x:T$.) □

LEMMA 3.10. *Suppose* $\Gamma, \prec \vdash P$ *and* $s \in \mathbf{T_O}$ *does not appear in* Γ, P*. Then,* $\Gamma, \prec\downarrow_{\{s\}} \vdash P$.

PROOF. If s has been removed somewhere in the derivation of $\Gamma, \prec \vdash P$, then $\prec$ cannot contain s, i.e., $\prec = \prec\downarrow_{\{s\}}$. So $\Gamma, \prec\downarrow_{\{s\}} \vdash P$ follows immediately. Otherwise, a derivation of $\Gamma, \prec\downarrow_{\{s\}} \vdash P$ is obtained from that of $\Gamma, \prec \vdash P$ just by replacing each ordering $\prec_1$ in the derivation of $\Gamma, \prec \vdash P$ with $\prec_1\downarrow_{\{s\}}$. □

LEMMA 3.11. *Suppose* $\Gamma, \prec \vdash P$*. Then* $\Gamma, \prec \cup \prec' \vdash P$ *holds if* $\prec'$ *satisfies the following conditions:*

—$(\prec \cup \prec')^+$ *is a strict partial order and*
—$\prec'\uparrow_{\mathbf{T_I} \cup S} = \emptyset$,
where S *is the set of time tags excluded from the tag ordering by* (T-NEW-1) *in the derivation of* $\Gamma, \prec \vdash P$.

PROOF. The derivation of $\Gamma, \prec \cup \prec' \vdash P$ is obtained from that of $\Gamma, \prec \vdash P$ just by replacing each ordering $\prec_1$ in the derivation of $\Gamma, \prec \vdash P$ with $\prec_1 \cup \prec'$. □

LEMMA 3.12. *Suppose* $\Gamma, \prec \vdash P$*, and* t *is a fresh time tag taken from* $\mathbf{T_O}$*; then* $\Gamma, \prec \cup [t/s] \prec \vdash P$.

PROOF. It is trivial that $(\prec \cup [t/s] \prec)^+$ is a strict partial order. The derivation of $\Gamma, \prec \cup [t/s] \prec \vdash P$ can be obtained from that of $\Gamma, \prec \vdash P$ just by replacing each ordering $\prec_1$ in the derivation of $\Gamma, \prec \vdash P$ with $\prec_1 \cup [t/s] \prec_1$. □

LEMMA 3.13. *The rule*

$$\frac{\Gamma, x : \rho^{t_x}, \prec \vdash P}{\Gamma + y : \rho^{t_y}, [t_y/t_x] \prec \vdash [y/x]P} \qquad \text{(T-REN)}$$

is admissible, i.e., if

—$\Gamma, x : \rho^{t_x}, \prec \vdash P$,
—$([t_y/t_x]\ \prec)^+$ *is a strict partial order, and*
—*the judgment* $\Gamma + y : \rho^{t_y}, [t_y/t_x]\ \prec \vdash [y/x]P$ *is well formed,*

then $\Gamma + y : \rho^{t_y}, [t_y/t_x]\ \prec \vdash [y/x]P$ *is derivable.*

PROOF. See Appendix A.1. □

4. TYPE SOUNDNESS AND PARTIAL DEADLOCK-FREEDOM

In this section, we present an operational semantics and show that typing is respected by the operational semantics: in particular, we check that reliable channels (i.e., linear, replicated input, or mutex) never cause deadlock in a well-typed process.

4.1 Reduction Semantics

As usual, the operational semantics is defined via two relations: a structural congruence and a reduction relation [Milner 1993].

The following structural congruence simplifies the definition of the reduction relation.

Definition 4.1.1. Structural congruence $\equiv$ is the least congruence closed under the following rules.

$$P \mid Q \equiv Q \mid P \qquad \text{(S-COMMUT)}$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad \text{(S-ASSOC)}$$
$$(\nu x{:}T)\, P \mid Q \equiv (\nu x{:}T)\, (P \mid Q) \quad x \text{ not free in } Q \qquad \text{(S-EXTR)}$$

Note that structurally congruent processes have the same typing property:

LEMMA 4.1.2. *If* $P \equiv Q$, *then* $\Gamma, \prec \vdash P$ *if and only if* $\Gamma, \prec \vdash Q$.

PROOF. See Appendix A.2. □

To simplify the arguments below, we introduce the notion of normal forms.

Definition 4.1.3. A process $(\nu\tilde{x}{:}\tilde{T})\, (P_1 \mid \cdots \mid P_n)$ is in *normal form* if $P_1, \ldots, P_n$ are guarded processes, where a process is *guarded* if it is an input, output, replicated input, or a conditional expression.

LEMMA 4.1.4. *For any process* P, *there is some* Q *in normal form such that* $P \equiv Q$.

PROOF. By the associativity and commutativity of $\mid$, and the fact that any $(\nu\tilde{x}{:}\tilde{T})$ not guarded by input or replicated prefixes can be moved to the outside using (S-EXTR). □

The reduction relation is basically the same as the standard one [Milner 1993; Sangiorgi 1992], but it is annotated with some extra information [Kobayashi et al. 1996] that is used later for stating properties of the semantics. The reduction relation is of the form $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. ($\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$ is a five-place relation as a whole. It says nothing about typing of P and P' and therefore, it should not be confused with $(\Gamma \vdash P) \wedge (P \xrightarrow{m} P') \wedge (\Gamma' \vdash P')$.) Intuitively, the type

environment Γ represents the capabilities that can be used before the reduction, while Γ' represents those that can be used after the reduction. The multiplicity m records on which kind of channel the reduction is performed. $rename(P)$ in the rule (R-RCOM) represents a process expression obtained from P by renaming all the time tags in $New(P)$ with fresh time tags. Note that for well-typed processes, the reduction relation defined below coincides with the standard reduction relation for untyped process calculi.

$$\Gamma, x : \updownarrow^m[\tilde{T}]^t \vdash x![\tilde{y}] \mid x?[\tilde{z}].\, P \xrightarrow{m} [\tilde{y}/\tilde{z}]P \dashv \Gamma, x : Rem(\updownarrow^m[\tilde{T}]^t) \quad \text{(R-COM)}$$

$$\Gamma, x : \updownarrow^*[\tilde{T}]^\star \vdash x![\tilde{y}] \mid x?^*[\tilde{z}].\, P \xrightarrow{*} [\tilde{y}/\tilde{z}]rename(P) \mid x?^*[\tilde{z}].\, P \dashv \Gamma, x : \updownarrow^*[\tilde{T}]^\star \quad \text{(R-RCOM)}$$

$$\frac{\Gamma \vdash P \xrightarrow{m} Q \dashv \Gamma'}{\Gamma \vdash P \mid R \xrightarrow{m} Q \mid R \dashv \Gamma'} \quad \text{(R-PAR)}$$

$$\frac{\Gamma, x : T \vdash P \xrightarrow{m} Q \dashv \Gamma', x : T'}{\Gamma \vdash (\nu x : T)\, P \xrightarrow{m} (\nu x : T')\, Q \dashv \Gamma'} \quad \text{(R-NEW)}$$

$$\Gamma \vdash \mathit{if\ true\ then}\ P\ \mathit{else}\ Q \xrightarrow{1} P \dashv \Gamma \quad \text{(R-IFT)}$$

$$\Gamma \vdash \mathit{if\ false\ then}\ P\ \mathit{else}\ Q \xrightarrow{1} Q \dashv \Gamma \quad \text{(R-IFF)}$$

$$\frac{P \equiv P' \qquad \Gamma \vdash P' \xrightarrow{m} Q' \dashv \Gamma' \qquad Q' \equiv Q}{\Gamma \vdash P \xrightarrow{m} Q \dashv \Gamma'} \quad \text{(R-CONG)}$$

Notation 4.1.5. When the label m is not important, we just write $\Gamma \vdash P \longrightarrow Q \dashv \Delta$ for $\Gamma \vdash P \xrightarrow{m} Q \dashv \Delta$. We write $\Gamma \vdash P \longrightarrow Q$ if $\Gamma \vdash P \longrightarrow Q \dashv \Delta$ for some Δ and write $\Gamma \vdash P \not\longrightarrow$ (or $P \not\longrightarrow$ when Γ is known from the context) if there is no Q such that $\Gamma \vdash P \longrightarrow Q$.

Definition 4.1.6. $\Gamma \vdash P \Longrightarrow Q \dashv \Delta$ is the least relation closed under the following rules.

$$\Gamma \vdash P \Longrightarrow P \dashv \Gamma$$

$$\frac{\Gamma \vdash P \longrightarrow Q \dashv \Delta \qquad \Delta \vdash Q \Longrightarrow R \dashv \Theta}{\Gamma \vdash P \Longrightarrow R \dashv \Theta}$$

We just write $P \Longrightarrow Q$ if $\Gamma \vdash P \Longrightarrow Q \dashv \Delta$ for some Γ and Δ. For well-typed processes, $P \Longrightarrow Q$ coincides with the reflexive and transitive closure of the standard reduction relation [Milner 1993].

4.2 Partial Confluence

As claimed in Kobayashi et al. [1996], communication on a linear or replicated input channel enjoys the confluence property. It implies that communication on those channels essentially preserves the equivalence class of a process. We do not give any proof here since it is the same as that in Kobayashi et al. [1996].

THEOREM 4.2.1 (PARTIAL CONFLUENCE). *Suppose that* $\Gamma, \prec \vdash P$ *for some* $\prec$ *and* $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$ *for* $m = 1$ *or* $*$. *If* $\Gamma \vdash P \xrightarrow{m'} Q \dashv \Delta$, *then* $P' \equiv Q$ *or there is some* Q', Δ' *such that* $\Gamma' \vdash P' \xrightarrow{m'} Q' \dashv \Delta'$ *and* $\Delta \vdash Q \xrightarrow{m} Q' \dashv \Delta'$.

4.3 Partial Deadlock-Freedom

Deadlock-freedom is ensured by two theorems: (1) subject reduction theorem (Theorem 4.3.1), which states that typing is preserved by reductions, and (2) Theorem 4.3.2, which states that reliable channels do not cause deadlock immediately in a well-typed process.

The subject reduction theorem is stated below. The major difference from the ordinary subject reduction is that the reduced process expression is well typed not under the original type environment but under the new type environment obtained by removing the used capabilities from the original one. The tag ordering is changed by the reduction, but not in a significant way: the new tag ordering is obtained just by some renaming of the original ordering.

THEOREM 4.3.1. *If* $\Gamma, \prec \vdash P$ *and* $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$, *then* $\Gamma', \prec_{\tilde{s} \approx \tilde{s'}} \vdash P'$ *for* $\tilde{s}, \tilde{s'}$ *such that*

(1) $\{\tilde{s'}\} = New(P') \backslash New(P)$ *and*
(2) $\{\tilde{s}\} \subseteq New(P)$.

Note that $\prec_{\tilde{s} \approx \tilde{s'}}$ is always a tag ordering because $\tilde{s'}$ are fresh time tags.

PROOF. Induction on the derivation of $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. See Appendix A.3. □

The theorem below ensures that reliable channels (linear, replicated input, and mutex channels) cannot immediately cause deadlock. We use the shorthand $\{\,|\,Q\}$ for either nothing or a parallel composition with the process Q. We say that x is *half used* in Γ if $\Gamma(x)$ is of the form $?^m[\tilde{T}]^t$ or $!^m[\tilde{T}]^t$. The theorem essentially says that if a well-typed process P can no longer be reduced, then either (1) some subprocess is trying to communicate over some unreliable or half used channel (in the first case below); or (2) all computation has been completed and only processes corresponding to process definitions (replicated input expressions) or stores (i.e., outputs on mutex channels) are left (in the second case below).

THEOREM 4.3.2. *Suppose* $\Gamma, \prec \vdash P$ *and* $\{x \mid \Gamma(x) = bool\} \subseteq \{true, false\}$. *If* $\Gamma \vdash P \not\longrightarrow$, *then one of the following conditions holds:*

(1) *Either* $P \equiv (\nu \tilde{w}{:}\tilde{U})\,(x?[\tilde{z}].\,Q\{\,|\,R\})$ *or* $P \equiv (\nu \tilde{w}{:}\tilde{U})\,(x![\tilde{z}]\{\,|\,R\})$; *and* x *has an unreliable channel type (in* Γ *or* $\tilde{w}{:}\tilde{U}$*) or is half used in* Γ.
(2) $P \equiv (\nu \tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ *and each* P_i *is either of the form* $x![\tilde{y}]$ *for a mutex channel* x *or of the form* $x?^*[\tilde{y}].\,Q$.

PROOF. Let $(\nu\tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ be a normal form of P. By Lemma 4.1.2, $\Gamma, \prec \vdash (\nu\tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$. By (T-NEW-1), (T-NEW-2), and (T-PAR), there exist $\Gamma_1, \ldots, \Gamma_n$ and $\prec_1$ such that

$$\Gamma_i, \prec_1 \vdash P_i \ (1 \le i \le n)$$
$$\Gamma_1 + \cdots + \Gamma_n = \Gamma, \tilde{w}{:}\tilde{U}.$$

None of $P_i (i \in \{1, \ldots, n\})$ is a conditional expression, because otherwise it must be that $P_i =$ *if true then* Q_1 *else* Q_2 or $P_i =$ *if false then* Q_1 *else* Q_2, which contradicts with the assumption $P \not\rightarrow$. Suppose that the first condition of the above theorem does not hold, i.e., no process is trying to communicate over an unreliable or half-used channel. Then, without loss of generality, we can assume that

—$P_i = x![\tilde{y}]$ and $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^*[\tilde{T}]^\star$ for some x for each $i (1 \le i \le j)$,
—$P_i = x?[\tilde{y}].\,Q$ and $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^M[\tilde{T}]^t$ for some x for each $i (j+1 \le i \le k)$,
—$(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^1[\tilde{T}]^t$ and either $P_i = x![\tilde{y}]$ or $P_i = x?[\tilde{y}].\,Q$ for some x for each $i (k+1 \le i \le l)$, and
—either $P_i = x![\tilde{y}]$ for some mutex channel x, or $P_i = x?^*[\tilde{y}].\,Q$ for some x, for each $i (l+1 \le i \le n)$

where $0 \le j \le k \le l \le n$. We show $l = 0$, which implies the second condition of the theorem.

First, suppose $j > 0$. Then, $P_1 = x![\tilde{y}]$ and $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^*[\tilde{T}]^\star$ for some x. By the typing rules, $\Gamma_i(x) = p^*[\tilde{T}]^\star$ and $I \in p$ for some $i \in \{2, \ldots, n\}$. Again by the typing rules, P_i must be of the form $x?^*[\tilde{v}].\,Q$, which contradicts with the assumption $P \not\rightarrow$. Therefore, j must be 0.

Let us define the set S by

$$\begin{aligned} S \;=\; & \{t_i \mid P_i = x_i?[\tilde{y}].\,Q_i \wedge \Gamma_i(x_i) = p^M[\tilde{T}]^{t_i} \wedge (1 \le i \le k)\} \\ \cup\; & \{t_i \mid P_i = x_i![\tilde{y}] \wedge \Gamma_i(x_i) = p^1[\tilde{T}]^{t_i} \wedge (k+1 \le i \le l)\} \\ \cup\; & \{t_i \mid P_i = x_i?[\tilde{y}].\,Q_i \wedge \Gamma_i(x_i) = p^1[\tilde{T}]^{t_i} \wedge (k+1 \le i \le l)\}. \end{aligned}$$

Note that $\star \notin S$. We shall show $S = \emptyset$ (i.e., $l = 0$) by contradiction, which completes the proof. Suppose that $S \neq \emptyset$. Then because $\prec_1^+$ is a strict partial order, there is $t_i \in S$ such that there is no $t' \in S$ such that $t' \prec_1 t_i$. The case analysis below shows that such t_i does not exist. The key point is that some process $P_{i'}$ must hold the other end of x_i, and since t_i is a minimal element of S, the first prefix of $P_{i'}$ must be on x_i, which implies P_i and $P_{i'}$ can be further reduced.

—Suppose that t_i comes from the first set of S, i.e.,

$$P_i = x_i?[\tilde{y}].\,Q \wedge \Gamma_i(x) = p^M[\tilde{T}]^{t_i} \wedge (1 \le i \le k).$$

By (T-IN), p must be ?. Since $(\Gamma, \tilde{w}{:}\tilde{U})(x_i) = \updownarrow^M[\tilde{T}]^{t_i}$, there is some $i' (\neq i)$ such that $\Gamma_{i'}(x_i) = q^M[\tilde{T}]^{t_i}$ and $O \in q$. Since t_i is a minimal element of S, it must be that $l+1 \le i' \le n$. So, $P_{i'}$ must be either $u?^*[\tilde{z}].\,R$ or $u![\tilde{v}]$. The first case contradicts with the condition of (T-RIN) ($\Gamma_{i'}(x_i)$ must be unlimited). In the second case, since the output end of a mutex channel cannot be put into a mutex channel (by the condition on type expressions), u must be x_i, which contradicts with the condition $P \not\rightarrow$.

—Suppose that t_i comes from the second set of S, i.e.,

$$P_i = x_i![\tilde{y}] \wedge \Gamma_i(x_i) = p^1[\tilde{T}]^{t_i} \wedge (k+1 \leq i \leq l).$$

By (T-OUT), p must be !. Since $(\Gamma, \tilde{w}{:}\tilde{U})(x_i) = \updownarrow^1[\tilde{T}]^{t_i}$, there is some $i'(\neq i)$ such that $\Gamma_{i'}(x_i) = ?^1[\tilde{T}]^{t_i}$. Because t_i is a minimal element of S, it must be that $k+1 \leq i' \leq n$. If $k+1 \leq i' \leq l$, then $P_{i'}$ must be of the form $x_i?[\tilde{u}].Q_{i'}$, which contradicts with the fact that $P \not\longrightarrow$. If $l+1 \leq i' \leq n$, then $P_{i'}$ is either an output on a mutex channel or a replicated input. The first case contradicts with the condition that the input end of a linear channel cannot be put into a mutex channel, and the second case contradicts with the condition of (T-RIN) ($\Gamma_{i'}(x_i)$ must be unlimited).

—Case where t_i comes from the third set of S: Similar to the second case. □

The subject reduction property (Theorem 4.3.1) together with the lack of immediate deadlock (Theorem 4.3.2) ensures the following *partial deadlock-freedom property.*

THEOREM 4.3.3 (PARTIAL DEADLOCK-FREEDOM). *Suppose that* $\Gamma, \prec \vdash P$ *and* $\{x \mid \Gamma(x) = bool\} \subseteq \{true, false\}$. *If* $\Gamma \vdash P \Longrightarrow P' \dashv \Gamma'$ *and* $\Gamma' \vdash P' \not\longrightarrow$, *then one of the following conditions holds:*

(1) *either* $P' \equiv (\nu\tilde{w}{:}\tilde{U})\,(x?[\tilde{z}].Q\{\mid R\})$, *or* $P' \equiv (\nu\tilde{w}{:}\tilde{U})\,(x![\tilde{z}]\{\mid R\})$; *and* x *has an unreliable channel type (in* Γ *or* $\tilde{w}{:}\tilde{U}$*) or is half used in* Γ.

(2) $P' \equiv (\nu\tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ *and each* P_i *is either of the form* $x![\tilde{y}]$ *for a mutex channel* x *or of the form* $x?^*[\tilde{y}].Q$.

PROOF. Corollary of Theorem 4.3.1 and Theorem 4.3.2. □

The following corollary is a special case of Theorem 4.3.3.

COROLLARY 4.3.4. *Suppose that* $\Gamma, \prec \vdash P$, $\{x \mid \Gamma(x) = bool\} \subseteq \{true, false\}$, *and that all the channel types in* Γ, P *are reliable channel types and that no channel in* Γ *is half used. If* $\Gamma \vdash P \Longrightarrow Q \dashv \Delta$, *then, either*

(1) $Q \equiv (\nu\tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ *and each* P_i *is either of the form* $x![\tilde{y}]$ *for a mutex channel* x *or of the form* $x?^*[\tilde{y}].R$ *or*

(2) $\Delta \vdash Q \longrightarrow Q' \dashv \Delta'$ *for some* Δ', Q'.

PROOF. A special case of Theorem 4.3.3. □

By the above corollary, a program never falls into deadlock if only reliable channels are used. For the cases where unreliable channels are used, Theorem 4.3.2 says just that, in a deadlocking process, some sub-process is always trying to communicate over an unreliable or half-used channel. However, its proof also implies that if there is a process blocked on a reliable channel, then some process is trying to communicate over an unreliable or half-used channel while blocking communication on a reliable channel. Moreover, the proof indicates that we can identify such a bad process by looking at the tag ordering.

5. TYPE CHECK

In this section, we show that there is an algorithm, which, given a type environment Γ and an expression P, decides whether or not there exists a tag ordering $\prec$ such that $\Gamma, \prec \vdash P$. It should be clear from the informal discussion below that the algorithm runs in time polynomial in the size of input: (Γ, P).

The algorithm consists of the following steps:

(1) First, ignore conditions on $\prec$ and construct a derivation tree of $\Gamma, \prec \vdash P$.
(2) Extract conditions on $\prec$ and decide whether or not there exists a tag ordering $\prec$.

The first step is done by reading the typing rules in a bottom-up manner. The nontrivial case is the one where we encounter (T-PAR): we must split the current type environment for two subprocesses. The key technique to avoid combinatorial explosion is to split the type environment *lazily*. In order to type check $\Gamma, \prec \vdash P_1 \mid P_2$ (with conditions on $\prec$ ignored), we first check P_1 (with information that its type environment should be a part of Γ). If the check succeeds, we can remove the used type environment Γ_1 from Γ, and check P_2 under the resulting environment Γ_2. The typing rules are reformulated in Appendix B in order to express this algorithm. The same idea is also used for proving a linear logic sequent "$\Gamma \vdash A \otimes B$" in linear logic programming languages [Hodas and Miller 1994] and for reconstructing types in linear functional programming languages [Mackie 1994].

Note that the first step can be done in a polynomial time (with respect to the size of Γ, P): only one rule can be applied at each step of the construction of a derivation tree,[3] the size of the derivation tree is linear in the size of P, and the required computation (for deciding which rule should be applied and removing the used type environment at the node of (T-PAR)) at each node of the derivation tree is linear in the size of Γ, P.

The second step is described as follows. First, introduce the set $\{\beta_{s,t} \mid s, t \in \mathbf{T}\}$ of variables ranging over $\{0, 1\}$. Then, $\prec$ is completely determined by the values of the variables: we can define $\beta_{s,t} = 1$ if $s \prec t$, and $\beta_{s,t} = 0$ otherwise. All the conditions on $\prec$ are expressed by constraints of the form

$$\begin{array}{l} \beta_{t_1,t_2} = 1 \\ \beta_{t_1,t_2} \geq \beta_{s_1,s_2} \\ \beta_{t_1,t_2} = 0. \end{array}$$

Constraints of the first form come from conditions of the form $t \prec^\sharp \Gamma$, the second constraints from $\theta \prec \subseteq \prec$, and the third from $S_1 \not\prec\not\succ S_2$. Then, the above constraints can be solved by the following simple iterations:

(1) First, let $\beta^{(0)}_{t_1,t_2} = 1$ if there is a constraint $\beta_{t_1,t_2} = 1$, and let $\beta^{(0)}_{t_1,t_2} = 0$ for other (t_1, t_2). Let $k \leftarrow 0$.
(2) If there is a constraint $\beta_{t_1,t_2} \geq \beta_{s_1,s_2}$, and if $\beta^{(k)}_{t_1,t_2} = 0, \beta^{(k)}_{s_1,s_2} = 1$, then $\beta^{(k+1)}_{t_1,t_2} = 1$. Otherwise $\beta^{(k+1)}_{t_1,t_2} = \beta^{(k)}_{t_1,t_2}$. If $\beta^{(k+1)}_{t_1,t_2} \neq \beta^{(k)}_{t_1,t_2}$ for some t_1, t_2, then let $k \leftarrow k+1$ and repeat this step.

[3](T-NEW-1) and (T-NEW-2) are regarded as the same rule in this step, since they differ only in a condition on time tags.

(3) For each constraint $\beta_{t_1,t_2} = 0$, check whether $\beta^{(k)}_{t_1,t_2} = 0$.

For the resulting relation $\prec$, whether or not $\prec^+$ is a strict partial order can be checked by a well-known cycle detection algorithm for directed graphs.

The second step can be done in time polynomial in the size of an input (Γ, P). Let n be the size of an input. Then the number of time tags appearing in the derivation is $O(n)$. Because the number of variables is $O(n^2)$, the number of steps of iterations is at most $O(n^2)$. Because the number of constraints is at most $O(n^4)$, the time required for each step is $O(n^4)$. Therefore, the total time used for solving constraints is $O(n^6)$. (This is a very rough estimate, and we could obtain a much better upper bound. For example, if we assume that the arity of a channel, i.e., the number of arguments passed through a channel, is bound by some constant, $O(n^4)$ is an upper bound.)

There is one important thing that has not been mentioned above: how to choose (T-NEW-1) or (T-NEW-2) when extracting the constraints on time tags in the beginning of the second phase. Actually, the extraction and solving of the constraints must be overlapped. However, it does not affect the above estimation of the computational cost: in order to extract the constraints from a derivation for $(\nu x : \rho^s)\, P$, we can first gather the constraints from the sub-derivation for P, solve the constraints on s locally, and then choose (T-NEW-1) or (T-NEW-2) according to whether or not the side condition $\{s\} \not\asymp \mathbf{T_I}$ is met by the local solution.

6. FUNCTIONS AND CONCURRENT OBJECTS ON TOP OF THE CALCULUS

We show that functions and typical concurrent objects are implemented by using reliable channels, and demonstrate applications of the theorems on partial deadlock-freedom. We need only type information for inferring that a function or method will return a result unless it falls into an infinite loop. Note that such reasoning, unlike in typed functional languages, has not been possible in previous typed concurrent programming languages (including typed process calculi and typed concurrent object-oriented languages).

We first discuss encoding of the call-by-value, simply typed λ-calculus in detail. Then, we show that the call-by-name and call-by-need simply typed λ-calculi can also be encoded into the deadlock-free fragment. Encodings of concurrent objects are also informally presented and discussed.

6.1 Encoding Call-by-Value λ-Calculus

We show encoding of call-by-value, simply typed λ-calculus into our process calculus.[4] As will be clear from the encoding, we could obtain the same result in the presence of recursion.

[4] We expect that it is possible to encode λ-calculi with more advanced type systems by appropriate extensions of our process calculus with subtyping [Pierce and Sangiorgi 1993], polymorphism [Pierce and Sangiorgi 1997], etc. It is because the communication behavior of a process obtained by encoding a function does not depend so much on the presence of subtyping or polymorphism. For example, a polymorphic identity function would be encoded into a process like $f?^*[\alpha, x : \alpha, r : !^1[\alpha]].\, r![x]$, which essentially has the same communication behavior as its monomorphic counterpart: $f?^*[x, r].\, r![x]$.

6.1.1 *Syntax and Typing of* $\lambda^{\rightarrow}$. We first introduce the syntax of the simply typed λ-calculus.

Definition 6.1.1.1. The sets of types and terms are given by the following syntax:

$$\begin{array}{rrll} \tau \text{ (types)} & ::= & b & \text{(base types)} \\ & & \tau_1 \rightarrow \tau_2 & \text{(function types)} \\ e \text{ (terms)} & ::= & x & \\ & & \lambda x^{\tau}.e & \\ & & e_1 e_2 & \end{array}$$

Definition 6.1.1.2. A term e has a type τ under a type environment $\mathcal{T}$ if $\mathcal{T} \vdash e : \tau$ can be derived by the following set of rules:

$$\mathcal{T}, x : \tau \vdash x : \tau \qquad (\lambda\text{T-VAR})$$

$$\frac{\mathcal{T}, x : \tau_1 \vdash e : \tau_2}{\mathcal{T} \vdash \lambda x^{\tau_1}.e : \tau_1 \rightarrow \tau_2} \qquad (\lambda\text{T-ABS})$$

$$\frac{\mathcal{T} \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{T} \vdash e_1 e_2 : \tau_2} \qquad (\lambda\text{T-APP})$$

LEMMA 6.1.1.3. *If* $\mathcal{T} \vdash e : \tau$ *and* $\mathcal{T} \vdash e : \tau'$, *then* $\tau = \tau'$.

PROOF. Trivial induction on the structure of e. □

Notation 6.1.1.4. By Definition 6.1.1.2 and Lemma 6.1.1.3, if e is well typed under the type environment $\mathcal{T}$, there is a unique τ such that $\mathcal{T} \vdash e : \tau$. We write $\mathcal{T}(e) = \tau$ if $\mathcal{T} \vdash e : \tau$.

6.1.2 *Encoding.* We encode $\lambda^{\rightarrow}$-terms into our calculus, following Milner [1990] and Pierce and Sangiorgi [1993]. We first give encoding of types and type environments. To understand the encoding of function types below, note that a function will be implemented as a process waiting on a replicated input channel for requests for evaluation, and that the invocation of the function will be implemented as a process that sends a request (consisting of an argument and a channel for receiving the result) on the replicated input channel (recall Example 2.2.4).

Definition 6.1.2.1. Types of $\lambda^{\rightarrow}$ are encoded by the following function $[\![\cdot]\!]$:

$$\begin{array}{rcl} [\![b]\!] & = & b^{\star} \\ [\![\tau_1 \rightarrow \tau_2]\!] & = & !^{*}[[\![\tau_1]\!], !^{1}[[\![\tau_2]\!]]^{u}]^{\star} \end{array}$$

Here, we assume that the same tag u is used in all encodings of function types. The encoding function is pointwise extended to type environments:

$$\begin{array}{rcl} [\![\emptyset]\!] & = & \emptyset \\ [\![\mathcal{T}, x : \tau]\!] & = & [\![\mathcal{T}]\!], x : [\![\tau]\!] \end{array}$$

LEMMA 6.1.2.2. *Let* τ *be a type of* $\lambda^{\rightarrow}$. *Then,* $[\![\tau]\!]$ *is an unlimited type.*

PROOF. Trivial by the definition of $[\![\tau]\!]$. □

Now we give an encoding of $\lambda^{\rightarrow}$-terms. Given a $\lambda^{\rightarrow}$-term e and its type environment $\mathcal{T}$, the encoding function $[\![e]\!]^a_{\mathcal{T}}$ returns a process that evaluates e and returns the result to the channel a. We use a linear channel as the result channel a. A variable just returns its name to the return address.

$$[\![x]\!]^a_{\mathcal{T}} = a![x]$$

The λ-abstraction $\lambda x^{\tau_1}.e$ first spawns a process which, given an argument x and a return address q, sends the result back to q and then it returns the address of the spawned process.

$$[\![\lambda x^{\tau_1}.e]\!]^a_{\mathcal{T}} = (\nu y : \updownarrow^*[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]^u]^\star)\,(a![y] \mid y?^*[x,q].\ [\![\ e]\!]^q_{\mathcal{T},x:\tau_1}\ )$$

Here $\tau_2 = (\mathcal{T}, x:\tau_1)(e)$.

A function application $e_1 e_2$ first spawns processes to evaluate the function e_1 and the argument e_2, and then applies the function to the argument.[5]

$$[\![e_1 e_2]\!]^a_{\mathcal{T}} = (\nu y : \updownarrow^1[[\![\tau_1 \rightarrow \tau_2]\!]]^{s_1})\,(\nu z : \updownarrow^1[[\![\tau_1]\!]]^{s_2})\ ([\![e_1\ ]\!]^y_{\mathcal{T}}\ \mid\ [\![\ e_2\ ]\!]^z_{\mathcal{T}}\ \mid y?[f].\, z?[v].\, f![v,a])$$

Here s_1, s_2 are fresh time tags and $\tau_1 = \mathcal{T}(e_2), \tau_2 = \mathcal{T}(e_1 e_2)$.

6.1.3 *Reasoning about Encoded Terms.* We show that a process $[\![e]\!]^x_{\mathcal{T}}$ will eventually return a value to x or fall into an infinite loop.[6] All we need to check is that $[\![e]\!]^x_{\mathcal{T}}$ is well typed: once it is checked, the required property is obtained as an immediate corollary of the theorems on partial deadlock-freedom.

THEOREM 6.1.3.1 (ENCODING PRESERVES TYPING). *If $\mathcal{T} \vdash e : \tau$, then we have $[\![\mathcal{T}]\!], a : !^1[[\![\tau]\!]]^t, \emptyset \vdash [\![e]\!]^a_{\mathcal{T}}$ for a fresh time tag t.*

PROOF. Proved by induction on the structure of e.

—Case $e = x$: In this case, $\mathcal{T} = \mathcal{T}', x:\tau$ and $P = a![x]$. Since $[\![\mathcal{T}']\!]$ is unlimited, we have $[\![\mathcal{T}']\!], a : !^1[[\![\tau]\!]]^t, x : [\![\tau]\!], \emptyset \vdash a![x]$ by (T-OUT), i.e., $[\![\mathcal{T}]\!], a : !^1[[\![\tau]\!]]^t, \emptyset \vdash P$.

—Case $e = \lambda x^{\tau_1}.e'$. In this case, $\mathcal{T}, x : \tau_1 \vdash e' : \tau_2$. Let $P_1 = [\![e']\!]^q_{\mathcal{T},x:\tau_1}$. By induction hypothesis, we have

$$[\![\mathcal{T}]\!], x : [\![\tau_1]\!], q : !^1[[\![\tau_2]\!]]^{s_1}, \emptyset \vdash P_1$$

for a fresh time tag s_1. By (T-RIN),

$$[\![\mathcal{T}]\!], y : ?^*[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]^u]^\star, \emptyset \vdash y?^*[x,q].\, P_1.$$

Because $[\![\mathcal{T}]\!]$ is unlimited, we also have

$$[\![\mathcal{T}]\!], a : !^1[[\![\tau_1 \rightarrow \tau_2]\!]]^t, y : [\![\tau_1 \rightarrow \tau_2]\!], \emptyset \vdash a![y].$$

[5]The encoding presented here evaluates a function and its argument in parallel. Alternative encoding, which evaluates a function and its argument sequentially, is of course possible, and satisfies the same properties as those proved below.

[6]Actually, as we encode the simply typed λ-calculus, the process never falls into an infinite loop. But our type information is not sufficient for such reasoning.

By (T-PAR), (T-NEW-2), and the fact that $[\![\tau_1 \to \tau_2]\!] = !^*[[\![\tau_1]\!], !^1[[\![\tau_2]\!]^u]^\star$,

$$[\![\mathcal{T}]\!], a : !^1[[\![\tau_1 \to \tau_2]\!]^t, \emptyset \vdash P.$$

—Case $e = e_1 e_2$: In this case, $\mathcal{T} \vdash e_1 : \tau_1 \to \tau_2, \mathcal{T} \vdash e_2 : \tau_1$. Let $P_1 = [\![e_1]\!]^y_{\mathcal{T}}$ and $P_2 = [\![e_2]\!]^z_{\mathcal{T}}$. By induction hypothesis, we have

$$[\![\mathcal{T}]\!], y : !^1[[\![\tau_1 \to \tau_2]\!]^{s_1}, \emptyset \vdash P_1$$
$$[\![\mathcal{T}]\!], z : !^1[[\![\tau_1]\!]^{s_2}, \emptyset, \vdash P_2$$

for fresh time tags s_1 and s_2. By Lemma 3.11,

$$[\![\mathcal{T}]\!], y : !^1[[\![\tau_1 \to \tau_2]\!]^{s_1}, \prec \vdash P_1$$
$$[\![\mathcal{T}]\!], z : !^1[[\![\tau_1]\!]^{s_2}, \prec, \vdash P_2$$

for $\prec = \{(s_1, s_2), (s_2, t), (s_1, t)\}$. On the other hand, by (T-IN) and (T-OUT),

$$[\![\mathcal{T}]\!], y : ?^1[[\![\tau_1 \to \tau_2]\!]^{s_1}, z : ?^1[[\![\tau_1]\!]^{s_2}, a : !^1[[\![\tau_2]\!]^t, \prec \vdash y?[f].\, z?[v].\, f![v, a].$$

By (T-PAR) and (T-NEW-1), $[\![\mathcal{T}]\!], a : !^1[[\![\tau_2]\!]^t, \emptyset \vdash P$. □

We obtain the theorem below as a corollary of Theorems 4.3.3 and 6.1.3.1. The theorem essentially says that the evaluation of an encoded term never gets stuck. Note that we also know by Theorem 4.2.1 that the returned result is unique up to some appropriate typed process congruence. (We do not give a formal definition of the process congruence, but a weak typed barbed congruence that is similar to the one developed for the linear channel type system [Kobayashi et al. 1996] will suffice.)

THEOREM 6.1.3.2. *Let e be a closed $\lambda^{\to}$-term (i.e., $\emptyset \vdash e : \tau$). If $a : !^1[[\![\tau]\!]^t \vdash [\![e]\!]^a_\emptyset \Longrightarrow P' \dashv \Delta$ for some Δ. Then either*

(1) $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(a![v]\{\,|\,Q\})$ or
(2) there is some Q such that $\Delta \vdash P' \longrightarrow Q$.

PROOF. Let $T = !^1[[\![\tau]\!]^t$. By Theorem 6.1.3.1, $a : T, \emptyset \vdash [\![e]\!]^a_\emptyset$. By the definition of the reduction relation, $\Delta = a : T$. Suppose that $\Delta \vdash P' \not\longrightarrow$. By the subject reduction theorem (Theorem 4.3.1), we have $a : T, \emptyset \vdash P'$. Since every channel in P' is a linear or replicated input channel, we have either

(1) $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(a![v]\{\,|\,Q\})$ or
(2) $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ and each P_i is of the form $x?^*[\tilde{y}].\, Q$

by Theorem 4.3.3. The second case cannot happen because the type environment Δ is not unlimited. □

6.2 Encoding Call-by-Name λ-Calculus

This subsection presents an encoding of the call-by-name, simply typed λ-calculus. We can again infer that a process which simulates a well-typed $\lambda^{\to}$-term never deadlocks by using type information. We use the same syntax for λ-terms and types as in Section 6.1.

We follow Ostheimer and Davie [1993] and Turner [1996] for the encoding of call-by-name reduction. In call-by-name reduction, an argument is passed to a function before it is evaluated. So, a function type is represented by the type of channel

along which a caller process passes a pair of channels, one of which is used by a callee process to trigger the evaluation of the argument, and the other of which is used to return the result. Thus, encoding of types and type environments is given as follows.

Definition 6.2.1. Types of $\lambda^{\rightarrow}$ are encoded by the following function $[\![\cdot]\!]$:

$$\begin{aligned} [\![b]\!] &= b^{\star} \\ [\![\tau_1 \rightarrow \tau_2]\!] &= !^{*}[!^{*}[!^{1}[[\![\tau_1]\!]]^{u}]^{\star}, !^{1}[[\![\tau_2]\!]]^{u}]^{\star} \end{aligned}$$

(The same time tag is used for u in all encodings of function types.)

A type binding $x : \tau$ is encoded into $x : !^{*}[!^{1}[[\![\tau]\!]]^{u}]^{\star}$. This is because the name of a variable in a λ-term is used as the name of a channel on which a request for a value should be sent.

Definition 6.2.2. Encoding of type environments is given by

$$\begin{aligned} [\![\emptyset]\!] &= \emptyset \\ [\![\mathcal{T}, x : \tau]\!] &= [\![\mathcal{T}]\!], x : !^{*}[!^{1}[[\![\tau]\!]]^{u}]^{\star}. \end{aligned}$$

We give an encoding of terms below. As in the previous section, the encoding function $[\![e]\!]^{a}_{\mathcal{T}}$ returns a process that evaluates e (in the call-by-name style) and returns the result to the channel a.

A variable is implemented as a process that requests the value of the variable to be sent to a. The name of a variable is used as the name of the channel on which the request should be sent.

$$[\![x]\!]^{a}_{\mathcal{T}} = x![a]$$

The encoding of a λ-abstraction $\lambda x^{\tau_1}.e$ is almost the same as the one in the call-by-value λ-calculus, except for a type annotation.

$$[\![\lambda x^{\tau_1}.e]\!]^{a}_{\mathcal{T}} = (\nu y : \updownarrow^{*}[!^{*}[!^{1}[[\![\tau_1]\!]]^{u}]^{\star}, !^{1}[[\![\tau_2]\!]]^{u}]^{\star})\,(a![y] \mid y?^{*}[x, q].\ [\![\ e]\!]^{q}_{\mathcal{T}, x:\tau_1}).$$

Here $\tau_2 = (\mathcal{T}, x : \tau_1)(e)$.

The function application $e_1 e_2$ first spawns a process to evaluate the function e_1, then it passes to the function two channels: one for triggering evaluation of the argument and the other for receiving the result. We use an auxiliary process $\langle\!\langle e \rangle\!\rangle^{x}_{\mathcal{T}}$. It waits on a channel x for a request for the value of e, and upon a request, it evaluates e and returns the value.

$$\begin{aligned} &[\![e_1 e_2]\!]^{a}_{\mathcal{T}} = \\ &\quad (\nu z : \updownarrow^{*}[!^{1}[[\![\tau_1]\!]]^{u}]^{\star})\,(\nu y : \updownarrow^{1}[[\![\tau_1 \rightarrow \tau_2]\!]]^{s})\,([\![e_1]\!]^{y}_{\mathcal{T}} \mid \langle\!\langle e_2 \rangle\!\rangle^{z}_{\mathcal{T}} \mid y?[f].\, f![z, a]) \\ &\langle\!\langle e \rangle\!\rangle^{z}_{\mathcal{T}} = z?^{*}[w].\ [\![\ e]\!]^{w}_{\mathcal{T}} \end{aligned}$$

Here $\tau_1 = \mathcal{T}(e_2), \tau_2 = \mathcal{T}(e_1 e_2)$, and s is a fresh time tag.

We can again prove the same theorems as those on the encoding of call-by-value reduction: typing is preserved by the encoding, and as its corollary, we can reason that the evaluation of an encoded term never gets stuck. Proofs are omitted here because they are almost the same as those for the call-by-value λ-calculus.

THEOREM 6.2.3. *If* $\mathcal{T} \vdash e : \tau$, *then* $[\![\mathcal{T}]\!], a : !^{1}[[\![\tau]\!]]^{t}, \emptyset \vdash [\![e]\!]^{a}_{\mathcal{T}}$ *for a fresh time tag* t.

THEOREM 6.2.4. *Let e be a closed $\lambda^{\rightarrow}$-term (i.e., $\emptyset \vdash e : \tau$). If $a : !^1[[\![\tau]\!]]^t \vdash [\![e]\!]^a_\emptyset \Longrightarrow P' \dashv \Delta$ for some Δ, then either*

(1) $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(a![v]\{\mid Q\})$ or

(2) there is some Q such that $\Delta \vdash P' \longrightarrow Q$.

6.3 Encoding Call-by-Need λ-Calculus

We can obtain encoding of the call-by-need λ-calculus by minor modification to the encoding of the call-by-name λ-calculus. We need to redefine $\langle\!\langle e \rangle\!\rangle^x_{\mathcal{T}}$. In the case of the call-by-need λ-calculus, it waits on a channel x for a request for the value of e, and upon the first request, evaluates e and supplies the value. After that, it just supplies the value computed before for each request. We also modify the encoding function so that it also returns an appropriate tag ordering under which the encoded process should be typed.

The encoding $\langle\!\langle e \rangle\!\rangle^x_{\mathcal{T}}$ is given by (t in $[\![e]\!]^{k,t}_{\mathcal{T}}$ below represents the time tag of k)

$$
\begin{array}{l}
\langle\!\langle e \rangle\!\rangle^x_{\mathcal{T}} = \\
\quad \textbf{let } (P, \prec) = [\![e]\!]^{k,t}_{\mathcal{T}} \\
\quad \textbf{in } ((\nu flag : \updownarrow^M[bool]^{s_1})\,(\nu r : \updownarrow^M[[\![\tau]\!]]^{s_2}) \\
\qquad (flag![true] \mid r![dummy] \mid \\
\qquad x?^*[y].\, flag?[b]. \\
\qquad\quad (\mathit{if}\ b \\
\qquad\quad \mathit{then}\ (\nu k : \updownarrow^1[[\![\tau]\!]]^t)\,(P \mid k?[v].\, r?[v'].\, (r![v] \mid flag![false] \mid y![v])) \\
\qquad\qquad \mathit{else}\ r?[v].\, (r![v] \mid flag![false] \mid y![v]))), \\
\qquad \prec\downarrow_{\{t\}} \cup\{(s_1, u), (s_2, u), (s_2, s_1)\}),
\end{array}
$$

where s_1, s_2, t are fresh time tags and where $\tau = [\![\mathcal{T}(e)]\!]$. The above encoding uses a mutex channel $flag$ to remember whether or not e has been evaluated, and another mutex channel r to memorize the value of e. For simplicity, we regard $dummy$ as a constant of type $[\![\tau]\!]$ here. Strictly speaking, for each function type $\tau = \tau_1 \rightarrow \tau_2$, there should be a free replicated input channel $dummy_\tau$, and a process $dummy_\tau?^*[x, r].\, r![dummy_{\tau_2}]$ should run in parallel to the processes obtained by the above encoding (and also type bindings $dummy_\tau : \updownarrow^*[!^*[!^1[[\![\tau_1]\!]]^u]^\star, !^1[[\![\tau_2]\!]]^u]^\star$ should appear in type environments).

Other encodings are given below. They are the same as the encoding of the call-by-name λ-calculus except for orderings.

$$
\begin{array}{l}
[\![x]\!]^{a,t}_{\mathcal{T}} = (x![a], \emptyset) \\
[\![\lambda x^{\tau_1}.e]\!]^{a,t}_{\mathcal{T}} = \\
\quad \textbf{let } (P, \prec) = [\![e]\!]^{q,s}_{\mathcal{T}, x:\tau_1} \\
\quad \textbf{in } ((\nu y : \updownarrow^*[!^*[!^1[[\![\tau_1]\!]]^u]^\star, !^1[[\![(\mathcal{T}, x : \tau_1)(e)]\!]]^u]^\star)\,(a![y] \mid y?^*[x, q].\, P), \\
\qquad \prec\downarrow_{\{s\}}) \\
[\![e_1 e_2]\!]^{a,t}_{\mathcal{T}} = \\
\quad \textbf{let } (P_1, \prec_1) = [\![e_1]\!]^{y,s}_{\mathcal{T}} \\
\qquad (P_2, \prec_2) = \langle\!\langle e_2 \rangle\!\rangle^z_{\mathcal{T}} \\
\quad \textbf{in } ((\nu z : \updownarrow^*[!^*[[\![\mathcal{T}(e_2)]\!]]^u]^\star)\,(\nu y : \updownarrow^1[[\![\mathcal{T}(e_1)]\!]]^s)\,(P_1 \mid P_2 \mid y?[f].\, f![z, a]), \\
\qquad (\prec_1 \cup \prec_2 \cup [t/u](\prec_1 \cup \prec_2)) \downarrow_{\{s\}})
\end{array}
$$

We can again prove that typing is preserved by the encoding, and as its corollary, we can reason, by using type information, that the evaluation of the encoded term never gets stuck. Unfortunately, because mutex channels are used, type information is not sufficient for reasoning about the uniqueness of the result.

THEOREM 6.3.1. *Suppose* $\mathcal{T} \vdash e : \tau$, *and let* $(P, \prec) = [\![e]\!]_{\mathcal{T}}^{a,t}$. *Then* $\prec^+$ *is a strict partial order, and* $[\![\mathcal{T}]\!], true : bool, false : bool, a : !^1[[\![\tau]\!]]^t, \prec \vdash P$.

THEOREM 6.3.2. *Let* e *be a closed* $\lambda^{\rightarrow}$*-term (i.e.,* $\emptyset \vdash e : \tau$*), and* $(P, \prec) = [\![e]\!]_{\emptyset}^{a,t}$. *If* $a : !^1[[\![\tau]\!]]^t \vdash P \Longrightarrow P' \dashv \Delta$ *for some* Δ. *Then, either*

(1) $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(a![v]\{\,|\,Q\})$ *or*
(2) *there is some* Q *such that* $\Delta \vdash P' \longrightarrow Q$.

6.4 Concurrent Objects

A concurrent object is a very useful data structure in concurrent programs. The main differences between a concurrent object and a functional object are that a concurrent object has an internal state and that a concurrent object must have a mechanism to ensure the atomic execution of a method (because more than one process may request method execution simultaneously).

We show two encodings of concurrent objects into the deadlock-free fragment of our typed calculus. The first one is simple and efficient, but restricted (for example, a concurrent object cannot invoke its own method during execution of a state-updating method). The second one is slightly more complicated, but more expressive.

6.4.1 *Simple Encoding.* Basically, a concurrent object is modeled by multiple processes, each of which handles each method. In order to avoid simultaneous executions of methods of the same objects (that may lead to inconsistency), a state is implemented by using a mutex channel. The encoding presented here is based on the one by Pierce and Turner [1995].

The process below corresponds to a definition of concurrent objects. We use records for readability: $\{l_1 = x_1, \ldots, l_n = x_n\}$ creates a record whose field l_i has value x_i for each $i(\in \{1, \ldots, n\})$, and $x.l$ extracts the value of field l from record x.

$$
\begin{array}{l}
new?^*[init, r]. \\
\quad (\nu st : \updownarrow^M [T_0]^{t_s})\,(\nu m_1 : \updownarrow^* [\tilde{T_1}]^\star) \cdots (\nu m_n : \updownarrow^* [\tilde{T_n}]^\star) \\
\quad (r![\{meth_1 = m_1, \ldots, meth_n = m_n\}] \\
\quad \mid st![init] \\
\quad \mid m_1?^*[v, c].\, st?[state].\, (\cdots (c![res] \mid st![state'])) \\
\quad \mid \cdots \\
\quad \mid m_n?^*[v, c].\, st?[state].\, (\cdots (c![res] \mid st![state'])))
\end{array}
$$

The above process waits for a request for the creation of a concurrent object with an initial state *init* and methods $meth_1, \ldots, meth_n$. When it receives a request, it stores the initial state in the internal (mutex) channel st, spawns processes to handle the methods, and returns a record each field of which contains a channel on which a request for method execution should be sent. The record of channels corresponds to the reference to a concurrent object. Each method handler $(m_i?^*[v, c].\, \cdots)$ repeatedly receives a message, reads the current state, and executes the corresponding

method. When it finishes the method execution, it replies with the result and writes a new state into st. Thus, the above concurrent objects are implemented by using linear channels (used for receiving the reference of a created object and for receiving the result of method execution), mutex channels (used for storing states), and replicated input channels (used for receiving requests for method executions).

In the above example, the mutex channel st plays an important role in guaranteeing safe execution of methods: because st is a mutex channel, other method handlers must wait at $st?[state].\cdots$ until a new state is written, so that the consistency of the object's state is guaranteed. Moreover, since a method handler having the lock on st will eventually free the lock, the other method handlers do not remain blocked forever (if the method does not fall into infinite loop and the fairness is preserved).

Example 6.4.1.1. The following process is a definition of counter objects. (For simplicity, we assume primitive operations on integers.)

$$\begin{array}{l} P = newcounter?^*[init, r]. \\ \quad (\nu st : \updownarrow^M[int]^t)\,(\nu inc : \updownarrow^*[!^1[\,]^{u_1}]^\star)\,(\nu read : \updownarrow^*[!^1[int]^{u_1}]^\star) \\ \quad (r![\{inc = inc, read = read\}] \\ \quad \mid st![init] \\ \quad \mid inc?^*[c].\, st?[n].\, (c![\,] \mid st![n+1]) \\ \quad \mid read?^*[c].\, st?[n].\, (c![n] \mid st![n]) \end{array}$$

P is typed as

$$newcounter : ?^*[int, !^1[\{inc : !^*[!^1[\,]^{u_1}]^\star, read : !^*[!^1[int]^{u_1}]^\star\}]^{u_2}]^\star, \{(t, u_1)\} \vdash P.$$

The following process Q creates a new counter object and invokes the methods inc and $read$:

$$\begin{array}{l} Q = (\nu r : \updownarrow^1[\{inc : !^*[!^1[\,]^{u_1}]^\star, read : !^*[!^1[int]^{u_1}]^\star\}]^{s_1}) \\ \quad (newcounter![3, r] \mid r?[o].\, (\nu w : \updownarrow^1[\,]^{s_2})\,(o.inc![w] \mid w?[\,].\, (o.read![result]))) \end{array}$$

We get 4 in the channel *result* when this process is executed in parallel to the counter definition process. By our type system, $P \mid Q$ is typed as

$$\begin{array}{l} newcounter : \updownarrow^*[int, !^1[\{inc : !^*[!^1[\,]^{u_1}]^\star, \\ read : !^*[!^1[int]^{u_1}]^\star\}]^{u_2}]^\star, result : !^1[int]^{s_3}, \{(t, u_1), (t, s_3)\} \vdash P \mid Q. \end{array}$$

By the theorems in Section 4, the above type judgment implies that if $P \mid Q$ is reduced to a process which can no longer be reduced, then the process must be congruent to $(\nu \tilde{w}{:}\tilde{W})\,(result![m] \mid R)$, i.e., we get a value in the channel *result*.

The following example shows limitations of the simple encoding of concurrent objects: invocation of its own method causes deadlock with the simple encoding (a similar deadlock can happen also when two concurrent objects call mutual methods before releasing their locks on the states). Our type system can correctly detect such cases.

Example 6.4.1.2. Let us consider a case where a method of a concurrent object

calls itself. With the above simple encoding, a definition is expressed as follows:

$$
\begin{array}{l}
new?^*[init, r]. \\
(\nu st : \updownarrow^M [int]^s)\,(\nu m : \updownarrow^* [!^1[int]^u]^\star) \\
\quad (r![m] \\
\quad \mid st![init] \\
\quad \mid m?^*[c].\, st?[n].\, ((\nu c' : \updownarrow^1 [int]^t)\,(m![c'] \mid c'?[res'].\, (c![res] \mid st![n'])))
\end{array}
$$

An object created by the above definition has a single method m. In the body of the method $(st?[n].\, \cdots)$, the method acquires a state, creates a new channel c', and calls itself. It results in deadlock as the following reductions show:

$$
\begin{array}{rl}
m![x] \mid st![k] \mid m?^*[c].\, Q & \longrightarrow\ st?[n].\, \cdots \mid st![k] \mid m?^*[c].\, Q \\
& \longrightarrow\ (\nu c')\,(m![c'] \mid c'?[res'].\, \cdots \mid m?^*[c].\, Q) \\
& \longrightarrow\ (\nu c')\,(st?[n].\, \cdots \mid c'?[res'].\, \cdots \mid m?^*[c].\, Q)
\end{array}
$$

In our type system, the above process $new?^*[init, r].\, \cdots$ is judged to be ill typed: We obtain $t \prec s$ from the subexpression $c'?[res'].\, (c![res] \mid st![n'])$, the ordering $s \prec u$ from $m?^*[c].\, \cdots$, and the condition $[t/u] \prec \subseteq \prec$ from $m![c']$. Thus, $\{(t, s), (s, t)\} \subseteq \prec$, which implies that $\prec^+$ is not a strict partial order.

6.4.2 *Alternative Encoding.* We show a slightly more complicated encoding of concurrent objects in order to allow a concurrent object to invoke a method of itself. The idea of the encoding is (1) to distinguish between a channel used for ensuring exclusive execution of methods and a channel for storing the state and (2) to spawn two processes for handling each method: one for handling method invocations from the inside of the object and the other for handling method invocations from the outside. The new process corresponding to a definition of a concurrent object looks like

$$
\begin{array}{l}
new?^*[init, r]. \\
\quad (\nu lock : \updownarrow^M []^{s_1})\,(\nu st : \updownarrow^M [T_0]^{s_2}) \\
\quad (\nu f_1 : \updownarrow^* [\tilde{T}_1]^\star) \cdots (\nu f_n : \updownarrow^* [\tilde{T}_n]^\star) \\
\quad (\nu m_1 : \updownarrow^* [\tilde{T}_1]^\star) \cdots (\nu m_n : \updownarrow^* [\tilde{T}_n]^\star) \\
\quad (r![\{meth_1 = m_1, \ldots, meth_n = m_n\}] \\
\quad \mid st![init] \\
\quad \mid f_1?^*[v, c].\, (\\
\qquad \textbf{let}\ self = \{meth_1 = f_1, \ldots, meth_n = f_n\}\ \textbf{in} \\
\qquad st?[state].\, (st![state] \mid \cdots st?[state].\, (c![v'] \mid st![state']))) \\
\quad \mid \cdots \\
\quad \mid f_n?^*[v, c].\, (\\
\qquad \textbf{let}\ self = \{meth_1 = f_1, \ldots, meth_n = f_n\}\ \textbf{in} \\
\qquad st?[state].\, (st![state] \mid \cdots st?[state].\, (c![v'] \mid st![state'])))) \\
\quad \mid m_1?^*[v, c].\, lock?[\,].\, (\nu c' : T_1')\,(f_1![v, c'] \mid c'?[res].\, (c![res] \mid lock![\,])) \\
\quad \mid \cdots \\
\quad \mid m_n?^*[v, c].\, lock?[\,].\, (\nu c' : T_n')\,(f_n![v, c'] \mid c'?[res].\, (c![res] \mid lock![\,])))
\end{array}
$$

where $\textbf{let}\ self = v\ \textbf{in}\ P$ is a shorthand for $(\nu x)\,(x![v] \mid x?[self].\, P)$. Note that we still use only reliable channels.

7. DISCUSSIONS

We discuss some of the remaining issues on our type system, and also mention several extensions and variants.

7.1 Compositional Type Checking and Type Reconstruction

Compositional type checking and type reconstruction are important in practice. Unfortunately, the current formulation of the type system is not completely suitable for the compositional type checking. Consider a parallel composition $P_1 \mid P_2$. Given typings $\Gamma_1, \prec_1 \vdash P_1$ and $\Gamma_2, \prec_2 \vdash P_2$, we cannot simply obtain $\Gamma_1 + \Gamma_2, \prec_1 \cup \prec_2 \vdash P_1 \mid P_2$; it must be checked that $\prec_1 \cup \prec_2$ satisfy the conditions required in the derivations of $\Gamma_1, \prec_1 \cup \prec_2 \vdash P_1$ and $\Gamma_2, \prec_1 \cup \prec_2 \vdash P_2$ (and we may need to add some orderings to $\prec_1 \cup \prec_2$). In order to make the rules look more compositional, it would be sufficient to use constraints C on the tag ordering in a type judgment and write $\Gamma, C \vdash P$, instead of using the tag ordering itself.

Type reconstruction is not so easy as type checking. However, it is not much more difficult than that for the linear channel type system [Kobayashi et al. 1996] because we can infer the required tag ordering automatically from a type environment and a type-annotated process expression. Since we have already developed a type reconstruction algorithm [Igarashi and Kobayashi 1997] for a variant of the linear channel type system, it would be possible to develop a (probably partial) type reconstruction algorithm along similar lines.

7.2 Variants and Extensions

7.2.1 *Cyclic Ordering.* The current type system requires that the transitive closure of a tag ordering should be a strict partial order; otherwise the process is ill typed and rejected. It may be too restrictive in practice because our type system sometimes rejects a process which actually does not deadlock. Instead of rejecting a cyclic ordering completely, we could provide it for the programmer as debugging information. Then, the programmer can examine the part warned by the type checker to know whether it really causes deadlock. For example, let t_x, t_y, and t_z be time tags of channels x, y, and z. Given a tag ordering $\{(t_x, t_y), (t_y, t_x), (t_y, t_z)\}$, a programmer knows that he or she needs to check the use of the channels x and y, but need not worry about the use of z.

7.2.2 *Lower Bound of the Order of Channel Use.* The tag ordering in our type system gives an *upper bound* of the dependency allowed in the use of channels. For example, let t_x, t_y be the time tags of x, y. $\{(t_x, t_y)\}$ means that a process *may* be blocked on x before performing communication on y, but it does not mean that a process *must* do so.

If we are interested in using the tag ordering as a specification of processes, a lower bound of the dependency would also be useful: for example, if the relation $\{(t_x, t_y)\}$ is given as a lower bound, x *must* be used before y. A new type judgment would be of the form $\Gamma, (\prec_1, \prec_2) \vdash P$ where $\prec_1$ is a lower-bound of the ordering on channel use while $\prec_2$ is an upper-bound of the ordering.

A lower bound would also be necessary if we want to guarantee deadlock-freedom for more general usage of channels which are currently classified as unreliable. For

example, consider the process $P = x?[\,].\, y?[\,].\, x![\,]$: it would be typed as

$$x : ?^1[\,]^{t_1}, x : ?^1[\,]^{t_2}, y : !^1[\,]^s, (\{(t_1, s), (s, t_2)\}, \{(t_1, s), (s, t_2)\}) \vdash P.$$

Note that we use here a multiset of type bindings as the type environment: two bindings $(x : ?^1[\,]^{t_1}, x : ?^1[\,]^{t_2})$ on x express that x is used twice for input in P. The lower and upper bounds of the ordering indicate that x is used for input at first, y is used next, and then x is used again for input. Let us consider another process Q that is typed as

$$x : !^1[\,]^{t_3}, x : !^1[\,]^{t_4}, y : !^1[\,]^s, (\{(t_3, s)\}, \{(t_3, s), (t_3, t_4)\}) \vdash Q.$$

It implies that Q uses y for output after it uses x either once or twice. If we run P and Q in parallel, we cannot tell whether the binding $x : !^1[\,]^{t_3}$ represents the first or second use of x. Thus, $P \,|\, Q$ would be typed as

$$\begin{array}{l} x : \updownarrow^1[\,]^{t_1}, x : \updownarrow^1[\,]^{t_2}, y : !^1[\,]^s, \\ (\{(t_1, s), (t_1, t_2)\}, \{(t_1, s), (t_1, t_2), (t_2, s), (s, t_2)\}) \vdash P \,|\, Q, \end{array}$$

and the ordering $\{(t_2, s), (s, t_2)\}$ indicates that $P \,|\, Q$ may deadlock: in fact, if Q is $x![\,] \,|\, x![\,].y![\,]$, then $P \,|\, Q$ can deadlock (here, we use an output guard $x![\,].P$ for simplicity). However, if we strengthen the requirement for Q by refining the lower-bound

$$x : !^1[\,]^{t_3}, x : !^1[\,]^{t_4}, y : !^1[\,]^s, (\{(t_3, s), (t_3, t_4)\}, \{(t_3, s), (t_3, t_4)\}) \vdash Q,$$

then $P \,|\, Q$ can be typed as

$$x : \updownarrow^1[\,]^{t_1}, x : \updownarrow^1[\,]^{t_2}, y : !^1[\,]^s, (\{(t_1, s), (t_1, t_2)\}, \{(t_1, s), (t_1, t_2)\}) \vdash P \,|\, Q.$$

Thus, deadlock-freedom is guaranteed. This example indicates that a lower bound of the ordering is strongly related to an upper bound and it sometimes plays an important role when we reason about deadlock.

7.2.3 *Subtyping on Time Tags.* Subtyping on time tags can be used to refine the deadlock-free fragment of the calculus. Consider the following type judgment for an output expression:

$$x : !^m[\rho_1^u]^{t_x}, y : \rho_2^{t_y}, \prec \vdash x![y].$$

In our type system, ρ_1 and ρ_2 must be exactly the same. This requirement sometimes propagates unnecessary orderings, which leads to the rejection of processes that actually do not deadlock. We can avoid it by allowing ρ_1 and ρ_2 to be different in their time tags. For example, if $\rho_1 = !^m[\rho_3^{u_1}]$ and $\rho_2 = !^m[\rho_3^{u_2}]$, then it is sufficient that the condition $[u_1/u_2]\ \prec \subseteq \prec$ holds. These conditions can be generalized to a kind of subtyping relation. The resulting rule for output expressions would (roughly) look like

$$\frac{\begin{array}{c} \Gamma \ \textit{unlimited} \\ t_x \prec^\sharp \Sigma\{\tilde{y} : \tilde{T}\} \\ \theta \vdash \tilde{T} \leq \tilde{S} \qquad \theta \prec \subseteq \prec \end{array}}{\Gamma + x : (!^m[\tilde{S}]^{t_x}) + \Sigma\{\tilde{y} : \tilde{T}\}, \prec \vdash x![\tilde{y}]}$$

where a relation $\theta \vdash \tilde{T} \leq \tilde{S}$ can be structurally defined on types $\tilde{T}, \tilde{S}$.

7.2.4 *Polymorphism on Time Tags.* The side condition of the rule (T-NEW-1) indicates that we cannot always forget the orderings on some of the channels bound by ν-prefixes. For example, in the encoding of concurrent objects shown in Section 6, the orderings on the time tags of mutex channels must be kept globally. It sometimes leads to the rejection of processes which actually do not deadlock.

For example, consider the following object definition:

$$\begin{array}{l} new?^*[init, r]. \\ (\nu st : \updownarrow^M [int]^s)\,(\nu m : \updownarrow^* [!^1[int]^u]^\star) \\ \quad (r![m] \\ \quad \mid st![init] \\ \quad \mid m?^*[o, c].\, st?[n].\, (\nu c')\,(o.m![dummy, c'] \mid c'?[res].\, (c![res] \mid st![n'])))). \end{array}$$

The above process creates an object with a method m. The method m invokes the method m of another object o. If o happens to be an object created by the above process, then our type system detects the cyclic dependency: $s \prec u, u \prec s$. This is because the orderings on the mutex channel used in o and those on st are merged by our type system.

We can avoid the above problem by allowing s to be passed via channels (and by introducing subtyping):

$$new?^*[init, r, s].\, (\nu st : \updownarrow^M [int]^s)\,(\nu m : \updownarrow^* [!^1[int]^u]^\star)\,(st![init] \mid \cdots)$$

7.2.5 *Infinite Set of Time Tags.* In the current type system, only a finite number of time tags can appear in a type environment and a process. Therefore, the same tag may be assigned to distinct channels, which may cause nonexisting deadlock to be detected. One solution would be to use a finite representation of an ordering on an infinite set of time tags: a "regular" infinite graph (an analogue of a regular tree) whose leafs are labelled with time tags. It would be useful, for instance, for introducing recursive types: a recursive type $\mu X.!^1[X]^t$ can be viewed as a regular tree $!^1[!^1[!^1[\cdots]^{t.1.1.0}]^{t.1.0}]^{t.0}$ with the relation $\{(t.0, t.1.0), (t.1.0, t.1.1.0), \cdots\}$.

By combining an infinite time tag set and polymorphism on time tags, we might be able to eliminate the ad hoc rule (T-NEW-1). For example, the process of Example 3.5 would be written as

$$P = f?^*[b, r, X].\, (\mathit{if}\ b\ \mathit{then}\ r![]\ \mathit{else}\ (\nu r' : \updownarrow^1 []^{X.0})\,(f![b, r', X.1] \mid r'?[].\, r![]))$$

where an infinite set of time tags $\{t.0, t.1.0, t.1.1.0, \ldots\}$ with the relation $\{(t.1.0, t.0), (t.1.1.0, t.1.0), \ldots\}$ should be passed through X.

7.3 Process Equivalence

Since the tag ordering restricts processes that can be composed, it might be interesting to develop a new theory of process equivalence that takes the ordering into account. However, we think that, with the current type system, the resulting process equivalence would not be much coarser than the weak typed barbed congruence studied in the linear channel type system [Kobayashi et al. 1996]. One might expect that $x?[].\, P$ is behaviorally equivalent to $x?[].\, \mathbf{0} \mid P$ if x is a reliable channel; however, it is not the case. There are two main reasons for this: one reason is that unreliable channels may be used to block a sender on x, as in $(\nu y : \updownarrow^\omega [])\,(y?[].\, x![])$; and the other reason is that, even if the use of unreliable channels is forbidden, a

process may be infinitely reduced without ever sending a value to x—for example, consider $y![x] \mid y?^*[z].\, y![z]$.

8. RELATED WORK

Although a number of type systems for process calculi have so far been proposed, as far as the author knows, our type system is the first one that can deal with deadlock in a satisfactory manner in name passing process calculi like the π-calculus (except for an earlier version of our type system [Kobayashi 1996b], which cannot ensure that typical concurrent objects are deadlock-free). It is because each type binding was handled separately in previous type systems (as in type systems for functional programming languages), while we need to keep track of dependency between type bindings to ensure deadlock-freedom. The type system by Berger et al. [1997] ensures deadlock-freedom, but the calculus is synchronous (in the sense that all processes must proceed in lock-step) and its communication topology is completely static.

Our type system is built on top of the previous type systems for process calculi: the input-only/output-only channel type system [Pierce and Sangiorgi 1993] and the linear channel type system [Kobayashi et al. 1996]. Especially, we used ideas of the latter to ensure the correct usage of communication channels, though ordering information was not introduced there.

Recently, Yoshida [1996] independently studied a type system for the monadic π-calculus where types can be represented by graph structures. They improved an equivalence theory of processes using the graph types, and proved the full abstraction theorem on the encoding of the polyadic π-calculus [Milner 1993] into the monadic π-calculus [Milner et al. 1992]. Although the purpose of their type system is different from ours, their type system and ours seem to have some similarities: an edge of their graphs corresponds roughly to an ordering between two time tags in our type system. Therefore, it might be interesting to develop a unifying, more general theory of their and our type system.

Sangiorgi [1997] introduced the notion of *receptiveness*: roughly speaking, a channel is receptive if its input end is immediately used after it is created. Their receptive channel is, therefore, very similar to our replicated input channel. They showed how receptiveness can improve an equivalence theory of processes.

Nestmann [1997] used our type system for showing partial correctness—the fact that the encoding does not introduce deadlock—of his encoding of a version of the π-calculus with the choice operator ($+$) into the one without choice.

Several techniques have been proposed for static analysis of communication in CML: Nielson and Nielson [1994] proposed a technique to extract terms of process algebra from CML programs and Colby [1995] proposed another technique based on abstract interpretation. Those techniques help programmers understand the behaviors of CML programs to some extent. However, (a possibly infinite number of) multiple communication channels are mapped to the same abstract channel during static analysis (which is called a *region* in Nielson and Nielson's analysis). Therefore, we cannot use those techniques to reason about deadlock, etc. To see why we cannot reason about deadlock, consider the process $x?[\,].\, \mathbf{0} \mid y![\,]$. If there is no process communicating over x or y in parallel to this process, it falls into deadlock. However, if we alias x and y to the same channel z (this can in fact

happen in Nielson's analysis when x and y are sent somewhere through the same channel), the resulting process $z?[\,].\,\mathbf{0} \mid z![\,]$ can be reduced to $\mathbf{0}$. On the other hand, consider the process $x?[\,].\,z?[\,].\,y![\,] \mid x![\,] \mid y?[\,].\,\mathbf{0} \mid z![\,]$. It is always reduced to $\mathbf{0}$. However, if we map x and y to the same channel w, then the resulting process

$$w?[\,].\,z?[\,].\,w![\,] \mid w![\,] \mid w?[\,].\,\mathbf{0} \mid z![\,]$$

may be reduced to the blocked process:

$$w?[\,].\,z?[\,].\,w![\,] \mid \mathbf{0} \mid z![\,].$$

As indicated by those examples, once distinct channels are mapped to the same channel, whether or not the original program deadlocks is irrelevant to whether or not its approximation does.

9. CONCLUSION

We have proposed a static type system for a process calculus that ensures partial deadlock freedom and shown that functions and typical concurrent objects can be implemented in the deadlock-free fragment of the calculus.

In addition to the issues discussed in Section 7, future work includes applications of our type system to static debuggers, compile-time optimizations, and run-time systems of concurrent programming languages. More studies of actual concurrent programs would also be necessary to judge whether or not the present type system is sufficient and to find an appropriate extension of the type system.

An idea of applying our type system to compile-time optimizations is as follows. Consider, for example, a process $(\nu x : \updownarrow^1[int]^t)\,(P \mid Q)$, and suppose that P uses the output end of x and Q uses the input end of x. Suppose also we know by type information that P does not deadlock. Then, if we schedule P before Q, there is no need to check whether a value is in the channel x before Q receives the value from x. We are now trying to formalize this idea using the compilation framework developed by Oyama et al. [1997].

APPENDIX

A. PROOFS OF MAJOR LEMMAS AND THEOREMS

A.1 Proof of Lemma 3.13

PROOF. The proof follows from the fact that (T-REN) applied just below (T-OUT) can be merged into (T-OUT), while (T-REN) applied just below other rules can be permuted upward. Because the other cases are similar and simpler, we show only the cases where (T-REN) is applied below (T-OUT), (T-IN), or (T-NEW-1).

—Case where (T-REN) is applied below (T-OUT): The derivation must be of the form

$$\dfrac{\dfrac{}{\Gamma_1 + w : (!^m[\tilde{\rho_z}^{\tilde{u_z}}]^{s_w}) + \Sigma\{\tilde{z} : \tilde{\rho_z}^{\tilde{s_z}}\}, \prec \vdash w![\tilde{z}]}\ \text{(T-OUT)}}{[t_y/t_x, y/x](\Gamma_1 + w : (!^m[\tilde{\rho_z}^{\tilde{u_z}}]^{s_w}) + \Sigma\{\tilde{z} : \tilde{\rho_z}^{\tilde{s_z}}\}), [t_y/t_x] \prec \vdash [y/x](w![\tilde{z}])}\ \text{(T-REN)}$$

(where $[t_y/t_x, y/x](\tilde{v}{:}\tilde{T})$ is defined as $\Sigma\{[y/x]v_1{:}[t_y/t_x]T_1, \ldots, [y/x]v_n{:}[t_y/t_x]T_n\}$). By the side conditions of (T-OUT), we have $[\tilde{s_z}/\tilde{u_z}] \prec \subseteq \prec$. By $\{t_x, t_y\} \cap \{\tilde{u}_z\} = \emptyset$

(which follows from $\tilde{u_z} \in \mathbf{T_I}$ and $t_x, t_y \in \mathbf{T_O}$),

$$\begin{array}{rcl} [t_y/t_x] \prec & \supseteq & [t_y/t_x][\tilde{s_z}/\tilde{u_z}] \prec \\ & = & [([t_y/t_x]\tilde{s_z})/\tilde{u_z}]([t_y/t_x] \prec). \end{array}$$

Therefore, the derivation can be replaced by

$$\frac{}{[t_y/t_x, y/x](\Gamma_1 + w : (!^m[\tilde{\rho_z}^{\tilde{u_z}}]^{s_w}) + \Sigma\{\tilde{z} : \tilde{\rho_z}^{\tilde{s_z}}\}), [t_y/t_x] \prec \vdash [y/x](w![\tilde{z}])} \text{ (T-OUT)}$$

—Case where (T-REN) is applied below (T-IN): The derivation must be of the form

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma_1, \tilde{z} : \tilde{\rho_z}^{\tilde{s_z}}, \prec_1 \vdash P}}{\Gamma_1 + w : ?^*[\tilde{\rho_z}^{\tilde{u_z}}]^{s_w}, \prec_1 \downarrow_{\{\tilde{s}_z\}} \vdash w?[\tilde{z}].\, P} \text{ (T-IN)}}{[t_y/t_x, y/x](\Gamma_1 + w : ?^*[\tilde{\rho_z}^{\tilde{u_z}}]^{s_w}), [t_y/t_x](\prec_1 \downarrow_{\{\tilde{s}_z\}}) \vdash [y/x](w?[\tilde{z}].\, P)} \text{ (T-REN)}$$

We show that it can be replaced by

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma_1, \tilde{z} : \tilde{\rho_z}^{\tilde{s_z}}, \prec_1 \vdash P}}{[t_y/t_x, y/x]\Gamma_1, \tilde{z} : \tilde{\rho_z}^{\tilde{s_z}}, [t_y/t_x] \prec_1 \vdash [y/x]P} \text{ (T-REN)}}{[t_y/t_x, y/x]\Gamma_1 + [t_y/t_x, y/x](w : ?^*[\tilde{\rho_z}^{\tilde{u_z}}]^{s_w}), ([t_y/t_x] \prec_1) \downarrow_{\{\tilde{s_z}\}} \vdash [y/x](w?[\tilde{z}].\, P)}$$

where the last step uses the rule (T-IN). Let $\prec_2 = [t_y/t_x] \prec_1$. Then, it suffices to check

(1) $\prec_2^+$ is a strict partial order,
(2) $[\tilde{u_z}/\tilde{s_z}] \prec_2 \subseteq \prec_2$,
(3) $\{\tilde{u_z}\} \not\prec_2 \not\succ \{\tilde{s_z}\}$, and
(4) $([t_y/t_x] \prec_1) \downarrow_{\{\tilde{s_z}\}} = [t_y/t_x](\prec_1 \downarrow_{\{\tilde{s_z}\}})$.

By the conditions on the original derivation, we have

$[\tilde{u_z}/\tilde{s_z}] \prec_1 \subseteq \prec_1$;
$\{\tilde{u_z}\} \not\prec_1 \not\succ \{\tilde{s_z}\}$;
$([t_y/t_x](\prec_1 \downarrow_{\{\tilde{s_z}\}}))^+$ is a strict partial order; and
$\{t_x, t_y\} \cap \{\tilde{u_z}, \tilde{s_z}\} = \emptyset$,

from which the third and fourth conditions immediately follow. We also have the second condition by

$$[t_y/t_x] \prec_1 \supseteq [t_y/t_x][\tilde{u_z}/\tilde{s_z}] \prec_1 = [\tilde{u_z}/\tilde{s_z}][t_y/t_x] \prec_1 .$$

Suppose $\prec_2^+$ is not a strict partial order. Then, by

$$\begin{array}{rcl} [\tilde{u_z}/\tilde{s_z}] \prec_2 & = & [t_y/t_x][\tilde{u_z}/\tilde{s_z}] \prec_1 \\ & \subseteq & [t_y/t_x](\prec_1 \downarrow_{\{\tilde{s_z}\}}), \end{array}$$

$([t_y/t_x](\prec_1 \downarrow_{\{\tilde{s_z}\}}))^+$ is not a strict partial order either: a contradiction. Thus we have shown the first condition.

—Case where (T-REN) is applied below (T-NEW-1): The derivation must be of the form

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, x:\rho^{t_x}, z:\rho_z^s, \prec_1 \vdash P}}{\Gamma, x:\rho^{t_x}, \prec_1 \Downarrow_{\{s\}} \vdash (\nu z:\rho_z^s)\, P}\ \text{(T-NEW-1)}}{\Gamma + y:\rho^{t_y}, [t_y/t_x](\prec_1 \Downarrow_{\{s\}}) \vdash [y/x](\nu z:\rho_z^s)\, P}\ \text{(T-REN)}$$

We show that it can be replaced by

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, x:\rho^{t_x}, z:\rho_z^s, \prec_1 \vdash P}}{\Gamma + y:\rho^{t_y}, z:\rho_z^s, [t_y/t_x]\ \prec_1 \vdash [y/x]P}\ \text{(T-REN)}}{\Gamma + y:\rho^{t_y}, ([t_y/t_x]\ \prec_1)\Downarrow_{\{s\}} \vdash (\nu z:\rho_z^s)\, [y/x]P}\ \text{(T-NEW-1)}$$

Notice that $([t_y/t_x]\ \prec_1)\Downarrow_{\{s\}} = [t_y/t_x](\prec_1 \Downarrow_{\{s\}})$ and $(\nu z:\rho_z^s)\,[y/x]P = [y/x]((\nu z:\rho_z^s)\,P)$. So, it is sufficient to show that $([t_y/t_x]\ \prec_1)^+$ is a strict partial order. Suppose it is not the case. Then either (1) $s \prec_1 s$ holds, or (2) $(([t_y/t_x]\ \prec_1)\Downarrow_{\{s\}})^+$ is not a strict partial order. In either case, it contradicts with the fact that both $\prec_1$ and $([t_y/t_x]\ \prec_1)\Downarrow_{\{s\}}$ are tag orderings. Therefore, $([t_y/t_x]\ \prec_1)^+$ must be a strict partial order. □

A.2 Proof of Lemma 4.1.2

Proof. This is proved by induction on the proof of $P \equiv Q$. The base cases (the leaves of the proof) consist of the four rules for structural congruence, plus reflexivity. The induction steps (inner nodes of the proof) arise from the definition of structural congruence as a congruence relation; they include rules allowing structural congruence to be applied to each subterm of each process constructor, plus symmetry and transitivity. Most cases are trivial: for example, the cases for the rules (S-COMMUT) and (S-ASSOC) follow from commutativity and associativity of "+." We present only the cases for the rule (S-EXTR) (in both directions).

Let $P = (\nu x:\rho^s)\, P_1 \mid P_2$, $Q = (\nu x:\rho^s)\,(P_1 \mid P_2)$, where x is not free in P_2, and suppose that $\Gamma, \prec \vdash (\nu x:\rho^s)\, P_1 \mid P_2$. Then there are some Γ_1, Γ_2 such that $\Gamma_1, \prec \vdash (\nu x:\rho^s)\, P_1$ and $\Gamma_2, \prec \vdash P_2$, with $\Gamma_1 + \Gamma_2 = \Gamma$. $\Gamma_1, \prec \vdash (\nu x:\rho^s)\, P_1$ must be derived by either (T-NEW-1) or (T-NEW-2).

—If it is derived by (T-NEW-1), then $\Gamma_1, x:\rho^s, \prec_1 \vdash P_1$ for some $\prec_1$ such that $\prec_1 \Downarrow_{\{s\}} = \prec$. By the side condition of (T-NEW-1), $\{s\} \not\asymp \mathbf{T_I}$, which implies $(\prec \setminus \prec_1) \subseteq \mathbf{T_O} \times \mathbf{T_O}$. So by Lemma 3.11, we have $\Gamma_1, x:\rho^s, \prec_1 \cup \prec \vdash P_1$ and $\Gamma_1, \prec_1 \cup \prec \vdash P_2$. By (T-PAR) and (T-NEW-1), we have $\Gamma, (\prec_1 \cup \prec)\Downarrow_{\{s\}} (= \prec) \vdash (\nu x:\rho^s)\,(P_1 \mid P_2)$.

—If $\Gamma_1, \prec \vdash (\nu x:\rho^s)\, P_1$ is derived by (T-NEW-2), it must be that $\Gamma_1, x:\rho^s, \prec \vdash P_1$. By (T-PAR) and (T-NEW-2), we have $\Gamma, \prec \vdash (\nu x:\rho^s)\,(P_1 \mid P_2)$.

On the other hand, suppose that $\Gamma, \prec \vdash (\nu x:\rho^s)\,(P_1 \mid P_2)$. It must be derived by either (T-NEW-1) or (T-NEW-2)

—If it is derived by (T-NEW-1), then $\Gamma, x:\rho^s, \prec_1 \vdash P_1 \mid P_2$ for some $\prec_1$ such that $\prec_1 \Downarrow_{\{s\}} = \prec$. By (T-PAR), there are $\Gamma_1, \Gamma_2, \rho_1$ such that $\Gamma_1, x:\rho_1^s, \prec_1 \vdash P_1$ and $\Gamma_2, \prec_1 \vdash P_2$, with $(\Gamma_1, x:\rho_1^s) + \Gamma_2 = \Gamma, x:\rho^s$. By Lemmas 3.8 and 3.9, $\Gamma_1, x:\rho^s, \prec_1 \vdash P_1$ and $\Gamma_2 \backslash x, \prec_1 \vdash P_2$. By $\{s\} \not\asymp \mathbf{T_I}$, $(\prec \setminus \prec_1) \subseteq \mathbf{T_O} \times \mathbf{T_O}$. We therefore obtain $\Gamma_2 \backslash x, \prec \vdash P_2$ by Lemmas 3.10 and 3.11. On the other hand, by

$\Gamma_1, x:\rho^s, \prec_1 \vdash P_1$ and Lemma 3.11, we also obtain $\Gamma_1, x:\rho^s, \prec_1 \cup \prec \vdash P_1$. By (T-NEW-1) and (T-PAR), we have $\Gamma, \prec \vdash (\nu x:\rho^s)\, P_1 \mid P_2$.

—If it is derived by (T-NEW-2), then $\Gamma, x:\rho^s, \prec \vdash P_1 \mid P_2$. By (T-PAR), there are some Γ_1, Γ_2 such that $\Gamma_1, \prec \vdash P_1$ and $\Gamma_2, \prec \vdash P_2$, with $\Gamma_1 + \Gamma_2 = \Gamma, x : \rho^s$. From the fact that x is not free in P_2 and Lemmas 3.8 and 3.9, we can assume $\Gamma_1(x) = \rho^s$ and $x \notin dom(\Gamma_2)$. By (T-NEW-2) and (T-PAR), we obtain $\Gamma_1 \backslash x + \Gamma_2, \prec \vdash (\nu x{:}\rho^s)\, P_1 \mid P_2$, that is, $\Gamma, \prec \vdash (\nu x{:}\rho^s)\, P_1 \mid P_2$. □

A.3 Proof of Subject Reduction Theorem (Theorem 4.3.1)

After proving several lemmas in A.3.1, we show the theorem by induction on the derivation of $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$.

A.3.1 *Lemmas.* The following two lemmas are used in the induction step for the rule (R-PAR).

LEMMA A.3.1.1. *If* $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$ *and* $\Gamma + \Delta$ *is well defined, then* $\Gamma + \Delta \vdash P \xrightarrow{m} P' \dashv \Gamma' + \Delta$.

PROOF. The proof follows by induction on the derivation of $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. The base cases (cases for the rules (R-COM) and (R-RCOM)) follow from the fact that if $\updownarrow^m[\tilde{T}]^t + S$ is well defined, then $Rem(\updownarrow^m[\tilde{T}]^t + S) = Rem(\updownarrow^m[\tilde{T}]^t) + S$. Induction steps are straightforward. □

LEMMA A.3.1.2. *If* $\Gamma_1 + \Gamma_2 \vdash P \xrightarrow{m} P' \dashv \Gamma'$ *and* $\Gamma_1, \prec \vdash P$, *then* $\Gamma_1 \vdash P \xrightarrow{m} P' \dashv \Gamma_1'$ *and* $\Gamma' = \Gamma_1' + \Gamma_2$ *for some* Γ_1'.

PROOF. This is proved by induction on the derivation of $\Gamma_1 + \Gamma_2 \vdash P \xrightarrow{m} P' \dashv \Gamma'$. Because the other cases are trivial, we show only the case for (R-PAR). In this case, $P = P_1 \mid P_2$, $P' = P_1' \mid P_2$, and $\Gamma_1 + \Gamma_2 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma'$. By $\Gamma_1, \prec \vdash P_1 \mid P_2$ and (T-PAR), we have

$$\begin{array}{l} \Gamma_3 + \Gamma_4 = \Gamma_1 \\ \Gamma_3, \prec \vdash P_1 \\ \Gamma_4, \prec \vdash P_2 \end{array}$$

for some Γ_3 and Γ_4. Since $\Gamma_1 + \Gamma_2 = \Gamma_3 + (\Gamma_4 + \Gamma_2)$, $\Gamma_3 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_3'$ and $\Gamma_3' + (\Gamma_4 + \Gamma_2) = \Gamma'$ for some Γ_3' by induction hypothesis. By Lemma A.3.1.1, $\Gamma_3 + \Gamma_4 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_3' + \Gamma_4$. Let $\Gamma_1' = \Gamma_3' + \Gamma_4$. Then, $\Gamma_1 \vdash P_1 \mid P_2 \xrightarrow{m} P' \dashv \Gamma_1'$ with $\Gamma_1' + \Gamma_2 = \Gamma'$. □

The following lemma, together with Lemma 3.9, plays a key role in the base cases for (R-COM) and (R-RCOM).

LEMMA A.3.1.3. *Let* $\prec$ *be a strict partial order, and* $\tilde{s}, \tilde{t}, \tilde{u}$ *time tags. If* $([\tilde{u}/\tilde{s}]\prec) \subseteq \prec$, $([\tilde{t}/\tilde{u}](\prec\downarrow_{\{\tilde{s}\}}) \subseteq \prec\downarrow_{\{\tilde{s}\}}$, *and* $\{\tilde{u}\} \not\prec\not> \{\tilde{s}\}$, *then* $([\tilde{t}/\tilde{s}]\prec) = \prec\downarrow_{\{\tilde{s}\}}$.

PROOF. $\prec\downarrow_{\{\tilde{s}\}} \subseteq [\tilde{t}/\tilde{s}]\prec$ follows from $\prec\downarrow_{\{\tilde{s}\}} = [\tilde{t}/\tilde{s}](\prec\downarrow_{\{\tilde{s}\}}) \subseteq [\tilde{t}/\tilde{s}]\prec$.
On the other hand, by $\{\tilde{u}\} \not\prec\not> \{\tilde{s}\}$, we have $\prec = \prec\downarrow_{\{\tilde{u}\}} \cup \prec\uparrow_{\{\tilde{u}\}} \subseteq \prec\downarrow_{\{\tilde{u}\}} \cup \prec\downarrow_{\{\tilde{s}\}}$.

Therefore,

$$\begin{array}{rcl}[\tilde{t}/\tilde{s}]\prec & \subseteq & [\tilde{t}/\tilde{s}](\prec\downarrow_{\{\tilde{u}\}}) \cup [\tilde{t}/\tilde{s}](\prec\downarrow_{\{\tilde{s}\}}) \\ & = & [\tilde{t}/\tilde{u}]([\tilde{u}/\tilde{s}](\prec\downarrow_{\{\tilde{u}\}})) \cup (\prec\downarrow_{\{\tilde{s}\}}) \\ & \subseteq & ([\tilde{t}/\tilde{u}](\prec\downarrow_{\{s\}})) \cup (\prec\downarrow_{\{\tilde{s}\}}) \\ & \subseteq & \prec\downarrow_{\{\tilde{s}\}} . \end{array}$$

□

A.3.2 *Proof of Theorem 4.3.1*

PROOF. This is proved by induction on the derivation of $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$, with case analysis on the last rule used. When $\Gamma = \Gamma', x : T$, we write $\Gamma - x$ for $\Gamma', x : \mathit{Rem}(T)$.

—Case (R-COM): In this case, $P = x![\tilde{y}] \mid x?[\tilde{z}].\, Q$, and $P' = [\tilde{y}/\tilde{z}]Q$. By (T-PAR) and (T-OUT), we have

$$\begin{array}{l}\Gamma_1 \text{ is unlimited} \\ \Gamma_2, \prec \vdash x?[\tilde{z}].\, Q \\ \Gamma_1 + x : !^m[\tilde{\rho}^{\tilde{u}}]^{t_x} + \tilde{y}{:}\tilde{\rho}^{\tilde{t}} + \Gamma_2 = \Gamma \\ \Gamma' = \Gamma - x \\ t_x \prec^\sharp \Sigma\{\tilde{y}{:}\tilde{\rho}^{\tilde{t}}\} \\ ([\tilde{t}/\tilde{u}]\ \prec) \subseteq \prec \end{array}$$

for some Γ_1, Γ_2. Without loss of generality (by Lemmas 3.8 and 3.9), we can assume that $\Gamma_1 = \emptyset$ and $x : !^m[\tilde{\rho}^{\tilde{u}}]^{t_x} + \tilde{y}{:}\tilde{\rho}^{\tilde{t}} + \Gamma_2 = \Gamma$. We show $\Gamma', \prec \vdash P'$ by case analysis on m.

—Case $m = 1$ or ω: In this case, by (T-IN),

$$\begin{array}{l}\Gamma_2 = x : ?^m[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Gamma_3 \\ \Gamma_3, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_1 \vdash Q \\ ([\tilde{u}/\tilde{t_z}]\ \prec_1) \subseteq \prec_1 \\ \{\tilde{u}\} \not\prec_1 \not\succ \{\tilde{t_z}\} \\ \prec_1\downarrow_{\{\tilde{t_z}\}} = \prec \end{array}$$

for some $\Gamma_3, \prec_1$. Note that

$$\begin{array}{rcl}\Gamma' & = & \Gamma - x \\ & = & (x : !^m[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}) + (\Gamma_3 + x : ?^m[\tilde{\rho}^{\tilde{u}}]^{t_x}) - x \\ & = & (x : \updownarrow^m[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_3) - x \\ & = & x : \mathit{Rem}(\updownarrow^m[\tilde{\rho}^{\tilde{u}}]^{t_x}) + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_3. \end{array}$$

The above equation also implies that $\Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_3$ is well defined. By Lemma A.3.1.3, we have $[\tilde{t}/\tilde{t_z}]\ \prec_1 = \prec$. Therefore, by repeated applications of Lemma 3.13 to $\Gamma_3, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_1 \vdash Q$, we have

$$\Gamma_3 + \Sigma\{\tilde{y}{:}\tilde{\rho}^{t}\}, \prec \vdash [\tilde{y}/\tilde{z}]Q.$$

By weakening (Lemma 3.9),

$$\Gamma_3 + \Sigma\{\tilde{y}{:}\tilde{\rho}^{t}\} + x : \mathit{Rem}(\updownarrow^m[\tilde{\rho}^{\tilde{u}}]^{t_x}), \prec \vdash [\tilde{y}/\tilde{z}]Q,$$

i.e., $\Gamma', \prec \vdash P'$.

—Case $m = M$: In this case, by (T-MIN),

$$\begin{array}{l}\Gamma_2 = \Gamma_3 + x : ?^M[\tilde{\rho}^{\tilde{u}}]^{t_x}\\ \Gamma_3, x : p^M[\tilde{\rho}^{\tilde{u}}]^{t_x}, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_1 \vdash Q\\ O \in p\\ ([\tilde{u}/\tilde{t_z}]\ \prec_1) \subseteq \prec_1\\ \{\tilde{u}\} \not\prec_1 \not\succ \{\tilde{t_z}\}\\ \prec_1 \downarrow_{\{\tilde{t_z}\}} = \prec\end{array}$$

for some Γ_3. Note that

$$\begin{array}{rcl}\Gamma' &=& \Gamma - x = \Gamma\\ &=& (x : !^M[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}) + (\Gamma_3 + x : ?^M[\tilde{\rho}^{\tilde{u}}]^{t_x})\\ &=& x : \updownarrow^M[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_3.\end{array}$$

The above equation also implies that $x : p^M[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_3$ is well defined. By Lemma A.3.1.3, we have $[\tilde{t}/\tilde{t_z}]\ \prec_1 = \prec$. Therefore, by repeated applications of Lemma 3.13 to $\Gamma_3, x : p^M[\tilde{\rho}^{\tilde{u}}]^{t_x}, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_1 \vdash Q$, we have

$$\Gamma_3 + x : p^M[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}, \prec \vdash [\tilde{y}/\tilde{z}]Q.$$

By weakening (Lemma 3.9),

$$\Gamma_3 + x : \updownarrow^M[\tilde{\rho}^{\tilde{u}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}, \prec \vdash [\tilde{y}/\tilde{z}]Q,$$

i.e., $\Gamma', \prec \vdash P'$.

—Case (R-RCOM): In this case, $P = x![\tilde{y}] \,|\, x?^*[\tilde{z}].\, Q$, and $P' = [\tilde{y}/\tilde{z}]Q' \,|\, x?^*[\tilde{z}].\, Q$, with $Q' = \mathit{rename}(Q)$. By replacing time tags in the derivation of $\Gamma, \prec \vdash P$, we also have the derivation of

$$\Gamma, [\tilde{s'}/\tilde{s}] \prec \vdash x![\tilde{y}] \,|\, x?^*[\tilde{z}].\, Q'$$

where $\{\tilde{s}\} = \mathit{New}(Q)$ and $\tilde{s'}$ are the corresponding time tags in Q' ($\{\tilde{s'}\} = \mathit{New}(Q')$). Let $\prec' = \prec_{\tilde{s} \approx \tilde{s'}} = ([\tilde{s'}/\tilde{s}]\ \prec)_{\tilde{s'} \approx \tilde{s}}$. Then by Lemma 3.12,

$$\Gamma, \prec' \vdash x![\tilde{y}] \,|\, x?^*[\tilde{z}].\, Q'.$$

By (T-PAR), (T-OUT), and (T-RIN),

$$\begin{array}{l}\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star, \prec' \vdash x?^*[\tilde{z}].\, Q'\\ \Gamma_2, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_2 \vdash Q'\\ (x : !^*[\tilde{\rho}^{\tilde{u}}]^\star + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}) + (\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star) = \Gamma\\ \Gamma' = \Gamma - x = \Gamma\\ ([\tilde{t}/\tilde{u}]\ \prec') \subseteq \prec'\\ \{\tilde{t_z}\} \not\prec_2 \not\succ \{\tilde{u}\}\\ [\tilde{u}/\tilde{t_z}]\ \prec_2 \subseteq \prec_2\\ \prec_2 \downarrow_{\{\tilde{t_z}\}} = \prec'\end{array}$$

for some unlimited type environment Γ_2.
By Lemma A.3.1.3, $\prec' = [\tilde{t}/\tilde{t_z}]\ \prec_2$. So by repeated applications of Lemma 3.13 to $\Gamma_2, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_2 \vdash Q'$, we obtain $\Gamma_2 + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}, \prec' \vdash [\tilde{y}/\tilde{z}]Q'$.

On the other hand, by replacing $\tilde{s'}$ with $\tilde{s}$ in the derivation of $\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star, \prec' \vdash x?^*[\tilde{z}].\, Q'$, we also have

$$\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star, \prec' \vdash x?^*[\tilde{z}].\, Q.$$

By (T-PAR),

$$(\Gamma_2 + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}) + (\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star), \prec' \vdash [\tilde{y}/\tilde{z}]Q' \mid x?^*[\tilde{z}].\, Q.$$

By Lemma 3.9 and $\Gamma' = \Gamma_2 + x : \updownarrow^*[\tilde{\rho}^{\tilde{u}}]^\star + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}$, we have $\Gamma', \prec' \vdash P'$.

—Case (R-PAR): In this case, $P = P_1 \mid P_2$, $P' = P_1' \mid P_2$, and $\Gamma \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma'$. Moreover,

$$\begin{array}{l} \Gamma_1, \prec \vdash P_1 \\ \Gamma_2, \prec \vdash P_2 \\ \Gamma_1 + \Gamma_2 = \Gamma \end{array}$$

for some Γ_1, Γ_2. By Lemma A.3.1.2, $\Gamma_1 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_1'$ for some Γ_1', and $\Gamma' = \Gamma_1' + \Gamma_2$. By induction hypothesis, $\Gamma_1', \prec_{\tilde{s} \approx \tilde{s'}} \vdash P_1'$ for $\{\tilde{s'}\} = New(P_1') \backslash New(P_1)$ and some $\{\tilde{s}\} \subseteq New(P_1)$. By repeated applications of Lemma 3.12, $\Gamma_2, \prec_{\tilde{s} \approx \tilde{s'}} \vdash P_2$. By (T-PAR), $\Gamma_1' + \Gamma_2, \prec_{\tilde{s} \approx \tilde{s'}} \vdash P_1' \mid P_2$, i.e., $\Gamma', \prec_{\tilde{s} \approx \tilde{s'}} \vdash P'$. Note that $New(P') \backslash New(P) = New(P_1') \backslash New(P_1) = \{\tilde{s'}\}$ and $\{\tilde{s}\} \subseteq New(P_1) \subseteq New(P)$.

—Case (R-NEW): In this case, $P = (\nu x : \rho^t)\, Q$, $P' = (\nu x : \rho'^t)\, Q'$, and $\Gamma, x : \rho^t \vdash Q \xrightarrow{m} Q' \dashv \Gamma', x : \rho'^t$. $\Gamma, \prec \vdash P$ can be derived by either (T-NEW-1) or (T-NEW-2).

—Case for (T-NEW-1): $\Gamma, x : \rho^t, \prec_1 \vdash Q$ for some $\prec_1$ such that $\prec_1 \Downarrow_{\{t\}} = \prec$. By Lemma 3.11, $\Gamma, x : \rho^t, \prec_1 \cup \prec \vdash Q$. By induction hypothesis, $\Gamma', x : \rho'^t, (\prec_1 \cup \prec)_{\tilde{s} \approx \tilde{s'}} \vdash Q'$ for $\{\tilde{s'}\} = New(Q') \backslash New(Q) (= New(P') \backslash New(P))$ and $\{\tilde{s}\} \subseteq New(Q) (\subseteq New(P))$. By (T-NEW-1), we have $\Gamma', ((\prec_1 \cup \prec)_{\tilde{s} \approx \tilde{s'}}) \Downarrow_{\{t\}} \vdash P'$. Note that $((\prec_1 \cup \prec)_{\tilde{s} \approx \tilde{s'}}) \Downarrow_{\{t\}} = \prec_{\tilde{s} \approx \tilde{s'}}$.

—Case for (T-NEW-2): $\Gamma, x : \rho^t, \prec \vdash Q$. By induction hypothesis, $\Gamma', x : \rho'^t, \prec_{\tilde{s} \approx \tilde{s'}} \vdash Q'$ for $\{\tilde{s'}\} = New(Q') \backslash New(Q) (= New(P') \backslash New(P))$ and $\{\tilde{s}\} \subseteq New(Q) (\subseteq New(P))$. By (T-NEW-2), $\Gamma', \prec_{\tilde{s} \approx \tilde{s'}} \vdash P'$.

—Cases (R-IFT) and (R-IFF) are trivial.

—Case (R-CONG) follows immediately from Lemma 4.1.2. □

B. RULES FOR TYPE CHECK

This appendix presents modified typing rules in order to clarify the first phase of the type check algorithm explained informally in Section 5.

In order to make the flow of type information in the algorithm more explicit, we consider a type judgment of the form $\Gamma_1, \prec \vdash P :: \Gamma_2$. A type environment Γ_1 represents capabilities that *may* be consumed by the process P, while Γ_2 represent capabilities that are actually consumed by P. Each of the new typing rules will express how Γ_2 can be computed from inputs Γ_1 and P. For example, the rule for parallel composition is given as follows:

$$\frac{\Gamma_1, \prec \vdash P_1 :: \Gamma_2 \qquad \Gamma_1 - \Gamma_2, \prec \vdash P_2 :: \Gamma_3}{\Gamma_1, \prec \vdash P_1 \mid P_2 :: \Gamma_2 + \Gamma_3}$$

Here $\Gamma_1 - \Gamma_2$, defined below, represents a type environment obtained from Γ_1 by removing capabilities in Γ_2.

We need several definitions before presenting the rules.

Definition B.1. The binary operation $-$ on types is defined by

$$\begin{array}{rcl} b - b & ::= & b \\ p^m[\tilde{T}] - q^m[\tilde{T}] & := & r^m[\tilde{T}] \text{ if } p \supseteq q \text{ and} \\ & & \quad r \text{ is the largest set s.t. } r^m[\tilde{T}] + q^m[\tilde{T}] = p^m[\tilde{T}]. \end{array}$$

In other cases, $T_1 - T_2$ is undefined. $-$ is extended to the operation on type environments: $\Gamma_1 - \Gamma_2$ is defined as follows, only if $dom(\Gamma_1) \supseteq dom(\Gamma_2)$ and $\Gamma_1(x) - \Gamma_2(x)$ is defined for each $x \in dom(\Gamma_2)$.

$$\begin{array}{l} dom(\Gamma_1 - \Gamma_2) = dom(\Gamma_1) \\ (\Gamma_1 - \Gamma_2)(x) = \begin{cases} \Gamma_1(x) - \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2) \end{cases} \end{array}$$

Definition B.2. The binary operation $\sqcup$ on types is defined by

$$\begin{array}{rcl} b \sqcup b & ::= & b \\ p^m[\tilde{T}] \sqcup q^m[\tilde{T}] & := & (p \cup q)^m[\tilde{T}]. \end{array}$$

In other cases, $T_1 \sqcup T_2$ is undefined. $\sqcup$ is extended to the operation on type environments: $\Gamma_1 \sqcup \Gamma_2$ is defined as follows, only if $\Gamma_1(x) \sqcup \Gamma_2(x)$ is defined for each $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$.

$$\begin{array}{l} dom(\Gamma_1 \sqcup \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2) \\ (\Gamma_1 \sqcup \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \sqcup \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \backslash dom(\Gamma_1) \end{cases} \end{array}$$

Notation B.3. We write $\Gamma \backslash S$ for the type environment Δ such that $dom(\Delta) = dom(\Gamma) \backslash S$ and $\Delta(x) = \Gamma(x)$ for each $x \in dom(\Delta)$. $\Gamma|_S$ represents the type environment Δ such that $dom(\Delta) = S \cap dom(\Gamma)$ and $\Delta(x) = \Gamma(x)$ for each $x \in dom(\Delta)$.

The new typing rules are given below. In each type judgment $\Gamma_1, \prec \vdash P :: \Gamma_2$ in the rules, we implicitly assume that $\Gamma_1 - \Gamma_2$ is well defined. Although the assumption is redundant for Theorem B.4, it is necessary in the actual type-checking algorithm: for example, in (TC-IN), the assumption is used to determine $m, \tilde{\rho}, t, etc.$

$$\frac{\Gamma_1, \prec \vdash P_1 :: \Gamma_2 \qquad \Gamma_1 - \Gamma_2, \prec \vdash P_2 :: \Gamma_3}{\Gamma_1, \prec \vdash P_1 \mid P_2 :: \Gamma_2 + \Gamma_3} \qquad \text{(TC-PAR)}$$

$$\frac{\begin{array}{c} s \prec^\sharp \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} \\ ([\tilde{t}/\tilde{u}] \prec) \subseteq \prec \end{array}}{\Gamma, \prec \vdash x![\tilde{y}] :: x : (!^m[\tilde{\rho}^{\tilde{u}}]^s) + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}} \qquad \text{(TC-OUT)}$$

$$\frac{\begin{array}{c}\Gamma, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P :: \Delta \\ (\tilde{z} : \tilde{\rho}^{\tilde{s}}) - (\Delta|_{\{\tilde{z}\}}) \text{ unlimited} \\ t \prec^\sharp \Delta \backslash \{\tilde{z}\} \\ ([\tilde{u}/\tilde{s}]\ \prec) \subseteq \prec \qquad \{\tilde{s}\} \not\prec\not> \{\tilde{u}\} \\ m = 1 \text{ or } \omega\end{array}}{\Gamma, \prec\downarrow_{\{\tilde{s}\}} \vdash x?[\tilde{z}].\, P :: \Delta\backslash\{\tilde{z}\} + x : ?^m[\tilde{\rho}^{\tilde{u}}]^t} \quad \text{(TC-IN)}$$

$$\frac{\begin{array}{c}\Gamma\backslash\{x\}, x : \updownarrow^M[\tilde{\rho}^{\tilde{u}}]^t, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P :: \Delta \\ (x : \updownarrow^M[\tilde{\rho}^{\tilde{u}}]^t, \tilde{z} : \tilde{\rho}^{\tilde{s}}) - (\Delta|_{\{x,\tilde{z}\}}) \text{ unlimited} \\ t \prec^\sharp \Delta \backslash \{x, \tilde{z}\} \\ ([\tilde{u}/\tilde{s}]\ \prec) \subseteq \prec \qquad \{\tilde{s}\} \not\prec\not> \{\tilde{u}\}\end{array}}{\Gamma, \prec\downarrow_{\{\tilde{s}\}} \vdash x?[\tilde{z}].\, P :: \Delta\backslash\{x, \tilde{z}\}, x : ?^M[\tilde{\rho}^{\tilde{u}}]^t} \quad \text{(TC-MIN)}$$

$$\frac{\begin{array}{c}\Gamma, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P :: \Delta \qquad \Delta\backslash\{\tilde{z}\} \text{ unlimited} \\ (\tilde{z} : \tilde{\rho}^{\tilde{s}}) - (\Delta|_{\{\tilde{z}\}}) \text{ unlimited} \\ ([\tilde{u}/\tilde{s}]\ \prec) \subseteq \prec \qquad \{\tilde{s}\} \not\prec\not> \{\tilde{u}\}\end{array}}{\Gamma, \prec\downarrow_{\{\tilde{s}\}} \vdash x?^*[\tilde{z}].\, P :: \Delta\backslash\{\tilde{z}\} + x : ?^*[\tilde{\rho}^{\tilde{u}}]^\star} \quad \text{(TC-RIN)}$$

$$\frac{\begin{array}{c}\Gamma, x : p^m[\tilde{T}]^s, \prec \vdash P :: \Delta \\ p \text{ is either } \updownarrow \text{ or } | \\ x : p^m[\tilde{T}]^s - \Delta|_{\{x\}} \text{ unlimited} \\ \{s\} \not\prec\not> \mathbf{T_I}\end{array}}{\Gamma, \prec \Downarrow_{\{s\}} \vdash (\nu x : p^m[\tilde{T}]^s)\, P :: \Delta\backslash\{x\}} \quad \text{(TC-NEW-1)}$$

$$\frac{\begin{array}{c}\Gamma, x : p^m[\tilde{T}]^s, \prec \vdash P :: \Delta \\ p \text{ is either } \updownarrow \text{ or } | \\ x : p^m[\tilde{T}]^s - \Delta|_{\{x\}} \text{ unlimited}\end{array}}{\Gamma, \prec \vdash (\nu x : p^m[\tilde{T}]^s)\, P :: \Delta\backslash\{x\}} \quad \text{(TC-NEW-2)}$$

$$\frac{\begin{array}{c}\Gamma, \prec \vdash P_1 :: \Delta_1 \qquad \Gamma, \prec \vdash P_2 :: \Delta_2 \\ (\Delta_1 \sqcup \Delta_2) - \Delta_1 \text{ unlimited}\end{array}}{\Gamma, \prec \vdash \mathit{if}\ x\ \mathit{then}\ P_1\ \mathit{else}\ P_2 :: (\Delta_1 \sqcup \Delta_2) + x : \mathit{bool}^\star} \quad \text{(TC-IF)}$$

We explain how to read a few of the above rules in the first phase of type check. The rule (TC-PAR) means that in order to check $P_1 \mid P_2$ under a type environment Γ_1, we should first check P_1 under Γ_1, and obtain Γ_2; then we can check P_2 under $\Gamma_1 - \Gamma_2$. The rule (TC-IN) means that in order to check $x?[\tilde{z}].\, P$ under Γ, we should first check P under $\Gamma, \tilde{z} : \tilde{\rho}^{\tilde{s}}$ and obtain Δ (here $\tilde{\rho}$ is known by looking at the binding of x in Γ); then we should check that $(\tilde{z} : \tilde{\rho}^{\tilde{s}}) - (\Delta|_{\{\tilde{z}\}})$ is unlimited (which means the obligations on $\tilde{z}$ are fulfilled in P) and output $\Delta\backslash\{\tilde{z}\} + x : ?^m[\tilde{\rho}^{\tilde{u}}]^t$.

It is a routine to check that the new rules are essentially equivalent to the original rules.

THEOREM B.4. *If $\Gamma, \prec \vdash P$, then $\Gamma, \prec \vdash P :: \Delta$ for some Δ such that $\Gamma - \Delta$ is unlimited. On the other hand, if $\Gamma, \prec \vdash P :: \Delta$ for some Δ such that $\Gamma - \Delta$ is unlimited, then $\Gamma, \prec \vdash P$.*

ACKNOWLEDGMENTS

The author would like to thank Benjamin C. Pierce and David N. Turner for joyful discussions and a number of useful suggestions. Among others, Benjamin C. Pierce suggested that we consider polymorphism on time tags, and David N. Turner suggested encoding of call-by-need λ-calculus. Their questions also helped us improve understanding of our type system. The author would also like to thank Nobuko Yoshida, Davide Sangiorgi, Simon Gay, and Uwe Nestmann for discussions on the relationships between their work and ours. Special thanks are also due to anonymous referees, for their very useful comments.

REFERENCES

Berger, M., Gay, S., and Nagarajan, R. 1997. A typed calculus of deadlock-free processes. Draft paper, Imperial College, London.

Colby, C. 1995. Analyzing the communication topology of concurrent programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, New York, 202–213.

Gay, S. J. 1993. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 429–438.

Hodas, J. S. and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput. 110,* 2, 327–365.

Honda, K. and Yoshida, N. 1995. On reduction-based process semantics. *Theor. Comput. Sci. 151,* 2, 437–486.

Igarashi, A. and Kobayashi, N. 1997. Type-based analysis of usage of communication channels for concurrent programming languages. In *Proceedings of International Static Analysis Symposium (SAS'97)*. Lecture Notes in Computer Science, vol. 1302. Springer-Verlag, Berlin, 187–201.

Kobayashi, N. 1996a. Concurrent linear logic programming. Ph.D. thesis, Dept. of Information Science, Univ. of Tokyo, Japan.

Kobayashi, N. 1996b. A partially deadlock-free typed process calculus (I)—a simple system. Tech. Rep. 96-02, Dept. of Information Science, Univ. of Tokyo, Japan.

Kobayashi, N. and Yonezawa, A. 1995. Towards foundations for concurrent object-oriented programming—types and language design. *Theory Pract. Object Syst. 1,* 4, 243–268.

Kobayashi, N., Pierce, B. C., and Turner, D. N. 1996. Linearity and the pi-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 358–371.

Mackie, I. 1994. Lilac: A functional programming language based on linear logic. *J. Funct. Program. 4,* 4 (Oct.), 1–39.

Milner, R. 1990. Function as processes. In *Automata, Language and Programming*. Lecture Notes in Computer Science, vol. 443. Springer-Verlag, Berlin, 167–180.

Milner, R. 1993. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Springer-Verlag, Berlin.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, I, II. *Inf. Comput. 100*, 1–77.

Nestmann, U. 1997. What is a 'good' encoding of guarded choice? In *Proceedings of EXPRESS'97*. Electronic Notes in Theoretical Computer Science, vol. 7. Elsevier Science Publishers, Amsterdam.

Nielson, H. R. and Nielson, F. 1994. Higher-order concurrent programs with finite communication topology. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 84–97.

Ostheimer, G. and Davie, A. J. T. 1993. π-calculus characterizations of some practical λ-calculus reduction strategies. Tech. rep., St. Andrews Univ., Scotland.

Oyama, Y., Taura, K., and Yonezawa, A. 1997. An efficient compilation framework for languages based on a concurrent process calculus. In *Proceedings of Euro-Par '97 Parallel Processing, Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 1300. Springer-Verlag, Berlin, 546–553.

Peyton Jones, S. 1996. Concurrent Haskell. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 295–308.

Pierce, B. and Sangiorgi, D. 1993. Typing and subtyping for mobile processes. In *Proceedings of IEEE Symposium on Logic in Computer Science*. IEEE, New York, 376–385.

Pierce, B. C. and Sangiorgi, D. 1997. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 242–255.

Pierce, B. C. and Turner, D. N. 1995. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP)*. Lecture Notes in Computer Science, vol. 907. Springer-Verlag, Berlin, 187–215.

Pierce, B. C. and Turner, D. N. 1997. Pict: A programming language based on the pi-calculus. Tech. rep., Computer Science Dept., Indiana Univ. To appear in Milner *Festschrift*, MIT Press, 1997.

Reppy, J. H. 1991. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*. ACM Press, New York, 293–305.

Sangiorgi, D. 1992. Expressing mobility in process algebras: First-order and higher-order paradigms. Ph.D. thesis, Univ. of Edinburgh, Edinburgh, Scotland.

Sangiorgi, D. 1997. The name discipline of uniform receptiveness (extended abstract). In *Proceedings of ICALP'97*. Lecture Notes in Computer Science, vol. 1256. Springer-Verlag, Berlin, 303–313.

Shapiro, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3 (Sept.), 413–510.

Turner, D. T. 1996. The polymorphic pi-calculus: Theory and implementation. Ph.D. thesis, Univ. of Edinburgh, Edinburgh, Scotland.

Vasconcelos, V. T. and Honda, K. 1993. Principal typing schemes in a polyadic π-calculus. In *Proceedings of CONCUR'93*. Lecture Notes in Computer Science, vol. 715. Springer-Verlag, Berlin, 524–538.

Walker, D. 1995. Objects in the π-calculus. *Inf. Comput. 116*, 253–271.

Yonezawa, A. and Tokoro, M. 1987. *Object-Oriented Concurrent Programming*. The MIT Press, Cambridge, Mass.

Yoshida, N. 1996. Graph types for monadic mobile processes. In *Proceedings of FST/TCS'16*. Lecture Notes in Computer Science, vol. 1180. Springer-Verlag, Berlin, 371–387. Full version as LFCS report, ECS-LFCS-96-350, Univ. of Edinburgh.

Received June 1997; accepted September 1997