TECHNICAL REPORT 97-02

A Partially Deadlock-free Typed Process Calculus (II)

Naoki Kobayashi

February 1997

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TITLE
A Partially Deadlock-free Typed Process Calculus (II)

AUTHORS
Naoki Kobayashi

KEY WORDS AND PHRASES
process calculus, type system, deadlock, linear type, concurrent objects

ABSTRACT
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 order of channel use as type information, and (2) classification of communication channels into reliable and unreliable channels based on their usage, with a guarantee of the usage by the type system. We can ensure that communications on reliable channels never cause deadlock, and also that certain reliable channels never introduce non-determinism. With the type system, for example, simply typed λ-calculus can be encoded into the deadlock-free and confluent fragment of our process calculus, thus, we can recover behavior of the typed λ-calculus in the level of process calculi. We also show that typical concurrent objects are also encoded into the deadlock-free fragment.

REPORT DATE	**WRITTEN LANGUAGE**
February 1997	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
40	23

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
A summary of this paper will appear in Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS'97).

DISTRIBUTION STATEMENT
First issue 35 copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

A Partially Deadlock-free Typed Process Calculus (II)

Naoki Kobayashi
Department of Information Science
University of Tokyo
email:koba@is.s.u-tokyo.ac.jp

1 Introduction

Backgrounds and Problems Various concurrent programming languages [22, 24, 19, 7, 18, 15] have recently been proposed and are attracting a great deal of attention because of their usefulness in describing symbolic/numeric computations for parallel/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 along those channels, and concurrent execution of processes – which have recently been studied extensively through process calculi [12, 20, 16, 4, 8]. They have been found to be expressive enough to provide higher-level mechanisms such as concurrent objects [9, 17]. On the other hand, such concurrency primitives are too powerful and low-level, and they suffer from 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 [19], a term of a function type may not behave like a function: it may return non-deterministic 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.

Our approach The goal of this paper 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 non-determinism: 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 the type system. To illustrate these ideas, we consider the following asynchronous process calculus (which is a subset of the calculus we consider in this paper):

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

Here $\tilde{y}$ abbreviates a sequence $y_1, \ldots, y_n$. As in the π-calculus [11], 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, $x?[\,].y![\,]$ 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 channel type constructor and keep ordering between them. So in our type system, the above type judgement 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 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 x 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 $\{(s_y, s_z)\}$ ($\mathbf{0}$ represents an inaction), 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 from deadlock completely. For example, in spite of the cyclic dependency between the uses of x and y, the process $P = x?[z].(y![\,] \mid z![\,]) \mid y?[\,].x![w]$ does not necessarily fall into deadlock: if there is a process $x![w] \mid w?[\,].x?[z].\mathbf{0}$ in parallel to P, then they can be reduced to $\mathbf{0}$ without deadlock. On the other hand, though 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 the latter kind of process, we need to require, in addition to the lack of cyclic dependency, that every process which communicates on a certain channel can eventually find its communication partner. We consider three kinds of such channels (which we call *reliable channels*): linear, replicated input, and mutex channels. For other kinds of channels (*unreliable channels*), we give up ensuring deadlock-freedom. A channel is *linear* [8] if it is used just once for sending a message and just once for receiving a message. A *replicated input channel* must appear just once in a replicated input prefix ($x?^*[\tilde{y}].\,$), but can be used an arbitrary number of times for sending messages. A *mutex channel* is similar to a binary semaphore: a single message must be sent after it is created, and a process that has extracted a value from the channel must eventually put a value into the channel. Such usage of channels is ensured by our static type system.

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 send/receive a message along unreliable channels). As will later be demonstrated by the encoding of λ-calculus and concurrent objects, large parts of actual concurrent programs are written with only reliable channels. For example, functions can be implemented by using linear and replicated input channels, and typical concurrent objects can be implemented by using linear, replicated input, and mutex channels. This means that we can reason from type information that concurrent processes that implement functions and typical concurrent objects never fall into deadlock: in other words, the evaluation of those processes never gets stuck. In addition, because communication on a linear or replicated input channel does not introduce nondeterminism, we also know from type information that a unique result (up to some typed process congruence [8]) is returned by the process that implements a function.

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. We also show that every well-typed term of call-by-value, call-by-name, and call-by-need simply typed λ-calculus is encoded into a well-typed process of our process calculus without using unreliable channels. Therefore, by applying the theorem on partial deadlock-freedom, we can reason only from type information that the evaluation of those processes never gets stuck. This kind of reasoning has 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 programmers need not supply an ordering).

From a pragmatic point of view, the reader may think that if we introduce functions and concurrent objects as primitives, it is not necessary to reason that certain processes behave like functions, etc. However, we believe that such reasoning is important especially in parallel/distributed environments for the following reasons.

- Because we need various kinds of concurrent objects – from simple ones to complex ones supporting transactions, internal concurrency in an object, etc. – depending on each application program, even if we provide some kinds of concurrent objects as primitives, we should still provide a programmer with a way to realize appropriate concurrent objects in terms of low-level communication primitives. (Otherwise, a programmer must use primitive concurrent objects that do not fit the requirement of an application program, which causes a program to be unnecessarily complicated and/or inefficient.) It is desirable that we can do the same reasoning about behavior of concurrent objects implemented by programmers just as we do about primitive concurrent objects.

- Suitable implementation of functions may also vary depending on application programs and environments, because of locations of data, parallelism to be pursued, etc. Therefore, we should again provide a programmer with a way to optimize their functions in terms of low-level communication primitives. So, it is again desirable that we can do the same reasoning about behavior of functions implemented by programmers just as we do about primitive functions.

- Even if we do not provide low-level concurrency primitives in high-level languages, we need

them in implementation languages. Reasoning about behavior of programs at the level of implementation languages is important for various optimizations and also for ensuring correctness of compilation.

Potential Applications In addition to the static detection of deadlock, there are several interesting potential applications outside the scope of this paper. Because an ordering $\prec$ in a judgement $x_1 : T_1, \ldots, x_n : T_n, \prec \vdash P$ 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 communicated along each channel. Therefore, our ordering information would provide a better specification for module systems of concurrent programming languages, and would also be useful for efficient implementation of concurrency primitives.

Structure of the Paper The rest of this paper 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 partial deadlock-freedom. Section 5 informally presents a polynomial-time type check algorithm for our type system. Section 6 shows encodings of functions and concurrent objects into the deadlock-free fragment of our process calculus. Section 8 discusses related work and Section 9 concludes this paper.

2 Notational Preliminaries

This section introduces the syntax of types, expressions, and type judgements. If types are ignored, our process calculus can be considered a subset of the polyadic π-calculus [11]. We dropped synchronous outputs, summations (choices), and matchings from the π-calculus, and we restricted replications to be only performed to input prefixes. We borrow the syntax used in the linear channel type system [8], which is similar to that of Pict [18]. In the course of introducing the notations, we also sharpen the intuitions given in Section 1.

2.1 Types

For simplicity in this paper, we consider channel types and base types as primary type constructors. $p^m[T_1, \ldots, T_n]^t$ (abbreviated as $p^m[\tilde{T}]^t$) represents a 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.

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*. 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 ($x?^*[\tilde{y}^{\tilde{s}}].$), and an arbitrary number of times in output expressions ($x![\tilde{y}]$), but cannot be used in any input prefix ($x?[\tilde{y}^{\tilde{s}}].$).

- If $m = M$, types are called *mutex channel types* and channels *mutex channels*. For a mutex channel, a single message must be put after its creation, and a process which retrieves a message from it must eventually sends it a single message. (Thus, a mutex channel is similar to 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 channel, a replicated input channel, or a mutex channel. The reason 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 (i.e., find partners of communication) unless the whole program falls into an infinite loop or a deadlock caused by bad use of unreliable channels.

Polarities We need to detect which (input or output) operations may be performed by each process, in order to ensure, for example, that there is no case where more than one process try to send a message along a linear channel. A polarity p is used to keep this information. A polarity ranges over subsets of input/output capabilities $\{I, O\}$. 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. A 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 obligation to send some value back to the channel.

The above usage of channels can perhaps 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 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 does not necessarily need to receive a value from the channel. Similarly, if a process holds a replicated input channel of type $\updownarrow^*[\tilde{T}]$, it has 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.

Time tags A time tag $s(, t, u, \ldots)$ intuitively describes *when* a channel of the type can be used for communication. As is indicated in Section 1, we will later associate time tags with a binary relation $\prec$. So, "x has a type $?^1[int]^{t_1}$, y has a type $!^1[int]^{t_2}$ and $t_1 \prec t_2$" means that x is first used for receiving an integer, *and then* y is used for sending an integer. Such intuition works only for understanding time tags attached to linear channels; a more correct intuition for $t_1 \prec t_2$ would be "a process may be blocked on a channel whose time tag is t_1 *before* it fulfills an *obligation* to perform communication on a channel whose time tag is t_2." In other words, time tags and a relation between them control "when obligation on communication should be fulfilled." Thus, if t_x is a time tag attached to the type of a mutex channel x and t_y is another time tag attached to the type of y, then

- If y is a linear channel, $t_x \prec t_y$ implies that a process may try to receive a message from x (i.e., acquire a lock x) before communicating along 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 some process tries to communicate along y before releasing a lock x (i.e., sending a message along x). For example, if y is also a mutex channel, a process $x?[\,].\,y?[\,].\,(x![\,] \mid y![\,])$ is allowed only if $t_y \prec t_x$ holds, since the process tries to communicate on the channel y before fulfilling the obligation to send a value to the channel x. Note that, in this case, it need not be that $t_x \prec t_y$, because receiving a value from the mutex channel y is not an obligation.

The set of time tags, written $\mathbf{T}^+$, includes a special tag $\star$. We write $\mathbf{T}$ for $\mathbf{T}^+ \backslash \{\star\}$. Replicated input channel types and unreliable channel types are always annotated with the special tag $\star$, in order to ignore when channels of those types are used. We could keep ordering on time tags attached to replicated input channels and give the same intuition "$t_x \prec t_y$ means a process may be blocked on the channel with time tag t_x before fulfilling the obligation on the channel with time tag t_y," where the obligation is to repeatedly receive values (by replicated input prefix) from the channel in the case of replicated input channels. We, however, do not so in this paper because keeping ordering on replicated input channels would damage the expressive power of the deadlock-free fragment of the calculus.[1] Instead, we take a more trivial approach to deadlock-freedom; we ensure (by typing rules and the definition of $\prec^\sharp$ given later) that "the input end of replicated input channel must immediately be bound to a replicated input prefix after its creation." In other words, "obligation on a replicated input channel must immediately be fulfilled after its creation."

Types Now we are ready to formally define types. We use the word *bare type* for a type whose outermost time tag is dropped, and sometimes call a type (with time tags) a *tagged type* in order to distinguish it with a bare type.

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

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

where $p^m[\tilde{T}]$ 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 includes '*bool*.'

[1] This is due to a very delicate reason which is hard to explain; we ourselves did not realize it until we had checked proofs of deadlock-freedom in details. It might be possible to keep ordering on replicated input channels by changing some typing conditions.

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 $\rho_1^{s_1}, \ldots, \rho_n^{s_n}$. The type $p^m[\tilde{T}]^t$ is called *even* if p is $|$ or $\updownarrow$, and *odd* otherwise. The reason for the restriction that output-only or input-only linear channels cannot be passed along mutex channels is closely related to what we mean by "deadlock-freedom." For mutex channels, we want to ensure that a receiver is not blocked forever, but we do allow a sender to be blocked forever (i.e., a value put into a mutex channel may never be retrieved). It is because a mutex channel is used as a reference cell extended with a mutual exclusion mechanism, and because the content of a message just corresponds to a stored value. Since it is not guaranteed that a value put into a mutex channel will eventually be extracted, we cannot put a channel containing some obligation. For example, let x be a mutex channel to send/receive the output end of a linear channel, and y a linear channel; if we removed the restriction, $x![y] \,|\, y?[\,].\mathbf{0}$ would be well typed but in deadlock (if there is no receiver on x in parallel to this process).

Because we must be very sensitive to when and how often each channel is used, the weakening is usually not allowed in typing rules. For example, if a process is well typed under a type environment $x : \updownarrow^1[int]$, we cannot replace the environment with $x : \updownarrow^1[int], y : !^1[int]$, because the environment $x : \updownarrow^1[int]$ tells us that y is not used at all in the process, which contradicts with $x : \updownarrow^1[int], y : !^1[int]$. Types for which weakening is allowed are called "*unlimited*" types. Intuitively, a type is unlimited if values of the type can appear an arbitrary number of places (in other words, can be "used" an arbitrary number of times "as a value of the type"; if the type is $|^m[\tilde{T}]$, this condition is vacuously true). Note that it depends on the polarity and the multiplicity whether the given type is unlimited or not. For example, for a replicated input channel, its output end can be used an arbitrary number of times, but its input end cannot.

2.2 Definition: We say that a type T is *unlimited* if

- T is a base type or a unreliable channel type,
- $T = p^*[\tilde{T}]^\star$ and $I \notin p$,
- $T = p^M[\tilde{T}]^t$ and $O \notin p$, or
- $T = |^1[\tilde{T}]^t$

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

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

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

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

2.4 Definition [removal of capability]: Let T be a type. The type $Rem(T)$ is defined as:

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

2.2 Process Expressions

Here we add some annotations and conditional expressions to the syntax used in Section 1.

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

P	::=	$P_1 \mid P_2$	(concurrent execution of P_1 and P_2)
		$x![\tilde{y}]$	(sends $\tilde{y}$ along the channel x)
		$x?[\tilde{z}^{\tilde{s}}].\,P$	(receives values $\tilde{v}$ along x and behaves like $[\tilde{v}/\tilde{z}]P$)
		$x?^*[\tilde{z}^{\tilde{s}}].\,P$	(repeatedly receives values $\tilde{v}$ along x and spawns the process $[\tilde{v}/\tilde{z}]P$)
		$(\nu x : T)\,P$	(creates a channel x of type T and executes P)
		$\mathit{if}\ x\ \mathit{then}\ P_1\ \mathit{else}\ P_2$	(conditional expression)

where $\tilde{y}$ abbreviates a sequence $y_1, \ldots, y_n$ (n may be 0), and $\tilde{z}^{\tilde{s}}$ abbreviates $z_1^{s_1}, \ldots, z_n^{s_n}$.

As usual, we assume that $\tilde{z}$ are bound by prefixes $x?[\tilde{z}^{\tilde{s}}].$ and $x?^*[\tilde{z}^{\tilde{s}}].$ and that x is bound by $(\nu x : T)$. We identify expressions up to 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 $(x?[\,].((\nu y : T)\,x![\,])) \mid y![\,]$. Time tags $\tilde{s}$ in (replicated) input prefixes can be ignored for the moment; they are used only when we state the conditions of well-formed type judgements. When they are not important, we just write $x?[\tilde{z}].\,P$ for $x?[\tilde{z}^{\tilde{s}}].\,P$. 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 $\mathbf{0}$ for $(\nu x : \updownarrow^M [\,]^t)(x![\,])$ where t is a fresh time tag.

We write $Nu_{\mathbf{T}}(P)$ for the set of time tags t of ν-prefixes $(\nu x : \rho^t)$ in P, $In_{\mathbf{T}}(P)$ for the set of time tags appearing in input and replicated input prefixes in P, and $\mathbf{IT}(P)$ for the set of time tags appearing in ρ of ν-prefixes $(\nu x : \rho^t)$ in P. They are formally defined by:

$$
\begin{array}{lcl}
Nu_{\mathbf{T}}(P_1 \mid P_2) & = & Nu_{\mathbf{T}}(P_1) \cup Nu_{\mathbf{T}}(P_2) \\
Nu_{\mathbf{T}}(x![\tilde{y}]) & = & \emptyset \\
Nu_{\mathbf{T}}(x?[\tilde{y}^{\tilde{s}}].\,P) & = & Nu_{\mathbf{T}}(P) \\
Nu_{\mathbf{T}}(x?^*[\tilde{y}^{\tilde{s}}].\,P) & = & Nu_{\mathbf{T}}(P) \\
Nu_{\mathbf{T}}((\nu x : \rho^s)\,P) & = & (\{s\} \backslash \{\star\}) \cup Nu_{\mathbf{T}}(P) \\
Nu_{\mathbf{T}}(\mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2) & = & Nu_{\mathbf{T}}(P_1) \cup Nu_{\mathbf{T}}(P_2) \\
In_{\mathbf{T}}(P_1 \mid P_2) & = & In_{\mathbf{T}}(P_1) \cup In_{\mathbf{T}}(P_2) \\
In_{\mathbf{T}}(x![\tilde{y}]) & = & \emptyset \\
In_{\mathbf{T}}(x?[\tilde{y}^{\tilde{s}}].\,P) & = & (\{\tilde{s}\} \backslash \{\star\}) \cup In_{\mathbf{T}}(P) \\
In_{\mathbf{T}}(x?^*[\tilde{y}^{\tilde{s}}].\,P) & = & (\{\tilde{s}\} \backslash \{\star\}) \cup In_{\mathbf{T}}(P) \\
In_{\mathbf{T}}((\nu x : \rho^s)\,P) & = & In_{\mathbf{T}}(P) \\
In_{\mathbf{T}}(\mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2) & = & In_{\mathbf{T}}(P_1) \cup In_{\mathbf{T}}(P_2) \\
\mathbf{IT}(P_1 \mid P_2) & = & \mathbf{IT}(P_1) \cup \mathbf{IT}(P_2) \\
\mathbf{IT}(x![\tilde{y}]) & = & \emptyset \\
\mathbf{IT}(x?[\tilde{y}^{\tilde{s}}].\,P) & = & \mathbf{IT}(P) \\
\mathbf{IT}(x?^*[\tilde{y}^{\tilde{s}}].\,P) & = & \mathbf{IT}(P) \\
\mathbf{IT}((\nu x : \rho^s)\,P) & = & \{t \mid t \in \mathbf{T} \text{ and } t \text{ appears in } \rho\} \cup \mathbf{IT}(P) \\
\mathbf{IT}(\mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2) & = & \mathbf{IT}(P_1) \cup \mathbf{IT}(P_2)
\end{array}
$$

For example, if $P = (\nu x : \updownarrow^1[T]^{t_x})\,x?[y^{s_y}].\,z?[w^{s_z}].\,x![w]$, then $Nu_{\mathbf{T}}(P) = \{t_x\}, In_{\mathbf{T}}(P) = \{s_y, s_z\}$.

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

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

2.8 Example: The process

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

corresponds to a definition of the factorial function. (For simplicity, we assume that we have $\times, =$ on integers as primitives in this example.) $P \mid fact![2, y]$ is reduced as follows:

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

2.3 Type Judgement Form

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

2.3.1 Type Environments

Type environments and operations on them are defined almost in the same manner as in the linear channel type system [8].

2.9 Definition: A type environment is a mapping from a finite set of variables to types.

We often use metavariables Γ, Δ for type environments. We write $dom(\Gamma)$ for the domain of Γ. If $true, false \in dom(\Gamma)$, $\Gamma(true)$ and $\Gamma(false)$ must be *bool*. 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 define the set of "outer time tags" $\mathbf{OT}(\Gamma)$ by $\mathbf{OT}(\tilde{x} : \tilde{\rho}^{\tilde{s}}) = \{\tilde{s}\} \backslash \{\star\}$, and the set of "inner time tags" $\mathbf{IT}(\Gamma)$ by $\mathbf{IT}(\tilde{x} : \tilde{\rho}^{\tilde{s}}) = \{t \mid t \text{ appears in } \tilde{\rho}\}$. We often write $\mathbf{IT}(\Gamma, P)$ for $\mathbf{IT}(\Gamma) \cup \mathbf{IT}(P)$. We say Γ is *unlimited* if $\Gamma(x)$ is unlimited for all $x \in dom(\Gamma)$.

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

$$dom(\Gamma_1 + \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$$

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

It is trivial from the definition that $+$ is associative and commutative. We sometimes write $\Sigma\{\Gamma_1, \ldots, \Gamma_n\}$ for $\Gamma_1 + \cdots + \Gamma_n$.

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

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

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: $\forall t.\neg(t\mathcal{R}t)$).[2] We write $t_1 \prec t_2$ when $(t_1, t_2) \in \prec$, $t_1 = \star$, or $t_2 = \star$.

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 a channel whose time tag is 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 an unlimited type cannot represent any obligation (because it is not ensured that a value of the unlimited type will be used).

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

$$
\begin{array}{ll}
t \prec^\sharp \emptyset & \\
t \prec^\sharp x : b^\star & \\
t \prec^\sharp x : p^m[\tilde{T}]^s \text{ iff } & (t \prec s \wedge m \neq *) \\
& \vee (m = \omega) \\
& \vee (m = 1 \wedge (p = | \vee p = \updownarrow)) \\
& \vee (m = * \wedge I \notin p) \\
& \vee (m = M \wedge O \notin p) \\
t \prec^\sharp \Gamma, x : T \text{ if } & (t \prec^\sharp \Gamma) \wedge (t \prec^\sharp x : T)
\end{array}
$$

Careful readers may notice 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 $t \prec^\sharp x : T$ never holds if a type T contains the obligation to receive a value from a replicated input channel regardless of the relation $\prec$. This means that a process must never try to communicate on a channel before using the input end of a replicated input channel.

We introduce several notations on tag orderings.

2.14 Definition: Let $\prec \subseteq \mathbf{T} \times \mathbf{T}$ and $S_1, S_2 \subseteq \mathbf{T}^+$. 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.

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

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

2.16 Definition: 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:

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

[2] For some very delicate technical reason, we cannot let $\prec$ itself to be a strict partial order.

2.3.3 Type Judgement Form

A type judgement is of the form $\Gamma, \prec \vdash P$, where Γ is a type environment and $\prec$ is a tag ordering. We require that all the time tags in $\mathbf{OT}(\Gamma), Nu_{\mathbf{T}}(P), In_{\mathbf{T}}(P)$ are unique; i.e., each $t \in \mathbf{OT}(\Gamma) \cup Nu_{\mathbf{T}}(P) \cup In_{\mathbf{T}}(P)$ has just a single occurrence in $\Gamma, \prec \vdash P$. (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. As may already be seen from examples, the ordering $\prec$ specifies an *upper-bound* of ordering of channel use in P, rather than an 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, but it can never happen that communication on x is blocked by that on y as in $y?[n].x![n]$. Note alsothat we may have 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 along x before it sends an integer along y.

3 Typing Rules

We give typing rules and mainly explain the conditions on tag orderings below. The conditions on type environments are basically the same as those for the linear channel type system and are explained there in detail [8]. (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. This 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, P_2 are respectively well typed under type environments $\Gamma_1 = x : ?^1[\ ]^{t_x}, y : !^1[\ ]^{t_y}, \Gamma_2 = x : !^1[\ ]^{t_x}, y : ?^1[\ ]^{t_y}$, 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 opposite 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 $\tilde{y}$ are sent along x via an expression $x![\tilde{y}]$, $\tilde{y}$ can be used only after they are received by communication on 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 $u \prec^{\sharp} \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}$ in the rule (T-Out). The condition: $([\tilde{t}/\tilde{s}]\prec) \subseteq \prec$, means that if there is an ordering on $\tilde{s}$ in $\prec$, then there must also be the corresponding ordering on $\tilde{t}$. It ensures that a receiver can use $\tilde{y}$ safely in an order specified by the ordering on $\tilde{s}$.

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

In the following rule for input prefix, the condition $[\tilde{t}/\tilde{s}]\prec \subseteq \prec$ ensures that $\tilde{z}$ must not be used in P in an order unspecified by the ordering on $\tilde{t}$. Since the body of an input expression can be executed only after communication on x is completed, we need the condition $u \prec^{\sharp} \Gamma$ to allow

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 via the ordering on $\tilde{t}$ (so, the annotation $\tilde{s}$ in an input prefix does not play any essential role and they can be renamed with other fresh time tags; it is kept just to avoid their reuse in the succeeding derivations).

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

The rule (T-MIn) for input prefixes on mutex channels is almost identical to (T-In), except that a message must be sent to x in the body of the input prefix (which is ensured by the condition $O \in p$).

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

The following rule (T-RIn) is the most tricky. It is similar to (T-In) but it has the following differences:

- The environment Γ on the free variables used in the body of a replicated input expression must be unlimited because the body may be executed more than once.
- The time tags of some channels created in the body can be ignored in the succeeding derivations (because the use of those channels is local).

$$\frac{\begin{array}{c}\Gamma, \tilde{z} : \tilde{\rho}^{\tilde{s}}, \prec \vdash P \qquad \Gamma \text{ unlimited} \\ S \subseteq Nu_{\mathbf{T}}(P) \\ S \not\prec\not> \mathbf{T}\backslash(\mathbf{OT}(\Gamma) \cup Nu_{\mathbf{T}}(P) \cup \{\tilde{s}\}) \\ ([\tilde{t}/\tilde{s}](\prec \Downarrow_S)) \subseteq \prec \qquad \{\tilde{s}\} \not\prec\not> \{\tilde{t}\}\end{array}}{\Gamma + x : ?^*[\tilde{\rho}^{\tilde{t}}]^\star, (\prec \Downarrow_S)\downarrow_{\{\tilde{s}\}} \vdash x?^*[\tilde{z}^{\tilde{s}}].\, P} \qquad \text{(T-RIn)}$$

The rule for channel creation moves the corresponding binding from the type environment to the ν-prefix. (Note that the ordering on s below cannot be removed from $\prec$ at this moment.)

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

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

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

We give several examples of valid/invalid type judgements below.

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

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

but

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

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

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

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

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

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

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

Thus, we obtain

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

The condition $t_x \prec t_u \prec t_v$ implies that x is first used for communication, then u is used, and then v is used. Note that we can *infer* the relation $\prec$ from the given type environment and process. The subexpression $x?[y^{s_y}, z^{s_z}].y?[n^\star].z![n]$ requires $u_y \prec u_z$. The fact $u_y \prec u_z$ and the subexpression $x![v, w]$ require $t_x \prec t_v, t_w$ and $[t_v/u_y, t_w/u_z]\prec \subseteq \prec$, which imply $t_x \prec t_v \prec t_w$. Such reconstruction of a strict partial order is discussed later in Section 5.

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

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

3.5 Example: Let
$P = f?^*[b, r^s].$
$\quad (\mathit{if}\ b\ \mathit{then}\ r![\,]\ \mathit{else}\ (\nu r' : \updownarrow^1[\,]^{t_r})(f![b, r'] \,|\, r'?[\,].r![\,])).$
(P is a typical form of recursive function definitions.) From (T-RIn) and

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

we obtain $f : \updownarrow^*[bool^\star, !^1[\,]^u]^\star, \emptyset \vdash P$.

The following example explains why we need the complex side condition:

$$S \not\asymp \mathbf{T} \backslash (\mathbf{OT}(\Gamma) \cup Nu_{\mathbf{T}}(P) \cup \{\tilde{s}\})$$

in (T-RIn) to forget ordering on some time tags in $Nu_{\mathbf{T}}(P)$.

3.6 Example: Let

$$P = (\nu x : \updownarrow^1[]^{t_x})(m?[y^{s_y}].y?[].x![] \mid n![x])$$
$$\Gamma = m : \updownarrow^\omega[?^1[]^{u_y}]^\star, n : \updownarrow^\omega[!^1[]^{u_x}]$$
$$\prec = \{(u_y, t_x)\}.$$

Then, $\Gamma, \prec \vdash P$. Consider $f?^*[].P$. By (T-RIN), we obtain

$$\Gamma, f : ?^*[]^\star, \prec \vdash f?^*[].P,$$

but cannot obtain

$$\Gamma, f : ?^*[]^\star, \prec \Downarrow_{\{t_x\}} (= \emptyset) \vdash f?^*[].P$$

by the side condition of (T-RIN). Note that if we allowed

$$\Gamma, f : ?^*[]^\star, \emptyset \vdash f?^*[].P,$$

we could have

$$\Gamma, f : \updownarrow^*[]^\star, \emptyset \vdash f?^*[].P \mid f![] \mid n?[z].m![z].$$

But $f?^*[].P \mid f![] \mid n?[z].m![z]$ is reduced to $f?^*[].P \mid (\nu x : \updownarrow^1[]^{t_x})(x?[].x![])$, which is in a deadlock state caused by the linear channel x.

Technical properties

We show several properties of our type system. Because those properties are only used in the proofs of the main theorems (Theorem 4.5 and 4.9) given in Section 4, readers who are not interested in the proofs can safely skip the rest of this section.

3.7 Lemma [Strengthening]: If $\Gamma, x : T, \prec \vdash P$ and x is not free in P, then T is unlimited and $\Gamma, \prec \vdash P$.

Proof: From 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 $\Gamma, x : T \vdash P$ by dropping all occurrences of $x : T$. □

3.8 Lemma [Weakening]: Let Γ be a type environment and T 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: Straightforward induction on typing derivations. □

3.9 Lemma [Order Weakening(1)]: Suppose $\Gamma, \prec \vdash P$. $\Gamma, \prec \cup \prec' \vdash P$ holds if $\prec'$ satisfies the following conditions.

- $(\prec \cup \prec')^+$ is a strict partial order, and
- $\prec' \uparrow_{\mathbf{IT}(\Gamma, P) \cup S} = \emptyset$,
  where S is the set of time tags excluded from the domain of $\prec$ in the derivation of $\Gamma, \prec \vdash P$ (so, S includes $In_{\mathbf{T}}(P)$ and some of $Nu_{\mathbf{T}}(P)$).[3]

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 $\prec_1 \cup \prec'$. Note that side conditions in the new derivation immediately follow from the corresponding side conditions in the original derivation. □

[3] We could formally define the set S by induction on the derivation of $\Gamma, \prec \vdash P$. We skip it since it should be clear.

3.10 Lemma [Order Weakening(2)]: Suppose $\Gamma, \prec \vdash P$, and $\tilde{u'}$ are fresh time tags, and $\tilde{u} \subseteq Nu_{\mathbf{T}}(P)$. Then, $\Gamma, \prec \cup[\tilde{u'}/\tilde{u}]\ \prec \vdash P$.

Proof: It is trivial that $(\prec \cup[\tilde{u'}/\tilde{u}]\ \prec)^+$ is a strict partial order. The derivation of $\Gamma, \prec \cup[\tilde{u'}/\tilde{u}]\ \prec \vdash P$ 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 $\prec_1 \cup[\tilde{u'}/\tilde{u}]\ \prec$. Note that side conditions in the new derivation immediately follow from the corresponding side conditions in the original derivation. □

3.11 Lemma: 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, from $\{\tilde{u}\} \not\prec\not> \{\tilde{s}\}$, we have

$$\prec = \prec\downarrow_{\{\tilde{u}\}} \cup \prec\uparrow_{\{\tilde{u}\}} \subseteq \prec\downarrow_{\{\tilde{u}\}} \cup \prec\downarrow_{\{\tilde{s}\}}.$$

Therefore,

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

□

3.12 Lemma [Renaming]: The rule

$$\frac{\Gamma, x : \rho^{t_x}, \prec \vdash P \qquad ([t_y/t_x]\ \prec)^+ \text{ is a strict partial order}}{\Gamma + y : \rho^{t_y}, [t_y/t_x]\ \prec \vdash [y/x]P} \qquad \text{(T-Ren)}$$

is admissible; that is, if

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

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

Proof: Follows from the fact that (T-Ren) can be permuted above each of the other rules, while (T-Ren) just below an instance of the axiom (T-Out) can be merged with the (T-Out). Because the other cases are similar and much simpler, we show only the case where (T-Ren) is applied below (T-RIn).

Suppose that a derivation is of the following form:

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma_1, \tilde{z} : \tilde{\rho_z}^{\tilde{s}}, \prec_1 \vdash P}}{\Gamma_1 + w : ?^*[\tilde{\rho_z}^{\tilde{s'}}], (\prec_1 \Downarrow_{\tilde{u}})\downarrow_{\{\tilde{s}\}} \vdash w?^*[\tilde{z}^{\tilde{s}}].P}\ \text{(T-RIn)}}{[t_y/t_x, y/x](\Gamma_1 + w : ?^*[\tilde{\rho_z}^{\tilde{s'}}]), [t_y/t_x]((\prec_1 \Downarrow_{\tilde{u}})\downarrow_{\{\tilde{s}\}}) \vdash [y/x]w?^*[\tilde{z}^{\tilde{s}}].P}\ \text{(T-Ren)}$$

(where $[t_y/t_x, y/x](\tilde{v}{:}\tilde{T})$ is defined to be $\Sigma\{[y/x]v_1{:}[t_y/t_x]T_1, \ldots, [y/x]v_n{:}[t_y/t_x]T_n\}$). From the side conditions of (T-RIn) and conditions on judgements, we have

$\{\tilde{u}\} \not\prec_1 \not\succ \mathbf{T}\backslash(\mathbf{OT}(\Gamma_1) \cup Nu_{\mathbf{T}}(P) \cup \{\tilde{s}\})$
$\{\tilde{u}\} \subseteq Nu_{\mathbf{T}}(P)$
$[\tilde{s'}/\tilde{s}](\prec_1 \Downarrow_{\{\tilde{u}\}}) \subseteq \prec_1$
$\{\tilde{s'}\} \not\prec_1 \not\succ \{\tilde{s}\}$
$\{t_x, t_y\} \cap \{\tilde{u}, \tilde{s}, \tilde{s'}\} = \emptyset$
Γ_1 unlimited

We show that the above derivation can be replaced with the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma_1, \tilde{z} : \tilde{\rho_z}^{\tilde{s}}, \prec_1 \vdash P}}{[t_y/t_x, y/x]\Gamma_1, \tilde{z} : \tilde{\rho_z}^{\tilde{s}}, [t_y/t_x]\ \prec_1 \vdash [y/x]P}\ (\text{T-Ren})}{[t_y/t_x, y/x](\Gamma_1 + w : ?^*[\tilde{\rho_z}^{\tilde{s'}}]), (([t_y/t_x]\ \prec_1)\Downarrow_{\{\tilde{u}\}})\downarrow_{\{\tilde{s'}\}} \vdash [y/x](w?^*[\tilde{z}^{\tilde{s}}].\, P)}\ (\text{T-RIn})$$

Note that $(([t_y/t_x]\ \prec_1)\Downarrow_{\{\tilde{u}\}})\downarrow_{\tilde{s'}} = [t_y/t_x]((\prec_1 \Downarrow_{\{\tilde{u}\}})\downarrow_{\{\tilde{s'}\}})$. Let $\prec_2 = [t_y/t_x]\ \prec_1$. All we must check are

- $\prec_2^+$ is a strict partial order, and
- $\prec_2$ satisfies the side conditions of (T-RIn).

The latter fact is trivial; Because $\{t_x, t_y\} \cap \{\tilde{u}, \tilde{s}, \tilde{s'}\} = \emptyset$,

- $\{\tilde{u}\} \not\prec_2 \not\succ \mathbf{T}\backslash(\mathbf{OT}([t_y/t_x, y/x]\Gamma_1) \cup Nu_{\mathbf{T}}([y/x]P) \cup \{\tilde{s}\})$ immediately follows from $\{\tilde{u}\} \not\prec_1 \not\succ \mathbf{T}\backslash(\mathbf{OT}(\Gamma_1) \cup Nu_{\mathbf{T}}(P) \cup \{\tilde{s}\})$.
- $[\tilde{s'}/\tilde{s}](\prec_2 \Downarrow_{\{\tilde{u}\}}) \subseteq \prec_2$ follows from $[\tilde{s'}/\tilde{s}](\prec_1 \Downarrow_{\{\tilde{u}\}}) \subseteq \prec_1$ and $[t_y/t_x]([\tilde{s'}/\tilde{s}](\prec_1 \Downarrow_{\{\tilde{u}\}})) = [\tilde{s'}/\tilde{s}](\prec_2 \Downarrow_{\{\tilde{u}\}})$, and
- $\{\tilde{s'}\} \not\prec_2 \not\succ \{\tilde{s}\}$ follows from $\{\tilde{s'}\} \not\prec_1 \not\succ \{\tilde{s}\}$.

Therefore, showing $\prec_2^+$ is a strict partial order completes the proof. Let $\prec_3 = \prec_2 \Downarrow_{\{\tilde{u}\}}$, and $\prec_4 = \prec_3 \downarrow_{\{\tilde{s}\}}$. Because $\prec_4^+ = ([t_y/t_x]((\prec_1 \Downarrow_{\{\tilde{u}\}})\downarrow_{\{\tilde{s}\}}))^+$ is a strict partial order, it is enough to show that

(1) if $\prec_2^+$ is not a strict partial order, then neither is $\prec_3^+$; and

(2) if $\prec_3^+$ is not a strict partial order, then neither is $\prec_4^+$.

(1) Suppose that there is a cycle

$$\alpha_1 \prec_2 \cdots \prec_2 \alpha_n \prec_2 \alpha_1.$$

Then, if $\{\alpha_1, \ldots, \alpha_n\} \subseteq \{\tilde{u}\}$, then there is also a cycle

$$\alpha_1 \prec_1 \cdots \prec_1 \alpha_n \prec_1 \alpha_1,$$

which contradicts with the fact that $\prec_1^+$ is a strict partial order. Hence a cycle must contain some time tags not contained in $\{\tilde{u}\}$. Therefore, by replacing each path of the form:

$$\alpha \prec_2 u_{i_1} \prec_2 \cdots \prec_2 u_{i_k} \prec_2 \alpha'$$

where $\alpha, \alpha' \notin \{\tilde{u}\}$ with $\alpha \prec_3 \alpha'$, we obtain a cycle:

$$\alpha_{j_1} \prec_3 \cdots \prec_3 \alpha_{j_l} \prec_3 \alpha_{j_1},$$

which implies $\prec_3^+$ is not a strict partial order.

(2) By the condition $[\tilde{s'}/\tilde{s}](\prec_2 \Downarrow_{\{\tilde{u}\}}) \subseteq \prec_2$, we have

$$\begin{array}{rcl} [\tilde{s'}/\tilde{s}] \prec_3 & = & ([\tilde{s'}/\tilde{s}](\prec_2 \Downarrow_{\{\tilde{u}\}})) \downarrow_{\{\tilde{s},\tilde{u}\}} \\ & \subseteq & \prec_2 \downarrow_{\{\tilde{s},\tilde{u}\}} \\ & \subseteq & \prec_3 \downarrow_{\{\tilde{s}\}} \\ & = & \prec_4 \end{array}$$

Therefore, if there is a cycle

$$\alpha_1 \prec_3 \cdots \prec_3 \alpha_n \prec_3 \alpha_1,$$

then we obtain a cycle

$$\alpha'_1 \prec_4 \cdots \prec_4 \alpha'_n \prec_4 \alpha'_1,$$

by replacing all the $\tilde{s}$ with $\tilde{s'}$.

□

4 Type Soundness and Partial Deadlock-Freedom

In this section, we present 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. It is ensured by the two theorems: (1) subject reduction theorem (Theorem 4.5), which states that typing is preserved by reductions, and (2) Theorem 4.9, which states that reliable channels do not cause deadlock immediately 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 [11].

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

4.1 Definition [structural congruence]: A structural congruence $\equiv$ is the least congruence closed under the following rules.

$$P \mid Q \equiv Q \mid P \qquad \text{(S-Commut)}$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad \text{(S-Assoc)}$$

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

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

4.2 Definition [normal forms]: 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.

4.3 Lemma: 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 [11, 20], but it is annotated with some extra information [8] that is used later for stating properties of the semantics. The reduction relation is of the form $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. 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. $\mathit{rename}(P)$ in the rule (R-RCom) represents a process expression obtained from P by renaming all the time tags in $In_{\mathbf{T}}(P)$ and $Nu_{\mathbf{T}}(P)$ 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.

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

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

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

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

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

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

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

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 $\Gamma \vdash P \longrightarrow Q$.

Partial Confluence

As claimed in [8], 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 show any proof here since it is the same as that in [8].

4.4 Theorem [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 $\Delta \vdash Q \xrightarrow{m} Q' \dashv \Delta'$.

4.2 Partial Deadlock Freedom

The subject reduction below ensures that well-typedness of a process expression is kept during reduction. The 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.

4.5 Theorem [subject reduction]: If $\Gamma, \prec \vdash P$ and $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$, then $\Gamma', \prec' \vdash P'$ for some $\prec'$ such that

1. $(\prec')^+$ is a strict partial order, and
2. $\prec'^+ \downarrow_{Nu_{\mathbf{T}}(P') \setminus Nu_{\mathbf{T}}(P)} = \prec^+$.

The latter condition of $\prec'$ implies that all the new orderings are on newly created time tags by (R-RCom). Note that $\prec' \neq \prec$ only when the reduction comes from communication on replicated input channels. By the proof of the theorem given later, we can construct a tag ordering $\prec'$ from the derivation of $\Gamma, \prec \vdash P$. We need several lemmas to prove Theorem 4.5.

4.6 Lemma: If $P \equiv Q$, then $\Gamma, \prec \vdash P$ iff $\Gamma, \prec \vdash Q$.

Proof: Induction on the proof of $P \equiv Q$. The base cases (the leaves of the proof) consist of the four rules for structural congruence given above, 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 $P = (\nu x{:}T)\, P_1 \mid P_2$, $Q = (\nu x{:}T)\,(P_1 \mid P_2)$, with x is not free in P_2, and suppose that $\Gamma, \prec \vdash (\nu x{:}T)\, P_1 \mid P_2$. Then there are some Γ_1, Γ_2 such that $\Gamma_1, \prec \vdash (\nu x{:}T)\, P_1$ and $\Gamma_2, \prec \vdash P_2$, with $\Gamma_1 + \Gamma_2 = \Gamma$. By (T-New) (the only rule by which $\Gamma_1, \prec \vdash (\nu x{:}T)\, P_1$ can be derived), $\Gamma_1, x{:}T, \prec \vdash P_1$. By (T-Par), $(\Gamma_1, x{:}T) + \Gamma_2, \prec \vdash P_1 \mid P_2$, that is, $\Gamma, x{:}T, \prec \vdash P_1 \mid P_2$ (because we can assume that $x \notin dom(\Gamma_2)$ by appropriate renaming of bound variables). By (T-New), $\Gamma, \prec \vdash (\nu x{:}T)\,(P_1 \mid P_2)$.

On the other hand, suppose that $\Gamma, \prec \vdash (\nu x{:}T)\,(P_1 \mid P_2)$. Then $\Gamma, x{:}T, \prec \vdash P_1 \mid P_2$. 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 $\Gamma_1 + \Gamma_2 = \Gamma, x : T$. From the fact that x is not free in P_2 and Lemmas 3.8 and 3.7, we can assume that $\Gamma_1(x) = T, x \notin dom(\Gamma_2)$. By (T-New) and (T-Par), we obtain $\Gamma_1 \backslash x + \Gamma_2, \prec \vdash (\nu x{:}T)\, P_1 \mid P_2$, that is, $\Gamma, \prec \vdash (\nu x{:}T)\, P_1 \mid P_2$. □

4.7 Lemma: 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: Induction on the proof 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. □

4.8 Lemma: 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: Induction on the proof 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 $\Gamma_1 + \Gamma_2 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma'$. From $\Gamma_1, \prec \vdash P_1 \mid P_2$, we have

$\Gamma_3 + \Gamma_4 = \Gamma_1$
$\Gamma_3, \prec \vdash P_1$
$\Gamma_4, \prec \vdash P_2$

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 4.7, $\Gamma_3 + \Gamma_4 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_3' + \Gamma_4$. Let $\Gamma_1' = \Gamma_3' + \Gamma_4$. Then, $\Gamma_1 \vdash P_1 \mid P_2 \xrightarrow{m} P' \dashv \Gamma_1'$ with $\Gamma_1' + \Gamma_2 = \Gamma'$. □

Proof of Theorem 4.5: We show a slightly stronger property. Suppose $\Gamma, \prec \vdash P$ and $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. We show either $\Gamma', \prec \vdash P$ or $\Gamma', \prec \cup([\tilde{u'}/\tilde{u}]\ \prec) \cup \prec' \vdash P'$ for some $\prec', \tilde{u}, \tilde{u'}$ such that

1. $(\prec \cup([\tilde{u'}/\tilde{u}]\ \prec) \cup \prec')^+$ is a strict partial order,
2. $(\prec \cup([\tilde{u'}/\tilde{u}]\ \prec) \cup \prec')^+ \downarrow_{Nu_{\mathbf{T}}(P') \setminus Nu_{\mathbf{T}}(P)} = \prec^+$,
3. $\{\tilde{u}\} = Nu_{\mathbf{T}}(P)$
4. $\{\tilde{u'}\} = Nu_{\mathbf{T}}(P') \setminus Nu_{\mathbf{T}}(P)$, and
5. $\prec' \subseteq (\mathbf{OT}(\Gamma') \cup \{\tilde{u'}\}) \times (\mathbf{OT}(\Gamma') \cup \{\tilde{u'}\})$,

by induction on the proof of $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$, from which Theorem 4.5 follows. Case analysis on the last rule used. When $\Gamma = \Gamma', x : T$, we write $\Gamma - x$ for $\Gamma', x : Rem(T)$.

- Case (R-Com)
  In this case, $P = x![\tilde{y}] \mid x?[\tilde{z}^{\tilde{t_z}}].Q$, and $P' = [\tilde{y}/\tilde{z}]Q$. By (T-Par) and (T-Out),

  Γ_1 is unlimited.
  $\Gamma_2, \prec \vdash x?[\tilde{z}^{\tilde{t_z}}].Q$
  $\Gamma_1 + x : !^m[\tilde{\rho^s}]^{t_x} + \tilde{y}{:}\tilde{\rho^t} + \Gamma_2 = \Gamma$
  $\Gamma' = \Gamma - x$
  $t_x \prec^\sharp \Sigma\{\tilde{y}{:}\tilde{\rho^t}\}$
  $([\tilde{t}/\tilde{s}]\ \prec) \subseteq \prec$

  for some Γ_1, Γ_2. Without loss of generality (by Lemma 3.8), we can assume that polarities of all the bindings in Γ_1 are $\mid$, and also that $x : !^m[\tilde{\rho^s}]^{t_x} + \tilde{y}{:}\tilde{\rho^t} + \Gamma_2 = \Gamma$. Case analysis on $m (\in \{1, \omega, M\})$.

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

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

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

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

The above equation also implies that $\Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}+\Gamma_3$ is well defined. From Lemma 3.11, we have $[\tilde{t}/\tilde{t_z}]\ \prec_1 = \prec$. Therefore, by repeated applications of Lemma 3.12 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 3.8),

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

That is,

$$\Gamma', \prec \vdash P'$$

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

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

for some Γ_3. Note that

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

The above equation also implies that $x : p^M[\tilde{\rho}^{\tilde{s}}]^{t_x} + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_3$ is well defined. By Lemma 3.11, we have

$$[\tilde{t}/\tilde{t_z}]\ \prec_1 = \prec\ .$$

Therefore, by repeated applications of Lemma 3.12 to $\Gamma_3, x : p^M[\tilde{\rho}^{\tilde{s}}]^{t_x}, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_1 \vdash Q$, we have

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

By weakening (Lemma 3.8),

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

That is,

$$\Gamma', \prec \vdash P'$$

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

$$\Gamma, [\tilde{u_1'}/\tilde{u_1}]\ \prec \vdash x![\tilde{y}]\,|\,x?^*[\tilde{z}^{\tilde{t_z}}].Q'$$

where $\{\tilde{u_1}\} = Nu_{\mathbf{T}}(Q)$ and $\tilde{u_1'}$ are the corresponding time tags in Q' ($\{\tilde{u_1'}\} = Nu_{\mathbf{T}}(Q')$). Let

$$\begin{aligned}\prec_1 &= [\tilde{u_1'}/\tilde{u_1}]\ \prec \cup [\tilde{u_1}/\tilde{u_1'}]([\tilde{u_1'}/\tilde{u_1}]\ \prec) \\ &= [\tilde{u_1'}/\tilde{u_1}]\ \prec \cup \prec\end{aligned}$$

Then by Lemma 3.10,

$$\Gamma, \prec_1 \vdash x![\tilde{y}] \,|\, x?^*[\tilde{z}^{\tilde{t_z}}].Q'$$
$$\Gamma, \prec_1 \vdash x![\tilde{y}] \,|\, x?^*[\tilde{z}^{\tilde{t_z}}].Q$$

By (T-Par), (T-Out), and (T-RIn),

$$\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{s}}]^\star, \prec_1 \vdash x?^*[\tilde{z}^{\tilde{t_z}}].Q'$$
$$\Gamma_2, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_2 \vdash Q'$$
$$(\Gamma_1 + x : !^*[\tilde{\rho}^{\tilde{s}}]^\star + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\}) + (\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{s}}]^\star) = \Gamma$$
$$\Gamma' = \Gamma - x = \Gamma$$
$$([\tilde{t}/\tilde{s}] \prec_1) \subseteq \prec_1$$
$$\{\tilde{u}'_2\} \not\prec_2 \not\succ (\mathbf{T} \backslash (\mathbf{OT}(\Gamma_2) \cup \{\tilde{u'_1}, \tilde{t_z}\}))$$
$$[\tilde{s}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u'_2}\}}) \subseteq \prec_2$$
$$(\prec_2 \Downarrow_{\{\tilde{u'_2}\}}) \downarrow_{\{\tilde{t_z}\}} = \prec_1$$
$$\{\tilde{u}'_2\} \subseteq \{\tilde{u}'_1\}$$

for some unlimited Γ_1,Γ_2, and time tags $\tilde{u}'_2$. By replacing $\tilde{u'_1}$ with $\tilde{u_1}$, we also have

$$\Gamma_2 + x : ?^*[\tilde{\rho}^{\tilde{s}}]^\star, \prec_1 \vdash x?^*[\tilde{z}^{\tilde{t_z}}].Q$$

Without loss of generality (by Lemma 3.8), we can assume that polarities of all the bindings in Γ_1 are $|.$ and that

$$\Gamma' = x : \updownarrow^*[\tilde{\rho}^{\tilde{s}}]^\star + \Sigma\{\tilde{y} : \tilde{\rho}^{\tilde{t}}\} + \Gamma_2.$$

Let $\prec_3 = [\tilde{t}/\tilde{t_z}] \prec_2$. We first show that $\prec_3^+$ is a strict partial order. Because $\prec_2$ is a strict partial order, $\prec_3^+$ is a strict partial order if $([\tilde{t}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u}'_2\}}))^+$ is. Because $\{\tilde{s}\} \not\prec_2 \not\succ \{\tilde{t_z}, \tilde{u}'_2\}$, we have

$$\begin{array}{lcl}
[\tilde{t}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u}'_2\}}) & \subseteq & [\tilde{t}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u}'_2\}} \downarrow_{\{\tilde{s}\}} \cup \prec_2 \Downarrow_{\{\tilde{u}'_2\}} \downarrow_{\{\tilde{t_z}\}}) \\
& = & ([\tilde{t}/\tilde{s}][\tilde{s}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u}'_2\}} \downarrow_{\{\tilde{s}\}})) \cup \prec_2 \Downarrow_{\{\tilde{u}'_2\}} \downarrow_{\{\tilde{t_z}\}} \\
& \subseteq & ([\tilde{t}/\tilde{s}](\prec_2 \downarrow_{\{\tilde{t_z}, \tilde{u'_2}\}})) \cup \prec_1 \\
& \subseteq & \prec_1
\end{array}$$

Therefore, $([\tilde{t}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u}'_2\}}))^+$ is a strict partial order, which implies that $\prec_3^+$ is also a strict partial order.

Therefore, by repeated applications of Lemma 3.12 to $\Gamma_2, \tilde{z} : \tilde{\rho}^{\tilde{t_z}}, \prec_2 \vdash Q'$, we have

$$\Gamma_2 + \tilde{y}{:}\tilde{\rho}^{\tilde{t}}, \prec_3 \vdash [\tilde{y}/\tilde{z}]Q'$$

Let $\prec_4 = \prec_3 \cup \prec_3 \Downarrow_{\{\tilde{u}'_2\}}$. Then, $\prec_4^+$ is also a strict partial order, and from the condition: $\{\tilde{u}'_2\} \not\prec_2 \not\succ (\mathbf{T} \backslash (\mathbf{OT}(\Gamma_2) \cup \{\tilde{u'_1}, \tilde{t_z}\}))$, all the pairs not in $\prec_3$ but in $\prec_4$ are in the set $(\mathbf{OT}(\Gamma_2) \cup \{\tilde{u'_1}, \tilde{t}\}) \times (\mathbf{OT}(\Gamma_2) \cup \{\tilde{u'_1}, \tilde{t}\})$. So, by Lemma 3.9, we have

$$\Gamma_2 + \tilde{y}{:}\tilde{\rho}^{\tilde{t}}, \prec_4 \vdash [\tilde{y}/\tilde{z}]Q'$$

On the other hand,

$$\begin{array}{lcl}
\prec_4 & \supseteq & \prec_3 \Downarrow_{\{\tilde{u'_2}\}} \\
& = & [\tilde{t}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u'_2}\}}) \\
& \supseteq & [\tilde{t}/\tilde{t_z}] \prec_1 \\
& = & \prec_1
\end{array}$$

We also have

$$\begin{array}{rcl}\prec_4 & = & [\tilde{t}/\tilde{t_z}](\prec_2 \cup \prec_2 \Downarrow_{\{\tilde{u_2'}\}}) \\ & = & [\tilde{t}/\tilde{t_z}](\prec_2 \Downarrow_{\{\tilde{u_2'}\}} \cup \prec_2 \uparrow_{\{\tilde{u_2'}\}}) \\ & \subseteq & \prec_1 \cup [\tilde{t}/\tilde{t_z}](\prec_2 \uparrow_{\{\tilde{u_2'}\}})\end{array}$$

Therefore, $\prec_4 = \prec_1 \cup \prec_5$ for some $\prec_5$ such that

$$\begin{array}{l}\prec_5 \subseteq (\mathbf{OT}(\Gamma_2) \cup \{\tilde{u_1'}, \tilde{t}\}) \times (\mathbf{OT}(\Gamma_2) \cup \{\tilde{u_1'}, \tilde{t}\}) \\ \quad (\subseteq (\mathbf{OT}(\Gamma) \cup \{\tilde{u_1'}\}) \times (\mathbf{OT}(\Gamma) \cup \{\tilde{u_1'}\}).\end{array}$$

So, by Lemma 3.9 and $\Gamma_2 + x : ?^*[\tilde{\rho^s}]^\star, \prec_1 \vdash x?^*[\tilde{z}^{\tilde{t_z}}].Q$, we have

$$\Gamma_2 + x : ?^*[\tilde{\rho^s}]^\star, \prec_4 \vdash x?^*[\tilde{z}^{\tilde{t_z}}].Q.$$

By (T-Par), weakening (Lemma 3.8), and the fact that Γ_2 is unlimited, we have

$$x : \updownarrow^*[\tilde{\rho^s}]^\star + \Sigma\{\tilde{y} : \tilde{\rho^t}\} + \Gamma_2, \prec_4 \vdash [\tilde{y}/\tilde{z}]Q' \mid x?^*[\tilde{z}^{\tilde{t_z}}].Q$$

That is,

$$\Gamma', \prec' \vdash P'$$

for $\prec' = \prec_4 = \prec \cup [\tilde{u'}/\tilde{u}] \prec \cup \prec_5$. Checking $\prec'^+ \downarrow_{\{\tilde{u_1'}\}} = \prec^+$ completes the proof of this case. It follows from $\prec' \supseteq \prec$ and the following equations.

$$\begin{array}{rcl}\prec'^+ \downarrow_{\{\tilde{u_1'}\}} & = & \prec_4^+ \downarrow_{\{\tilde{u_1'}\}} \\ & = & (\prec_3 \cup \prec_3 \Downarrow_{\{\tilde{u_2'}\}})^+ \downarrow_{\{\tilde{u_1'}\}} \\ & = & \prec_3^+ \downarrow_{\{\tilde{u_1'}\}} \\ & = & ([\tilde{t}/\tilde{t_z}] \prec_2)^+ \downarrow_{\{\tilde{u_1'}\}} \\ & = & ([\tilde{t}/\tilde{t_z}] \prec_2 \Downarrow_{\{\tilde{u_2'}\}})^+ \downarrow_{\{\tilde{u_1'}\}} \\ & \subseteq & \prec_1^+ \downarrow_{\{\tilde{u_1'}\}} \\ & = & \prec_1^+\end{array}$$

- 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 4.8, $\Gamma_1 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_1'$ for some Γ_1', and $\Gamma' = \Gamma_1' + \Gamma_2$. By induction hypothesis, $\Gamma_1', \prec'' \vdash P_1'$ where $\prec'' = \prec$ or $\prec'' = \prec \cup [\tilde{u'}/\tilde{u}] \prec \cup \prec'$. In either case, by Lemma 3.9 and 3.10, $\Gamma_2, \prec'' \vdash P_2$. By (T-Par), $\Gamma_1' + \Gamma_2, \prec'' \vdash P_1' \mid P_2$. That is, $\Gamma', \prec'' \vdash P'$.

- Cases (R-New), (R-IfT), and (R-IfF) are trivial.

- Case (R-Cong) follows immediately from Lemma 4.6.

□

The following theorem 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 $\Gamma(x) = ?^m[\tilde{T}]^t$ or $!^m[\tilde{T}]^t$. The theorem essentially says that if a well-typed process P cannot make any more reduction by itself, either (1)some subprocess is trying to send or receive a message along some unreliable or half used channel (in the first case below); or (2)all computation has been completed and only processes which correspond to process definitions (replicated input prefix) or stores (i.e., messages put on mutex channels) are left (in the second case below).

4.9 Theorem [lack of immediate deadlock]: Suppose $\Gamma, \prec \vdash P$ and $\{x \mid \Gamma(x) = \mathit{bool}\} \subseteq \{\mathit{true}, \mathit{false}\}$. 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 $x![\tilde{y}]$ where x has a mutex channel type or of the form $x?^*[\tilde{y}].Q$. (That is, all communication on channels except for replicated input and mutex channels have been completed.)

Proof: Let $(\nu\,\tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ be a normal form of P. Then, by Lemma 4.6, $\Gamma, \prec \vdash (\nu\,\tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$. By (T-New) and (T-Par), there exist $\Gamma_1, \ldots, \Gamma_n$ such that

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

None of $P_i (i \in \{1, \ldots, n\})$ is a conditional expression, because otherwise it must be that $P_i = \mathit{if\ true\ then}\ Q_1\ \mathit{else}\ Q_2$ or $P_i = \mathit{if\ false\ then}\ Q_1\ \mathit{else}\ Q_2$, which contradicts with the assumption $P \not\longrightarrow$. Suppose that the first condition of the above theorem does not hold, i.e., no process is trying to send/receive a value along unreliable channels or half-used channels. 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 $P_i = x![\tilde{y}]$ where x is a mutex channel, 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 holds.

First, suppose that $j > 0$. 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 $I \ni p$ for some $i \in \{2, \ldots, n\}$. Again from the typing rules, P_i must be of the form $x?^*[\tilde{v}].Q$, which contradicts with the assumption $P \not\longrightarrow$. Therefore, j must be 0.

Let us define the set S by:

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

Note that $\star \notin S$ from typing rules. Suppose that $S \neq \emptyset$. Because $\prec^+$ is a strict partial order, there is $t_i \in S$ such that there is no $t' \in S$ s.t. $t' \prec t_i$.

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

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

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

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

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

  From (T-Out), p must be !. Since $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^1[\tilde{T}]^\star$, there is some $i'(\neq i)$ such that $\Gamma_{i'} = ?^1[\tilde{T}]^{t_i}$. 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 $x?[\tilde{u}].Q_{i'}$, 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 input ends of linear channels cannot be communicated along mutex channels, and the second case contradicts with the condition of (T-RIn) ($\Gamma_{i'}(x)$ must be unlimited).

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

The above case analysis implies $S = \emptyset$, i.e., $l = 0$, which completes the proof of the theorem. □

The subject reduction property (Theorem 4.5) together with the lack of immediate deadlock (Theorem 4.9) ensures the following *partial deadlock freedom*.

4.10 Theorem [partial deadlock freedom]: Suppose that $\Gamma, \prec \vdash P$ and $\{x \mid \Gamma(x) = \mathit{bool}\} \subseteq \{\mathit{true}, \mathit{false}\}$. 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 $x![\tilde{y}]$ where x has the multiplicity M or $x?^*[\tilde{y}].Q$

Proof: Corollary of Theorem 4.5 and Theorem 4.9. □

The following corollary is a special case of Theorem 4.10.

4.11 Corollary: Suppose that $\Gamma, \prec \vdash P$, $\{x \mid \Gamma(x) = \mathit{bool}\} \subseteq \{\mathit{true}, \mathit{false}\}$, and that all the channel types in Γ, P are reliable channel types and 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 $x![\tilde{y}]$ where x has the multiplicity M 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 4.10. □

From the above corollary, we know that a program never falls into deadlock if only reliable channels are used. Theorem 4.10 also implies that if unreliable channels are used, deadlock can occur, but it is caused by some bad use of unreliable channels. The theorem itself says just that if a process is in a deadlock state, then it contains some sub-process which is trying to communicate along an unreliable or half-used channel. However, its proof also indicates that if there is a process blocked on a reliable channel, then there is always a process which is trying to communicate along an unreliable or half-used channel while blocking communication on reliable channels. Moreover, the proof indicates that we can identify the wrong process by looking at the tag ordering. The theorem below states this property. We need to define a few terms. If $\Gamma \vdash (\nu \tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$ for a normal form $(\nu \tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$, there are $\Gamma_1, \ldots, \Gamma_n$ such that $\Gamma_i \vdash P_i$ and $\Gamma_1 + \cdots + \Gamma_n = \Gamma, \tilde{w}{:}\tilde{U}$ (Note that such decomposition is unique for non-unlimited bindings). Suppose that $\Gamma_i(x) = p^m[\tilde{T}]^t$. We say that P_i *has an input occurrence* of x if $I \in p$ and $?^m[\tilde{T}]^t$ is not unlimited, and say that P_i *has an output occurrence* of x if $O \in p$ and $!^m[\tilde{T}]^t$ is not unlimited.

4.12 Theorem [strong partial deadlock freedom]: Suppose that $\Gamma, \prec \vdash P$, and $\{x \mid \Gamma(x) = \mathit{bool}\} \subseteq \{\mathit{true}, \mathit{false}\}$. Suppose also that $\Gamma \vdash P \longrightarrow^* P' \dashv \Gamma'$, and $\Gamma' \vdash P' \equiv (\nu \tilde{u}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n) \not\longrightarrow$ for guarded processes $P_1, \ldots, P_n$. If P_1 is either (1)an output on a linear or replicated input channel x or (2)an input on a linear or mutex channel x and if the binding of x in $\Gamma, \tilde{u}{:}\tilde{U}$ has polarity $\updownarrow$, then there are $i, j \in \{1, \ldots, n\}$ such that

- P_i is either (1)an output on a linear or replicated input channel y or (2)an input on a linear or mutex channel y and the binding of y has polarity $\updownarrow$; and
- P_j is an input or an output on an unreliable or half-used channel. Moreover, P_j contains an input occurrence of y if P_i is an output, and P_j contains an output occurrence of y if P_i is an input.

Proof: Almost the same as the proof of Theorem 4.9. (Take as P_i a process which is blocked on a reliable channel whose time tag is minimal among those attached to reliable channels with polarity $\updownarrow$ on which a process is currently blocked.) □

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 $\Gamma, \prec \vdash P$. Although we do not give a formal proof, it would be clear from the 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, construct a derivation tree of $\Gamma, \prec \vdash P$, with conditions on $\prec$ ignored in each step.
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 that we encounter (T-Par), when we must divide the current type environment for two sub processes. The key technique to avoid combinatorial explosion is to divide 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 that check succeeds, we can remove the used type environment Γ_1 from Γ, and check P_2 under the resulting environment Γ_2. This first step can be done in a polynomial time (with respect to the size of Γ, P), because only one rule can be applied at each step of construction of the derivation tree, the size of the derivation tree is linear to the size of P, and because 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 to the size of Γ, P. The curious reader may consult [6], which describes a similar (but more complex, because the two phases are mixed in [6]) type check algorithm. The same idea is also used to prove a linear logic sequent "$\Gamma \vdash A \otimes B$" in several linear logic programming languages[3].

The second step is described as follows. First, introduce the set of variables $\{\beta_{s,t} \mid s, t \in \mathbf{T}\}$, where each variable ranges over $\{0, 1\}$. Then, $\prec$ is completely determined by 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:

$$\beta_{t_1,t_2} = 1$$
$$\beta_{t_1,t_2} \geq \beta_{s_1,s_2}$$
$$\beta_{t_1,t_2} = 0$$

Constraints of the first form comes from the condition of the form $t \prec^\sharp \Gamma$, those of the second form from $\theta \prec \subseteq \prec$, and those of the third from $S_1 \not\prec\not\succ S_2$. Then, the above constraints can be basically 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. 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 well-known cycle detection algorithms for directed graphs. Let n be the size of an input: (Γ, P). 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 $O(n^4)$, time required for each step is $O(n^4)$. Therefore, the total time of 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 channels, i.e., the number of arguments passed through channels, is bound by some constant, $O(n^4)$ is an upper-bound.)

We did not mention above a phase to extract constraints from the type derivation. Because of the rule (T-Rin), it must be overlapped with iterations (since the set S of time tags in the rule cannot be determined until solving some constraints). However, this does not affect the above estimation of the computational cost, because we can first do iterations locally over constraints extracted from the sub-derivation for the premise of the rule (T-Rin), and then determine the set S, add the required constraints, and continue iterations.

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 can reason

from type information that a result of function invocation or method invocation of a concurrent object will be returned unless the invocation falls into an infinite loop; in other words, evaluation never gets stuck as in typed functional languages. Note that it has not been ensured in previous type systems for concurrent programming languages (including typed process calculi and typed concurrent object-oriented languages).

We first restate encoding of the call-by-value, simply typed λ-calculus presented in [6]. Then, we show the call-by-name and call-by-need simply typed λ-calculus can also be encoded into the deadlock-free fragment. As will be clear from the encoding, we can obtain the same result in the presence of recursion. We also informally present encoding of various concurrent objects.

6.1 Encoding Call-by-value λ-calculus

We show encoding of call-by-value, simply-typed λ-calculus into our process calculus.[4]

6.1.1 Syntax and Typing of $\lambda^{\rightarrow}$

We first introduce the syntax of the simply-typed λ-calculus.

6.1 Definition: The sets of types and preterms are given by the following syntax.

$$\begin{array}{lcll} \tau(types) & ::= & b & \text{(base types)} \\ & & \tau_1 \rightarrow \tau_2 & \text{(function types)} \end{array}$$

$$\begin{array}{lcl} e(terms) & ::= & x \\ & & \lambda x^{\tau}.e \\ & & e_1 e_2 \end{array}$$

6.2 Definition: A preterm e is a term of type τ under the type environment $\mathcal{T}$ if we have a derivation for $\mathcal{T} \vdash e : \tau$ by the following set of rules.

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

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

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

6.3 Lemma: If $\mathcal{T} \vdash e : \tau$ and $\mathcal{T} \vdash e : \tau'$, then $\tau = \tau'$.

Proof: Trivial induction on the structure of e. □

6.1.2 Encoding

Now, we show encoding of λ-terms into our calculus, following [10, 16]. By Definition 6.2 and Lemma 6.3, if e is a λ-term 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 $\mathcal{T} \vdash e : \tau$. We first give encoding of types and type environments. To understand the encoding of function types below, notice that function invocation is implemented as an output of a pair consisting of an argument of the function and the output end of a linear channel to obtain a result, to the corresponding replicated input channel (recall Example 2.8).

[4] Though we do not show in this paper, λ-calculus with more advanced type systems could also be captured by appropriate extensions of our process calculus with subtyping, polymorphism, etc.

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

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

The above encoding of types have arbitrariness on the choice of time tags attached to linear channel type constructors. For simplicity of statements of theorems, we use below the same tag s for those attached to linear channel type constructors in the encoding of types. The encoding function is pointwise extended to type environments:

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

6.5 Lemma: Let τ be a type of $\lambda^{\rightarrow}$. Then, $[\![\tau]\!]$ is an unlimited type.

Proof: Trivial from 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 $[\![e]\!]_{\mathcal{T}}^{p,t}$ returns (1) a process that evaluates e and returns the result to the channel p (whose time tag is t), and (2) a tag ordering $\prec$. We use a linear channel as the result channel p. A variable just returns its name to the return address.

$$[\![x]\!]_{\mathcal{T}}^{p,t} = (p![x], \emptyset)$$

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 its address.

$$\begin{array}{l} [\![\lambda x^{\tau_1}.e]\!]_{\mathcal{T}}^{p,t} = \\ \quad \textbf{let } (P, \prec) = [\![e]\!]_{\mathcal{T}, x:\tau_1}^{y,u} \\ \quad \textbf{in } ((\nu y : \updownarrow^{*}[[\![\tau_1]\!], !^{1}[[\![\tau_2]\!]]^{s}]^{\star})(p![y] \mid y?^{*}[x, q].P), \emptyset) \end{array}$$

Here $\tau_2 = (\mathcal{T}, x : \tau_1)(e)$ and u is a fresh time tag. 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]

$$\begin{array}{l} [\![e_1 e_2]\!]_{\mathcal{T}}^{p,t} = \\ \quad \textbf{let } (P_1, \prec_1) = [\![e_1]\!]_{\mathcal{T}}^{y,u_1} \\ \qquad (P_2, \prec_2) = [\![e_2]\!]_{\mathcal{T}}^{z,u_2} \\ \quad \textbf{in } ((\nu y : \updownarrow^{1}[[\![\tau_1 \rightarrow \tau_2]\!]]^{u_1})(\nu z : \updownarrow^{1}[[\![\tau_1]\!]]^{u_2})(P_1 \mid P_2 \mid y?[f].z?[v].f![v, p]), \\ \qquad \prec_1 \cup \prec_2 \cup \{(u_1, u_2), (u_2, t), (u_1, t)\}) \end{array}$$

Here u_1, u_2 are fresh time tags and $\tau_1 = \mathcal{T}(e_2), \tau_2 = \mathcal{T}(e_1 e_2)$.

The following theorem, proved by straightforward induction on the structure of e, essentially says that a well-typed λ-term is encoded into a process well-typed under the corresponding type environment.

6.6 Theorem [Encoding preserves typing]: Suppose $\mathcal{T} \vdash e : \tau$, and let $(P, \prec) = [\![e]\!]_{\mathcal{T}}^{p,t}$. Then $\prec^{+}$ is a strict partial order and $[\![\mathcal{T}]\!], p : !^{1}[[\![\tau]\!]]^{t}, \prec \vdash P$.

Proof: We show $\prec \subseteq (Nu_{\mathbf{T}}(P) \cup \{t\}) \times (Nu_{\mathbf{T}}(P) \cup \{t\})$ in addition to the properties in Theorem 6.6 by induction on the structure of e.

[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 property below. Proofs of the properties below are slightly easier for the alternative encoding.

- Case $e = x$: In this case, $\mathcal{T} = \mathcal{T}', x : \tau$, $P = p![x]$ and $\prec = \emptyset$. Since $[\![\mathcal{T}']\!]$ is unlimited, we have

  $$[\![\mathcal{T}']\!], p : !^1[[\![\tau]\!]]^t, x : [\![\tau]\!], \emptyset \vdash p![x]$$

  by (T-OUT). That is,

  $$[\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]]^t \prec \vdash P.$$

- Case $e = \lambda x^{\tau_1}.e'$. In this case, $\mathcal{T}, x : \tau_1 \vdash e' : \tau_2$. Let $(P_1, \prec_1) = [\![e']\!]^{q,t_1}_{\mathcal{T},x:\tau_1}$. Then,

  $$P = (\nu y : \updownarrow^*[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]^s]^\star)(p![y] \mid y?^*[x, q].\, P_1)$$
  $$\prec = \emptyset$$

  By induction hypothesis, we have

  $$[\![\mathcal{T}]\!], x : [\![\tau_1]\!], q : !^1[[\![\tau_2]\!]]^{t_1} \prec_1 \vdash P_1$$
  $$\prec_1 \subseteq (Nu_{\mathbf{T}}(P_1) \cup \{t_1\}) \times (Nu_{\mathbf{T}}(P_1) \cup \{t_1\})$$

  By (T-RIN),

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

  Because $[\![\mathcal{T}']\!]$ is unlimited, we also have:

  $$[\![\mathcal{T}]\!], p : !^1[[\![\tau_1 \to \tau_2]\!]], y : [\![\tau_1 \to \tau_2]\!], \emptyset \vdash p![y]$$

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

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

- 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 $(P_1, \prec_1) = [\![e_1]\!]^{y,u_1}_{\mathcal{T}}$, $(P_2, \prec_2) = [\![e_2]\!]^{z,u_2}_{\mathcal{T}}$. Then,

  $$P = (\nu y : \updownarrow^1[[\![\tau_1 \to \tau_2]\!]]^{u_1})(\nu z : \updownarrow^1[[\![\tau_1]\!]]^{u_2})(P_1 \mid P_2 \mid y?[f].\, z?[v].\, f![v, p])$$
  $$\prec = \prec_1 \cup \prec_2 \cup \{(u_1, u_2), (u_2, t), (u_1, t)\}$$

  By induction hypothesis,

  $$[\![\mathcal{T}]\!], y : !^1[[\![\tau_1 \to \tau_2]\!]]^{u_1}, \prec_1 \vdash P_1$$
  $$[\![\mathcal{T}]\!], z : !^1[[\![\tau_1]\!]]^{u_2}, \prec_2, \vdash P_2$$
  $$\prec_1 \subseteq (Nu_{\mathbf{T}}(P_1) \cup \{u_1\}) \times (Nu_{\mathbf{T}}(P_1) \cup \{u_1\})$$
  $$\prec_2 \subseteq (Nu_{\mathbf{T}}(P_2) \cup \{u_2\}) \times (Nu_{\mathbf{T}}(P_2) \cup \{u_2\}).$$

  From the last two conditions, $\prec \subseteq (Nu_{\mathbf{T}}(P) \cup \{t\}) \times (Nu_{\mathbf{T}}(P) \cup \{t\})$ and $\prec^+$ is a strict partial order. By Lemma 3.9,

  $$[\![\mathcal{T}]\!], y : !^1[[\![\tau_1 \to \tau_2]\!]]^{u_1}, \prec \vdash P_1$$
  $$[\![\mathcal{T}]\!], z : !^1[[\![\tau_1]\!]]^{u_2}, \prec \vdash P_2.$$

  By typing rules, we have

  $$[\![\mathcal{T}]\!], p : !^1[[\![\tau_2]\!]]^t, \prec \vdash P.$$

□

We obtain the following theorem as a corollary of Theorems 4.10 and 6.6. The theorem essentially says that an encoded term eventually returns a result or falls into an infinite loop. That is, the evaluation of the encoded term never gets stuck. Note that we also know from Theorem 4.4 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 similar to the one developed for the linear channel type system [8] will suffice.)

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

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

Proof: Let $T = !^1[[\![\tau]\!]]^t$. By Theorem 6.6, $p : T, \prec \vdash P$. By the definition of the reduction relation, $\Delta = p : T$. Suppose that $\Delta \vdash P' \not\longrightarrow$. By the subject reduction theorem (Theorem 4.5), we have $p : T, \prec' \vdash P'$ for some tag ordering $\prec'$. Since every channel in P' is a linear or replicated input channel, we have either

1. $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(p![v]\{\,|\,Q\})$; 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 4.10. 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 call-by-name, simply typed λ-calculus. We can again reason from type information that a process which simulates a well-typed λ-term never deadlocks, i.e., it eventually returns a result or falls into an infinite loop. We use the same syntax for λ-terms and types as in Subsection 6.1.

We follow [14, 23] for the encoding of call-by-name reduction. In call-by-name reduction, an argument of a function is passed before it is evaluated. So, a function type is represented by a type of a channel along which a caller process pass a pair of channels, one of which is used by a callee process to trigger an evaluation of the argument, and the other to return the result. Thus, encoding of types and type environments is given as follows.

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

$$\begin{array}{lcl} [\![b]\!] & = & b^\star \\ [\![\tau_1 \to \tau_2]\!] & = & !^*[!^*[!^1[[\![\tau_1]\!]]^s]^\star, !^1[[\![\tau_2]\!]]^s]^\star \end{array}$$

(Note that the same time tag s is used for linear channel type constructors.)

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

6.9 Definition: Encoding of type environments is defined by:

$$\begin{array}{lcl} [\![\emptyset]\!] & = & \emptyset \\ [\![\mathcal{T}, x : \tau]\!] & = & [\![\mathcal{T}]\!], x : !^*[!^1[[\![\tau]\!]]^s]^\star \end{array}$$

We give an encoding of terms below. As in the previous section, the encoding function $[\![e]\!]^{p,t}_{\mathcal{T}}$ returns a pair of (1) a process which evaluates e (in the call-by-name style) and returns the result to the channel p, and (2)a tag ordering $\prec$.

A variable is implemented as a process which requests the value bound to the variable to be sent to p. A name of a variable is used as a name of a channel along which a request should be sent.

$$[\![x]\!]^{p}_{\mathcal{T}} = (x![p], \emptyset)$$

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

$$\begin{array}{l}
[\![\lambda x^{\tau_1}.e]\!]^{p,t}_{\mathcal{T}} = \\
\quad \textbf{let } (P, \prec) = [\![e]\!]^{q,u}_{\mathcal{T},x:\tau_1} \\
\quad \textbf{in } ((\nu y : \updownarrow^*[!^*[!^1[[\![\tau_1]\!]]^s]^\star, !^1[[\![\tau_2]\!]^s]^\star)\,(p![y] \mid y?^*[x,q].\,P), \emptyset)
\end{array}$$

where $\tau_2 = (\mathcal{T}, x:\tau_1)(e)$ and u is a fresh time tag.

The function application $e_1 e_2$ first spawns a process to evaluate the function e_1, and passes to it a channel along which a request for the value of e_2 is sent, and a result channel. We use an auxiliary encoding $\langle\!\langle e \rangle\!\rangle^{x}_{\mathcal{T}}$. It waits on a channel x for a request for the value of e, and upon a request, evaluates e and returns the value.

$$\begin{array}{l}
[\![e_1 e_2]\!]^{p,t}_{\mathcal{T}} = \\
\quad \textbf{let } (P_1, \prec_1) = [\![e_1]\!]^{y,u_1}_{\mathcal{T}} \\
\qquad P_2 = \langle\!\langle e_2 \rangle\!\rangle^{z}_{\mathcal{T}} \\
\quad \textbf{in } ((\nu z : \updownarrow^*[!^1[[\![\tau_1]\!]]^s]^\star)\,(\nu y : !^1[[\![\tau_1 \to \tau_2]\!]]^{u_1})\,(P_1 \mid P_2 \mid y?[f].\,f![z,p]), \prec_1 \cup \{u_1, t\}) \\
\langle\!\langle e \rangle\!\rangle^{z}_{\mathcal{T}} = \\
\quad \textbf{let } (Q, \prec_2) = [\![e]\!]^{w,u_2}_{\mathcal{T}} \\
\quad \textbf{in } z?^*[w].\,Q
\end{array}$$

where $\tau_1 = \mathcal{T}(e_2), \tau_2 = \mathcal{T}(e_1 e_2)$, and u_1, u_2 are fresh time tags.

We can again prove the same theorems as those with the encoding of call-by-value reduction: typing is preserved by the encoding, and as its corollary, we can reason from type information that evaluation of the encoded process never gets stuck. We skip proofs, because they are almost the same as those for the call-by-value λ-calculus.

6.10 Theorem [Encoding preserves typing]: Suppose $\mathcal{T} \vdash e : \tau$, and let $(P, \prec) = [\![e]\!]^{p,t}_{\mathcal{T}}$. Then $\prec^+$ is a strict partial order, and $[\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]]^t, \prec \vdash P$.

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

1. $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(p![v]\{\mid Q\})$, 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 call-by-need λ-calculus by a modification to the encoding of call-by-name λ-calculus. Except for orderings on time tags, we just need to redefine $\langle\!\langle e \rangle\!\rangle^{x}_{\mathcal{T}}$ used in the encoding of applications. $\langle\!\langle e \rangle\!\rangle^{x}_{\mathcal{T}}$ in the case of call-by-need λ-calculus waits on a channel x for a request for the value of e, and for the first request, evaluates e and replies the value. After that, it just replies the value computed before for each request.

The encoding $\langle\!\langle e \rangle\!\rangle^{x}_{\mathcal{T}}$ is given by:

$$
\begin{array}{l}
\langle\!\langle e \rangle\!\rangle^{x}_{\mathcal{I}} = \\
\quad \textbf{let } (P, \prec) = [\![e]\!]^{k,t_1}_{\mathcal{I}} \\
\quad \textbf{in } ((\nu\, flag : \updownarrow^{M}[bool]^{u_1})\,(\nu r : \updownarrow^{M}[[\![\tau]\!]]^{u_2}) \\
\qquad (flag![true] \mid r![dummy] \mid \\
\qquad x?^{*}[y].\, flag?[b]. \\
\qquad\quad (\mathit{if}\ b\ \mathit{then}\ (\nu k : \updownarrow^{1}[[\![\tau]\!]]^{t_1})\,(P \mid k?[v].\, r?[v'].\, (r![v] \mid flag![false] \mid y![v])) \\
\qquad\qquad \mathit{else}\ r?[v].\, (r![v] \mid flag![false] \mid y![v]))), \\
\qquad \prec\downarrow_S \cup \{(u_1, s), (u_2, s), (u_2, u_1)\})
\end{array}
$$

where $\tau = [\![\mathcal{T}(e)]\!]$, u_1, u_2, t_1 are fresh time tags, and S is the set:

$$\{t \mid t \in Nu_{\mathbf{T}}(P) \wedge t \text{ is a time tag attached to a linear channel type}\}.$$

The above encoding uses a mutex channel $flag$ to remember whether or not e is evaluated before, 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 $\tau = \tau_1 \to \tau_2$, there should be a free replicated input channel $dummy_\tau$, and a process $dummy_\tau?^{*}[x, r].\, r![dummy_{\tau_2}]$ should be run in parallel to processes obtained by the above encoding (and also type bindings $dummy_\tau : \updownarrow^{*}[!^{*}[!^{1}[[\![\tau_1]\!]]^{s}]^{\star}, !^{1}[[\![\tau_2]\!]]^{s}]^{\star}$ should appear in type environments).

Other encodings are the same as encoding of call-by-name λ-calculus except for orderings:

$$[\![x]\!]^{p}_{\mathcal{I}} = (x![p], \emptyset)$$

$$
\begin{array}{l}
[\![\lambda x^{\tau_1}.e]\!]^{p,t}_{\mathcal{I}} = \\
\quad \textbf{let } (P, \prec) = [\![e]\!]^{y,u}_{\mathcal{I}, x:\tau_1} \\
\quad \textbf{in } ((\nu y : \updownarrow^{*}[!^{*}[!^{1}[[\![\tau_1]\!]]^{s}]^{\star}, !^{1}[[\![\tau_2]\!]]^{s}]^{\star})\,(p![y] \mid y?^{*}[x, q].\, P), \prec\downarrow_S)
\end{array}
$$

where $\tau_2 = (\mathcal{T}, x : \tau_1)(e)$, u is a fresh time tag, and S is the set $\{t \mid t \in Nu_{\mathbf{T}}(P) \wedge t \text{ is a time tag attached to a linear channel type}\}$.

$$
\begin{array}{l}
[\![e_1 e_2]\!]^{p,t}_{\mathcal{I}} = \\
\quad \textbf{let } (P_1, \prec_1) = [\![e_1]\!]^{y,u_1}_{\mathcal{I}} \\
\qquad (P_2, \prec_2) = \langle\!\langle e_2 \rangle\!\rangle^{z}_{\mathcal{I}} \\
\quad \textbf{in } ((\nu z : \updownarrow^{*}[!^{1}[[\![\tau_1]\!]]^{s}]^{\star})\,(\nu y : !^{1}[[\![\tau_1 \to \tau_2]\!]]^{u_1})\,(P_1 \mid P_2 \mid y?[f].\, f![z, p]), \\
\qquad \prec_1 \cup \prec_2 \cup \{u_1, t\})
\end{array}
$$

We can again prove that typing is preserved by the encoding, and as its corollary, we can reason from type information that evaluation of the encoded process never gets stuck. Unfortunately, we cannot reason that the result is unique only from type information, because mutex channels are used.

6.12 Theorem [Encoding preserves typing]: Suppose $\mathcal{T} \vdash e : \tau$, and let $(P, \prec) = [\![e]\!]^{p,t}_{\mathbb{T}}$. Then $\prec^{+}$ is a strict partial order, and $[\![\mathcal{T}]\!], true : bool, false : bool, p : !^{1}[[\![\tau]\!]]^{t}, \prec \vdash P$.

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

1. $P' \equiv (\nu \tilde{w}{:}\tilde{U})\,(p![v]\{\mid Q\})$, 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 atomic execution of methods (because more than one processes 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 a method of itself during execution of a state-updating method). The second one is a little bit more complicated, but more expressive.

6.4.1 Simple Encoding

Basically, a concurrent object is modelled 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 [17]. The following process corresponds to a definition of concurrent objects (we use records for readability).

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

The above process waits for a request of creating 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, and spawns processes to handle each method, and returns a record whose each field contains a channel to which a request of method execution should be sent. The record of channels corresponds to a reference of 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 the result and write a new state into st. Other method handlers are blocked until the new state is written, because st is a mutex channel. Thus, the above concurrent objects are realized with 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).

6.14 Example: The following process is a definition of counter objects. (For simplicity, we assume primitive operations on integers. $n + 1$ could instead be represented by processes.)

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

P is typed as follows:

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

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

$$\begin{aligned} Q = {} & (\nu r : \updownarrow^1[\{inc : !^*[!^1[\,]^s]^\star, read : !^*[!^1[int]^s]^\star\}]^{u_2}) \\ & (newcounter![3,r] \,|\, r?[o].(\nu w : \updownarrow^1[\,]^{u_3})(o.inc![w] \,|\, w?[\,].(o.read![result]))) \end{aligned}$$

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

$$newcounter : \updownarrow^*[int^\star, !^1[\{inc : !^*[!^1[\,]^s]^\star, read : !^*[!^1[int]^s]^\star\}]^{u_1}]^\star, result : !^1[int^\star]^{u_4}, \prec \vdash P \,|\, Q$$

where $\prec = \{(t,s),(t,u_3),(t,u_4),(u_2,u_4),(u_3,u_4)\}$. By the theorems in Section 4, the above type judgement implies that if $P \,|\, Q$ is reduced to a process which cannot make reduction any more, then the process must be congruent to $(\nu \tilde{w}{:}\tilde{W})(result![m] \,|\, R)$, i.e., we get a value back in the channel *result*.

The following example shows limitations of the simple encoding of concurrent objects; an invocation of self methods cause deadlock with the simple encoding (the similar deadlock can happen also when two concurrent objects mutually call methods before releasing their locks on states). Our type system can correctly detect such cases.

6.15 Example: Let us consider a case where a method of a concurrent object calls itself before terminating its execution. With the above simple encoding, a definition is expressed as follows (for simplicity, we omit unimportant time tags below):

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

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:

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

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

6.4.2 Alternative Encoding

We show a little bit more complicated encoding of concurrent objects in order to allow concurrent objects to invoke methods of themselves. The idea of the encoding is (1)to use different channels for a channel to realize exclusive execution of methods and for a channel to keep the state, and (2)to use different processes for handling method invocations from the inside and those from the outside. The following process corresponds to a definition of a concurrent object.

$$
\begin{array}{l}
new?^*[init, r]. \\
\quad (\nu lock : \updownarrow^M[]^{u_1})\,(\nu st : \updownarrow^M[T_0]^{u_2}) \\
\quad (\nu f_1 : \updownarrow^*[\tilde{T}_1]^\star) \cdots (\nu f_n : \updownarrow^*[\tilde{T}_n]^\star) \\
\quad (\nu m_1 : \updownarrow^*[\tilde{T}_1]^\star) \cdots (\nu m_n : \updownarrow^*[\tilde{T}_n]^\star) \\
\quad (r![\{meth_1 = m_1, \ldots, meth_n = m_n\}] \\
\quad \mid st![init] \\
\quad \mid f_1?^*[v, c].(\\
\qquad \textbf{let } self = \{meth_1 = f_1, \ldots, meth_n = f_n\} \textbf{ in} \\
\qquad st?[state].(st![state] \mid \cdots st?[state].(c![v'] \mid st![state']))) \\
\quad \mid \cdots \\
\quad \mid f_n?^*[v, c].(\\
\qquad \textbf{let } self = \{meth_1 = f_1, \ldots, meth_n = f_n\} \textbf{ in} \\
\qquad st?[state].(st![state] \mid \cdots st?[state].(c![v'] \mid st![state']))) \\
\quad \mid m_1?^*[v, c].lock?[].(\nu c' : T_1')\,(f_1![v, c'] \mid c'?[res].(c![res] \mid lock![])) \\
\quad \mid \cdots \\
\quad \mid m_n?^*[v, c].lock?[].(\nu c' : T_n')\,(f_n![v, c'] \mid c'?[res].(c![res] \mid lock![])))
\end{array}
$$

where **let** $self = v$ **in** P is a shorthand for $(\nu x)\,(x![v] \mid x?[self].\,P)$. Note that we again 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 will be sufficient to use constraints on the tag ordering in a type judgment, 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 [8], 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 [5] for a variant of the linear channel type system, it would be possible to develop a (probably partial) type reconstruction algorithm along the similar line.

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, and otherwise the process is ill typed and rejected. It may be too restrictive in practice because our type system sometimes rejects a process which does not actually 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 needs to check use of the channels x and y, but need not worry about use of the channel 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 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 judgement 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 use of channels. For example, consider the process $P = x?[].y?[].x?[].\mathbf{0}$: 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 occurrences of 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: 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:

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

and the ordering $\{(t_2,s),(s,t_2)\}$ indicates that $P \,|\, Q$ may deadlock: in fact, if Q is

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

then $P \,|\, Q$ can deadlock. 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 lower and upper bounds of the ordering are strongly related to each other; therefore not only an upper bound but a lower bound play 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 judgement for an an output expression:

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

In our type system, ρ_1 and ρ_2 must be exactly the same. This requirement sometimes propagates unnecessary orderings, which leads to rejection of processes that do not actually cause 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:

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

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

7.2.4 First-class time tags

The rule (T-RIN) indicates that we must propagate the orderings on some of channels created in bodies of replicated input expressions. For example, in the encoding of concurrent objects shown in Section 6, orderings on time tags of mutex channels must be kept globally. This sometimes leads to rejection of processes which do not actually cause deadlock. For example, consider the following object definition:

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

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 orderings on the mutex channel used in o and those on st are merged by our type system. We can avoid this by allowing u to be passed along channels (and by introducing subtyping):

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

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. It implies that more than one channels may hold the same time tag, which may cause non-existing deadlock to be detected. Instead, we could use a finite representation of an ordering on an infinite set of time tags (which would be a "regular" infinite graph as the analogue of a regular tree). 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 treat replicated input expressions more elegantly. For example, the process of Example 3.5 would be written as:

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

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. 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 $(\nu y : \updownarrow^\omega[\,])\,(y?[\,].\, x![\,])$; the other reason is that, even if we forbid the use of unreliable channels, a process may perform infinite reductions without ever sending a value to x, as in $y![x] \mid y?^*[z].\, y![z]$. Therefore, 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 [8].

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 such as the π-calculus (except for an earlier version of our type system [6], 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, Gay, and Nagarajan [1] 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 [16] and the linear channel type system [8]. Especially, we used ideas of the latter to ensure usage of communication channels, though ordering information was not introduced there.

Recently, Yoshida [25] independently studied a type system for process calculi 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 [11] into the monadic π-calculus [12]. 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 correspond 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 [21] 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.

Several techniques have been proposed for static analysis of communication in CML: for example, Nielson and Nielson [13] proposed a technique to extract terms of process algebra from CML programs and Colby [2] proposed another technique based on abstract interpretation. Those techniques help programmers understand behaviors of CML programs to some extent. However, to the best of our knowledge, (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 on 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 $\mathbf{0}$. This example indicates that when we approximate programs by mapping multiple channels to the same channel, there is a case in which the original program may fall into deadlock but its approximation does not. On the other hand, consider the process $x?[\,].z?[\,].y![\,] \mid x![\,] \mid y?[\,].\mathbf{0} \mid z![\,]$. This process 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:

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

From the above examples, we can see that, once multiple channels are mapped to the same channel, whether or not the original program falls into deadlock 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.

Acknowledgement

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 us consider first-class 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, and Simon Gay for discussions on the relationships between their work and ours.

References

[1] Martin Berger, Simon Gay, and Rajagopal Nagarajan. A typed calculus of deadlock-free processes. draft paper, 1997.

[2] Christopher Colby. Analyzing the communication topology of concurrent programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1995.

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

[4] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 1995.

[5] Atsushi Igarashi and Naoki Kobayashi. Type-based analysis of usage of communication channels for concurrent programming languages. Technical report, Department of Information Science, University of Tokyo, 1997. to appear.

[6] Naoki Kobayashi. A Partially Deadlock-free Typed Process Calculus (I) – A Simple System –. Technical Report 96-02, Department of Information Science, University of Tokyo, September 1996.

[7] Naoki Kobayashi. Concurrent linear logic programming. Doctoral Thesis, Department of Information Science, Unversity of Tokyo, April 1996.

[8] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371, January 1996.

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

[10] Robin Milner. Function as processes. In *Automata, Language and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.

[11] Robin Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

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

[13] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 84–97, 1994.

[14] Gerald Ostheimer and Antony J. T. Davie. π-calculus characterizations of some practical λ-calculus reduction strategies. Technical Report, St Andrews University, 1993.

[15] Simon Peyton-Jones. Concurrent haskell. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 295–308, 1996.

[16] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 376–385, 1993.

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

[18] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997. To appear in Milner *Festschrift*, MIT Press, 1997.

[19] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[20] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[21] Davide Sangiorgi. Typed π-calculus at work: a proof of jones's parallelisation transformation on concurrent objects. In *Fourth Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, 1997.

[22] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

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

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

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