TECHNICAL REPORT 96-02

A Partially Deadlock-free Typed Process Calculus (I) – A Simple System –

Naoki Kobayashi

September 1996

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 (I)
– A Simple System –

AUTHORS
Naoki Kobayashi

KEY WORDS AND PHRASES
process calculus, type system, deadlock, linear channels, linear types

ABSTRACT
Concurrency primitives play an important role in describing programs on parallel/distributed environments and also in writing interactive programs. Theoretical supports for concurrency primitives, however, have so far been very limited. Several type systems have been recently proposed through process calculi, but most of them do not solve inherent problems in concurrent programs: deadlock and non-determinism. We propose a novel type system that ensures both partial deadlock-freedom and partial confluence, along the line of Kobayashi, Pierce, and Turner's linear channel type system. The technical novelty lies in the use of a poset as a type environment, by which capturing the order of channel uses. With the type system, for example, call-by-value 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.

REPORT DATE	**WRITTEN LANGUAGE**
September 1996	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
32	26

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

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 (I)
– A Simple System –

Naoki Kobayashi
Department of Information Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 JAPAN
email:koba@is.s.u-tokyo.ac.jp

1 Introduction

1.1 Background and Problems

The ability of expressing concurrency and communication brings a great expressive power to programming languages. Such concurrency primitives are useful for writing programs on parallel/distributed environments. Although it is possible to design high-level programming languages that hide such environments, concurrency primitives are indispensable at least for implementation-level languages. Also, even on sequential environments, concurrency primitives are useful for describing interactive applications such as graphical user interfaces[1]. For these purposes, a number of concurrent programming languages or concurrent extensions of sequential programming languages have been proposed[23, 25, 20, 8, 19, 15]. However, naive introduction of concurrency primitives obviously loses many good properties of programs. For example, if we extend functional programming languages with concurrency primitives[20, 15], then programs do not behave like functions any more. In order to illustrate the problem, consider the following CML "function" definitions (we follow the syntax in [20]):

```
fun f n = let
             val ch=channel()
          in
             accept(ch)+n+1
           end;
fun g n = let
             val ch=channel()
          in
             (spawn (fn ()=> send(ch, 2));
              accept(ch)+n+1)
          end;
fun h n = let
             val ch=channel()
          in
             (spawn (fn ()=> send(ch, 2));
              spawn (fn ()=> send(ch, 0));
              accept(ch)+n+1)
          end;
```

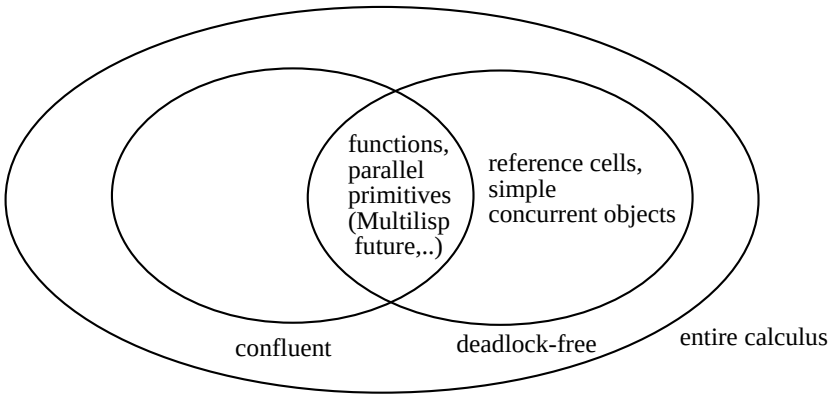
The function `f` creates a new channel `ch` (`channel()` is a CML primitive to create a new channel), waits for a value v from the channel (`accept(ch)` is another CML primitive to wait for a value from the channel `ch`. When a value is received, `accept(ch)` is evaluated to the value), and returns $v+n+1$. Since there is no sender to the channel `ch`, the function invocation `f(1)` is obviously blocked forever. On the other hand, consider a slightly different function `g(1)`. `g(1)` creates a new channel `ch`, spawns a process to send `2` to `ch`, and then waits for a value from `ch` (`spawn (fn ()=>e)` spawns a process to evaluate `e`, `send(ch, v)` sends `v` along `ch`, and `e1; e2` evaluates `e1` and `e2` sequentially.) Therefore, `g(1)` is not blocked and returns `4`. In spite of this difference of the behaviors, both `f` and `g` have the same type "$int \rightarrow int$" in CML. Thus, the property that evaluation of well-typed programs never gets stuck has been lost. Although it is easy to see whether the above programs are blocked or not, it is difficult to see it for more complex programs. The third function h also has the same type and returns a result, but the result is non-deterministic depending on which value is received in `accept`. Thus, a term of the function type does not behave any more like a function; it may be blocked in the middle of evaluation or it may even be non-deterministic. These examples illustrate the difficulty of writing and reasoning about concurrent programs, and indicate requirements for better theoretical treatments of concurrency primitives. The sources of the above problems, deadlock and non-determinism, also make efficient implementation hard, because it is not easy to utilize stack and registers effectively with the assumption that evaluation may be blocked.

Recently, a number of studies focused on concurrency primitives and tried to obtain better foundations. There are a number of researches on process equivalence[13, 21, 5], type systems[3, 17, 9], relationship with logic[8, 22], etc. Concurrent programming languages directly based on such new foundations have also been designed and implemented[19, 10]. However, such foundations essentially have not been able to solve the above problems. Rather, if we naively throw away function primitives and work only with concurrency primitives as in the π-calculus[11], the problem becomes much greater because high-level program structures are completely lost or hidden in combinations of concurrency primitives.

1.2 Our Approach

The important observation made in this paper is that not all parts of concurrent programs suffer from deadlock and non-determinism. Most parts of realistic programs should be deadlock-free and/or deterministic. The purpose of this paper is, therefore, to identify deadlock-free and/or confluent parts in concurrent programs,[1] by classifying communication channels based on their usage, and by enforcing the usage by a static type system. Since the fact that communication on certain channels satisfies the confluence property has already been addressed in previous work[4, 9], we mainly focus on ensuring *partial deadlock freedom*. By *partial deadlock freedom*, we mean that (1)as far as only certain reliable channels are used, deadlock never occurs; and that (2)if unreliable channels are also used, then deadlock may occur, but we know that the deadlock was caused by wrong communication on the *unreliable* channels; that is, some process is trying to send or receive a value on some unreliable channel for which there is no partner of the communication. The resulting image of our typed process calculus is shown in Figure 1.2. Although the entire calculus may behave in an unstructured manner, certain parts of the calculus are ensured to be deadlock-free and/or confluent. For example, as is shown later, Milner's encoding of functions into the π-calculus[11] essentially uses the deadlock-free and confluent fragment, hence the behavior of functions is well captured in our calculus. The deadlock-free but non-deterministic fragment in the figure contains reference cells, and more generally

[1] By a "confluent" program, we mean that the reduction of the program is confluent. That is, the order of evaluation can be non-deterministic, but the result should always be the same.



concurrent objects (though only very simple concurrent objects are captured in the simple type system presented here).

Contributions Our major contributions in this paper are listed as follows:

1. The Use of Posets as Type Environments – Our type system has been built upon previous type systems[17, 9] (especially the linear channel type system[9]) for process calculi, but the novelty of our type system lies in the use of *posets* (partially ordered sets) of bindings as type environments. The ordering represents in which order each value is used. For example, the type environment $x : int; y : real$ (the semicolon represent the ordering $x \leq y$) means that y is available as a value of the type *real only after* x is used as a value of the type int. The use of the ordering is the key to ensure partial deadlock freedom, which will be illustrated in more details in Section 2. We believe that this idea is intrinsically novel compared with previous type systems not only for concurrent calculi but also for functional programming languages.

2. The Notion of Partial Deadlock-freedom – Because it limits its expressive power too much to require an entire concurrent calculus to be deadlock-free (in fact, the only known previous type system that ensures deadlock-freedom is for a very restrictive process calculus[2] where name passing is not allowed at all), it is more realistic to require a concurrent calculus to be only *partially* deadlock-free. This paper is the first to define what partial deadlock-freedom means in a precise way, setting up a goal for future more advanced type systems for deadlock-freedom.

3. A Type System for Partial Deadlock-freedom – By using posets of type bindings as type environments, we developed a type system that ensures partial deadlock-freedom. As far as the author knows, this is the first type system that ensures (partial) deadlock-freedom for name passing process calculi.

1.3 Applications

An immediate application of our type system is to let programmers or debuggers know whether their programs may fall into deadlock or not, and if so, *where* they may do so. Our type system also gives a specification of programs, because, given a typing $\Gamma \vdash e$ of the code fragment e, we can know on which channels the process e may be blocked. For example, let $?^1[int]$ be a type of channels to receive an integer, and $!^1[int]$ a type of channels to send an integer. If a process e is well typed under $x : ?^1[int]; y : !^1[int]$, we know that e may be blocked to receive

an integer from the channel x first, and then may be blocked to send an integer to the channel y. Thus, our type system gives a better specification of concurrent programs than previous type systems. As a theoretical application, we show in Section 6 that we can reason from type information that functions realized by concurrency primitives (in the style of Milner's "functions as processes"[11]) will eventually return the unique result or fall into an infinite loop (i.e., evaluation never gets stuck as ensured by ordinary typed *functional* languages), which has previously been possible only by looking at the code in detail. Other potential applications – though we have not yet seriously studied them and will need some extensions of our type system – include

1. Aggressive code optimization – for example, in order to compile concurrent execution of e_1 and e_2 for execution on the same processor, we can use the above described information to decide (statically) which process should be scheduled first. This optimization can potentially enable implementation of channels using registers, execution of multiple threads on a single stack,[2] etc.

2. Applications to run-time systems – Debuggers can benefit from the type system to determine the wrong process when deadlock occurs. Garbage collectors can apply different algorithms to collect garbage channels depending on types of the channels; with the assumptions that all remote references are implemented by communication channels, channels which do not cause deadlock can be collected by a reference counting method rather than a mark and sweep-based method which is costly and difficult in distributed environments (because uncollected cyclic reference can only be created by blocked processes).

1.4 Outline

The rest of this paper is structured as follows. We first informally present basic ideas in Section 2. Then, after introducing basic notations and definitions (Section 3), we present our type system (Section 4), and show its correctness (Section 5). Section 6 shows simple applications of the type system to reasoning about programs. Section 7 describes type check algorithm for the type system. Section 8 discusses extensions of the type system, and type reconstruction. Section 9 discusses related work and Section 10 concludes this paper.

2 Informal Preview of Basic Ideas

Before explaining basic ideas of our type system, we introduce a (simplified) syntax of processes. More formal definitions are found in the next section or in [9]. $x![y]$ sends y along the channel x. $x?[y].P$ waits for a value v along the channel x, binds it to y, and executes $[v/y]P$. $P_1 \mid P_2$ executes P_1 and P_2 in parallel. $(\nu x)\,P$ creates a new channel x and executes P.

2.1 Serialized Linear Channels

One of the key notions to obtain partial deadlock-freedom is a linear channel type[9]. By a *linear* channel, we mean a channel which is used just once for communication (as a result, it is used once for an input operation and once for the corresponding output operation). In a type judgement $x_1 : T_1, \ldots, x_n : T_n \vdash P$ (which should be read "P is a well typed process under the type environment $x_1 : T_1, \ldots, x_n : T_n$), if T_i is a linear channel type, a binding

[2] [24] enables stack-based execution of multiple threads by a clever *run-time* technique, which works well for some typical pattern of concurrent programs. On the other hand, our type system will *statically* be able to judge whether or not and how stack-based execution works for more general concurrent programs.

$x_i : T_i$ corresponds to exactly one use (or, one pair of input/output operations) of the channel x_i in P. Therefore, by introducing ordering between bindings on linear channel types, we can represent in which order each linear channel is used. For example, let $!^1[T]$ and $?^1[T]$ be types of linear channels used respectively for sending and for receiving values of type T. By the type environment (with ordering introduced) $x : ?^1[int]; y : !^1[int]$ (we use a semicolon to represent orderings), we can represent the fact that x is first used for receiving an integer, *and then* y is used for sending an integer. So, a process $x?[z].y![z]$, which receives a value along x and sends it along y, is typed under $x : ?^1[int]; y : !^1[int]$, but is not typed under the similar type environment with the different ordering: $y : !^1[int]; x : ?^1[int]$. Note that ordering between type bindings need not be linear: we can consider a type environment like $x : T_1; (y : T_2, z : T_3)$, which means that x is first used as a value of type T_1, and then y and z are independently used as a value of type T_2 and T_3 respectively. Thus, it is natural to regard a type environment as a *poset* of type bindings. Such posets in type environments help us detect deadlock. Consider processes $x?[z].y![z]$ and $y?[z].x![z]$. The first process is well typed under $x : ?^1[int]; y : !^1[int]$, while the second one is well typed under $y : ?^1[int]; x : !^1[int]$. Since there is a conflict between the orderings (the first environment says that x is used before y, while the second one says that y is used before x), parallel composition: $x?[z].y![z] \mid y?[z].x![z]$ of the two processes cannot be typed under any poset type environment that maps x and y to linear channel types. In fact, the parallel composition is in a deadlock state, hence considered a bad program. Based on this idea, we can develop a new type system which ensures not only that well typed processes never go wrong in ordinary sense, but also that in well typed processes, deadlock is never caused by linear channels.

As a by-product, the new type system can also improve properties of linear channels. In the original linear channel type system[9], it is not ensured that linear channels are really used exactly once because of possibility of the deadlock. For example, x and y in the process $x?[z].y![z] \mid y?[z].x![z]$ can be judged to be linear channels under the type system in [9], but no communication on x and y occurs. If we send the output end of a linear channel in order to receive just one reply message (as in a function invocation or a method invocation for concurrent objects), it was ensured that we never get more than one reply messages, but it was *not* ensured that we will eventually get a reply message. The fact that linear channels may not be used also reduces advantages of linear channels in memory management; linear channels still require garbage collection. With the new type system, if we send the output end of a linear channel, it is ensured that we will eventually get a reply message unless the receiver of the linear channel falls into an infinite loop or uses unreliable channels (i.e., channels except for linear, replicated input, mutex channels described below). As for the memory management, if programs do not use any unreliable channels, we do not need garbage collection for linear channels.

2.2 Replicated Input Channels and Mutex Channels

Since the expressive power of linear channels themselves is too limited, meaningful programs cannot be ensured to be deadlock-free by the type system. We add replicated input channels and mutex channels to improve the expressive power of our process calculus. In particular, combinations of linear and replicated input channels allow us to encode λ-calculus, while combinations of linear, replicated input, and mutex channels allow us to encode typical concurrent objects.

Replicated Input Channels A replicated input channel is a kind of a channel, whose input end is used just once in a replicated input prefix immediately after being created. That is, a replicated input channel x should be created (by the prefix (νx)) and used only in the following

manner:

$$(\nu x)\,(x?^*[y].\,e_1 \mid e_2)$$

where x can be used only for output in e_1, e_2. $x?^*[y].\,e_1$ repeatedly inputs a value z along the channel x, and spawns the process $[z/y]e_1$ (therefore, the replicated input behaves like the definition of recursive process such that $x(y) = e_1$). Then, it is easy to see that replicated input channels satisfy both confluence and deadlock-free properties.

Mutex channels A mutex channel, a special kind of a channel for which only at most one message can be put and a receiver of the message must eventually send another message, is a very useful data structure to realize mutual exclusion in concurrent environments. In this paper, we just consider the following very restricted mutex channels for the simplicity. A more sophisticated treatment of mutex channels is found in [7].

1. A value must be put *immediately* after a mutex channel is created.

   $$(\nu L)\,(L![\tilde{v}] \mid e)$$

2. A process that retrieves a value from the mutex channel must *immediately* put a replacing value.

   $$L?[\tilde{v}].\,(L![\tilde{v}'] \mid e)$$

3 Notational Preliminaries

Our process calculus used here is considered an asynchronous counterpart of the π-calculus (in the sense that synchronous outputs and summations are dropped). We follow the syntax used in [9], except that conditional expressions are omitted. Typing of conditional expressions should be treated in a slightly special manner, but they are omitted in this paper for the clarity of the arguments.

3.1 Expressions

3.1 Definition: The set of process expressions is given by the following syntax (the definition of types is deferred until the next subsection):

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

where $\tilde{y}$ is an abbreviated form of a (possibly null) sequence $y_1, \ldots, y_n$.

As usual, we assume that $\tilde{z}$ are bound by prefixes $x?[\tilde{z}].$ and $x?^*[\tilde{z}].$, and x is bound by $(\nu x : T)$. We identify processes up to renaming of bound variables (i.e., α-conversion). For example, we identify $(\nu x : T)\,x![]$ with $(\nu y : T)\,y![]$. We give prefixes $(x?[\tilde{z}].\,, x?^*[\tilde{z}].\,, (\nu x : T))$ a higher precedence than parallel composition $(\mid)$, so that $x?[].(\nu y : T)\,x![] \mid y![]$ means $(x?[].((\nu y : T)\,x![])) \mid y![]$. We write $(\nu \tilde{x}{:}\tilde{T})\,P$ for $(\nu x_1 : T_1) \cdots (\nu x_n : T_n)\,P$.

3.2 Example: A 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.

3.2 Types

In this paper, we consider channel types as primary type constructors for the simplicity. $p^m[T_1, \ldots, T_n]$ (abbreviated as $p^m[\tilde{T}]$) represents the type of channels which carry values of types $T_1, \ldots, T_n$. p, called *polarity*, describes for which operations the channels can be used, while m, called *multiplicity* or *mode*[3], describes the mode of channel usage. Multiplicities 1, r, and M respectively represents linear, replicated input, and mutex channels described in Section 2. The multiplicity ω represents *unreliable channels*, which can be used any times. For example, $!^1[\tilde{T}]$ is the type of channels which can be used just once for output, and $?^\omega[\tilde{T}]$ is the type of channels which can be used any times for input. Polarities, multiplicities, and types are formally defined as follows.

3.3 Definition [polarities, multiplicities]: The metavariable p ranges over *polarities*, which are subsets of the set of input/output capabilities $\{i, o\}$. We use the abbreviations $\updownarrow = \{i, o\}$, $! = \{o\}$, $? = \{i\}$, and $| = \{\}$. The polarity p is called *even* if p is either $\updownarrow$ or $|$, otherwise it is called *odd*. The metavariable m ranges over the *multiplicities* 1, r, M, and ω.

3.4 Definition [types]:
The set of types is given by the following syntax. By *communicatable value types*, we mean types of values which can be communicated along channels.

T, S *(types)*
- $::=$ $p^m[\tilde{T_c}]$ — if $m = M$, then T_c must not have the multiplicity 1.
- b (base types)

T_c, S_c *(communicatable value types)*
- $::=$ $p^m[\tilde{T_c}]$ — $o \notin p$ if $m = M$, and $i \notin p$ if $m = r$; if $m = M$, then T_c must not have linear multiplicity
- b (base types)

Intuitively, the condition "if $m = M$, then T_c must not have the multiplicity 1" forbids linear channels from being sent along mutex channels, because values written into mutex channels may never be read, which causes deadlock. The other condition "$o \notin p$ if $m = M$, and $i \notin p$ if $m = r$" ensures that the output ends of mutex channels and the input ends of replicated inputs channels can never be passed as first-class values. We often write $\mathbf{0}$ for $(\nu x : \updownarrow^M[\,])(x![])$.

The type $p^m[\tilde{T}]$ is called *even* or *odd* according to its polarity p. The type $p^m[\tilde{T}]$ is also called *linear* if $m = 1$. We use the word "*unlimited*" for types of values which can be used unbounded times. 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 input end cannot be used multiple times, hence $p^r[\tilde{T}]$ is unlimited if and only if $i \notin p$.

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

- T is a base type b,
- $T = p^\omega[\tilde{T}]$,
- $T = p^r[\tilde{T}]$ and $i \notin p$,
- $T = p^M[\tilde{T}]$ and $o \notin p$, or
- $T = |^1[\tilde{T}]$

[3] It has been called multiplicity in [9], but "*mode*" might be a more suitable name in the context of this paper.

3.6 Definition: The *combination* of two types T_1 and T_2, written $T_1 + T_2$, is defined as follows:

$$\begin{array}{lcl} b + b & = & b \\ p^{\omega}[\tilde{T}] + q^{\omega}[\tilde{T}] & = & (p \cup q)^{\omega}[\tilde{T}] \\ p^{1}[\tilde{T}] + q^{1}[\tilde{T}] & = & (p \cup q)^{1}[\tilde{T}] \ \text{ if } p \cap q = \emptyset \\ p^{r}[\tilde{T}] + q^{r}[\tilde{T}] & = & (p \cup q)^{r}[\tilde{T}] \ \text{ if } i \notin p \cap q \\ p^{M}[\tilde{T}] + q^{M}[\tilde{T}] & = & (p \cup q)^{M}[\tilde{T}] \ \text{ if } o \notin p \cap q \end{array}$$

For other cases, $T_1 + T_2$ is undefined. In particular, $p^{m_1}[\tilde{T}] + q^{m_2}[\tilde{T}]$ is undefined if m_1 is not the same as m_2.

3.3 Type Environments

We introduce a type environment as a poset of type bindings, and define several operations.

3.7 Definition: A type environment is a triple $(X, \leq, \psi)$ where X is a set of variables, $\leq$ ($\subseteq X \times X$) is a partial-order, and ψ is a map from the set X to types.

If $\Gamma = (X, \leq, \psi)$ is a type environment, we refer to $\leq$ by $\leq_\Gamma$. We often write $x : T$ for $(\{x\}, \{(x,x)\}, \{x \mapsto T\})$, and $\emptyset$ for $(\emptyset, \leq, \psi)$. If $\Gamma = (X, \leq, \psi)$ and $x \in X$, then we often write $\Gamma(x)$ for $\psi(x)$, and $dom(\Gamma)$ for X. The partial-order in a type environment represents in which order each variable will be used. For example, a type environment $(\{x,y\}, \{(x,y),(x,x),(y,y)\}, \{x \mapsto ?^1[int], y \mapsto !^1[int]\})$, written as $x : ?^1[int]; y : !^1[int]$, means that "A channel x should be used for receiving an integer, and then a channel y should be used for sending an integer." Because an unlimited binding may represent multiple use of the variable, we ignore a partial-order on variables labelled with unlimited types. Ordering on variables labelled with $\updownarrow^1[\tilde{T}]$ also need not be kept, because those channels are used up inside, and cannot be used outside, which means that their ordering does not affect the possibility of deadlock with the outside environment. We also ignore a partial-order on replicated input channels and mutex channels; instead we will add appropriate side conditions (which can be considered essentially syntactic restrictions) to typing rules so that they do not cause deadlock. Therefore, we only need to consider type environments that are *canonical* in the following sense.

3.8 Definition [canonical type environment]: A type environment $(X, \leq, \psi)$ is *canonical* if for any $(x,y) \in \leq$ such that $x \neq y$, both $\psi(x)$ and $\psi(y)$ are linear and odd (i.e., either of the form $!^1[\tilde{T}]$ or $?^1[\tilde{T}]$).

For any type environment, we can obtain the corresponding canonical type environment by forgetting unnecessary ordering. Therefore, we consider only canonical type environments below.

3.9 Definition: The *canonical form* $\mathcal{N}(\Gamma)$ of type environment $\Gamma = (X, \leq, \psi)$ is given by:

$$\mathcal{N}(\Gamma) = (X, \{(x,y) \mid x = y;\ \textit{or}\ x \leq y \ \textit{ and } \ \psi(x), \psi(y)\ \textit{are linear and odd.}\ \}, \psi)$$

We identify type environments Γ_1 and Γ_2 if $\mathcal{N}(\Gamma_1)$ and $\mathcal{N}(\Gamma_2)$ are identical.

We define several operations on (canonical) type environments below. The following *parallel combination* of two type environments plays an important role in defining composability of two concurrent processes. We require not only the channel-wise consistency of communication (consistency of the type of values sent along each channel, I/O mode, channel usage mode), but also the consistency of ordering of channels. Therefore, $(x : ?^1[int]; y : !^1[int]) + (x : !^1[int]; y : !^1[int])$ is well defined as $x : \updownarrow^1[int], y : \updownarrow^1[int]$, but $(x : ?^1[int]; y : !^1[int]) + (y : !^1[int]; x : !^1[int])$ is undefined.

3.10 Definition: Let $\Gamma_1 = (X_1, \leq_1, \psi_1)$ and $\Gamma_2 = (X_2, \leq_2, \psi_2)$ be canonical type environments. We define $\Gamma_1 + \Gamma_2 = \mathcal{N}(X, \leq, \psi)$ by

$$
\begin{array}{l}
X = X_1 \cup X_2 \\
\leq = (\leq_1 \cup \leq_2)^+ \text{ (if } (\leq_1 \cup \leq_2)^+ \text{ is not a partial-order, then } \Gamma_1 + \Gamma_2 \text{ is undefined)} \\
\psi(x) = \left\{ \begin{array}{ll} \psi_1(x) + \psi_2(x) & \text{if } x \in X_1 \cap X_2 \\ \psi_1(x) & \text{if } x \in X_1 \backslash X_2 \\ \psi_2(x) & \text{if } x \in X_2 \backslash X_1 \end{array} \right.
\end{array}
$$

where R^+ is a transitive closure of R.

3.11 Example: $(x : ?^1[\tilde{T}]; y : !^1[\tilde{T}]) + (y : ?^1[\tilde{T}]; z : !^1[\tilde{T}]) = (x : ?^1[\tilde{T}]; z : !^1[\tilde{T}]), y : \updownarrow^1[\tilde{T}]$

3.12 Example: $(x : ?^1[\tilde{T}]; y : !^1[\tilde{T}]) + (y : ?^1[\tilde{T}]; x : !^1[\tilde{T}])$ is undefined, because $(\{(x,x),(x,y),(y,y)\} \cup \{(x,x),(y,x),(y,y)\})^+ = \{(x,x),(x,y),(y,x),(y,y)\}$ is not a partial order (since it does not satisfy the antisymmetric law).

As is already used in the examples above, we write Γ_1, Γ_2 if there is no ordering between bindings in Γ_1 and those in Γ_2. It is formally defined as follows.

3.13 Definition: Let $\Gamma_1 = (X_1, \leq_1, \psi_1)$ and $\Gamma_2 = (X_2, \leq_2, \psi_2)$ be canonical type environments. If $X_1 \cap X_2 = \emptyset$, we define *independent combination*: $\Gamma_1, \Gamma_2 = (X, \leq, \psi)$ by

$$
\begin{array}{l}
X = X_1 \cup X_2 \\
\leq = \leq_1 \cup \leq_2 \\
\psi(x) = \left\{ \begin{array}{ll} \psi_1(x) & \text{if } x \in X_1 \\ \psi_2(x) & \text{if } x \in X_2 \end{array} \right.
\end{array}
$$

The following *sequential combination* of type environments is another important operation. $\Gamma_1; \Gamma_2$ means that bindings of linear channels in Γ_1 must be used before those in Γ_2. The sequential combination $x : ?^1[int]; x : !^1[int]$ is undefined because both ends of linear channels must be simultaneously used.

3.14 Definition: Let $\Gamma_1 = (X_1, \leq_1, \psi_1)$ and $\Gamma_2 = (X_2, \leq_2, \psi_2)$ be canonical type environments. If $\{x \mid x \in X_1 \text{ and } \psi_1(x) \text{ is linear and odd}\} \cap \{x \mid x \in X_2 \text{ and } \psi_2(x) \text{ is linear and odd}\} = \phi$, we define a *sequential combination* $\Gamma_1; \Gamma_2 = (X, \leq, \psi)$ by

$$
\begin{array}{l}
X = X_1 \cup X_2 \\
\leq = \leq_1 \cup \leq_2 \cup \{(x,y) \mid x \in X_1, y \in X_2, \\
\qquad\qquad \textit{and both } \psi_1(x_1), \psi_2(x_2) \textit{ are linear and odd } \} \\
\psi(x) = \left\{ \begin{array}{ll} \psi_1(x) + \psi_2(x) & \text{if } x \in X_1 \cap X_2 \\ \psi_1(x) & \text{if } x \in X_1 \backslash X_2 \\ \psi_2(x) & \text{if } x \in X_2 \backslash X_1 \end{array} \right.
\end{array}
$$

3.15 Definition: Let $\Gamma = (X, \leq, \psi)$ be a canonical type environment. If $x \in X$, a restriction $\Gamma \backslash x = (X', \leq', \psi')$ is defined by:

$$
\begin{array}{l}
X' = X \backslash \{x\} \\
\leq' = \leq \cap (X' \times X') \\
\psi'(x) = \psi(x) \text{ for } x \in X'
\end{array}
$$

Once a linear channel x of type $\updownarrow^1[\tilde{T}]$ is used, the binding should become $|^1[\tilde{T}]$. The following operators on types and type environments represent this change of types, which will later be used to show correctness of the type system.

3.16 Definition [removal of capability]: Let T be a type. The type T^- is defined by:

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

3.17 Definition [removal of capability from type environments]: Let $\Gamma = (X, \leq, \psi)$ be a type environment. We write $\Gamma - x$ for $\mathcal{N}(X, \leq, \psi\{x \mapsto \Gamma(x)^-\})$.

3.18 Example: Let $\Gamma = x : \updownarrow^1[\tilde{T}], y : \updownarrow^M[\tilde{T}]$. Then, $\Gamma - x = x : |^1[\tilde{T}], y : \updownarrow^M[\tilde{T}]$, while $\Gamma - y = \Gamma$.

Technical Properties

3.19 Lemma:
1. Operators: "+"(parallel combination), ","(independent combination) are commutative and associative. ";"(sequential combination) is associative.
2. If the independent combination Γ_1, Γ_2 is well defined, then so is $\Gamma_1 + \Gamma_2$ and $\Gamma_1 + \Gamma_2 = \Gamma_1, \Gamma_2$.
3. If $(x : T_1; \Gamma_1) + (x : T_2; \Gamma_2)$ is well defined, then $(x : T_1; \Gamma_1) + (x : T_2; \Gamma_2) = x : (T_1 + T_2); (\Gamma_1 + \Gamma_2)$
4. If T is not linear and odd, then $x : T; \Gamma = \Gamma; x : T = x : T + \Gamma$.

Proof: Trivial from the definitions. □

4 Typing Rules

A type judgement is of the form $\Gamma \vdash P$, where Γ is a type environment and P is a process expression. It should be read "P is a well typed process under the type environment Γ." The partial-order in Γ represents in which order each linear channel will be used in P. For example, $x : ?^1[int]; y : !^1[int] \vdash P$ implies that x is used for input and *then* y is used for output in the process P. $\tilde{z}{:}\tilde{T}$ represents the independent combination $z_1 : T_1, \ldots, z_n : T_n$.

As in [9], the following (T-PAR) is the key rule. The rule should be read "if P_1, P_2 are respectively well typed under Γ_1, Γ_2, *and if* $\Gamma_1 + \Gamma_2$ *is well defined*, then $P_1 \,|\, P_2$ is well typed under $\Gamma_1 + \Gamma_2$."

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

The ordering between x and $y_1, \ldots, y_n$ in the following rule represents the fact that $y_1, \ldots, y_n$ can be used only after communication on x.

$$\frac{\Gamma \quad \textit{unlimited}}{\Gamma + (x : !^m[\tilde{T}]; (y_1 : T_1 + \cdots + y_n : T_n)) \vdash x![y_1, \ldots, y_n]} \qquad \text{(T-OUT)}$$

The following rule indicates that the body of an input prefix is invoked only after the input is performed. Note that there must be no dependency between Γ and $\tilde{z}{:}\tilde{T}$. In addition, we require that no input operation can be performed either before releasing mutex channels or before binding the input end of replicated input channels, so that the holder of a mutex channel

must immediately release it, and so that the input end of a replicated input channel is bound immediately after it is created.[4]

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P \qquad m = 1 \text{ or } \omega \qquad \text{There is no } y \text{ such that } \Gamma(y) = p^M[\tilde{U}](o \in p), \text{ or } p^r[\tilde{U}](i \in p)}{x : ?^m[\tilde{T}]; \Gamma \vdash x?[\tilde{z}].P} \tag{T-In}$$

If x is a mutex channel, we must check that a receiver will output a value along x (so that other processes waiting for a message along x cannot be blocked).

$$\frac{\Gamma, x : p^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash P \qquad o \in p \qquad \text{There is no } y \text{ such that } \Gamma(y) = q^M[\tilde{U}](o \in q), \text{ or } q^r[\tilde{U}](i \in q)}{x : ?^M[\tilde{T}], \Gamma \vdash x?[\tilde{z}].P} \tag{T-MIn}$$

The environment Γ used in the body of replicated input prefix must be unlimited, because the body may be executed unbounded times.

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P \qquad \Gamma \text{ unlimited}}{x : ?^r[\tilde{T}]; \Gamma \vdash x?^*[\tilde{z}].P} \tag{T-RIn}$$

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

4.1 Example: Let $\emptyset$ be an empty context. Then,

$$\emptyset \vdash (\nu x : \updownarrow^1[\,])(\nu y : \updownarrow^1[\,])\,(x?[\,].y![\,] \mid x![\,] \mid y?[\,].\mathbf{0}),$$

but

$$\emptyset \not\vdash (\nu x : \updownarrow^1[\,])(\nu y : \updownarrow^1[\,])\,(x?[\,].y![\,] \mid y?[\,].x![\,]).$$

Technical properties

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

Proof: From the typing rules, $x : T$ must be introduced in (T-Out). Therefore, T must be unlimited and we can obtain $\Gamma \vdash P$ from a proof of $\Gamma, x : T$ by dropping all occurrences of $x : T$. □

4.3 Lemma [Weakening]: Let Γ be a type environment and T an unlimited type such that $\Gamma + x : T$ is well defined. Then $\Gamma \vdash P$ implies $\Gamma + x : T \vdash P$.

Proof: Straightforward induction on typing derivations. □

4.4 Lemma [Renaming]: The rule

$$\frac{\Gamma, x : T \vdash P}{\Gamma + y : T \vdash [y/x]P} \tag{E-Ren}$$

is admissible; that is, if $\Gamma, x : T \vdash P$ and $\Gamma + y : T$ is well defined, then $\Gamma + y : T \vdash [y/x]P$.

Proof: (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). □

[4] These conditions could be relaxed by extending the type system. See Section 8 for the discussions.

5 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 channels judged to be linear, replicated input, or mutex by the type system do not cause deadlock.

5.1 Reduction Semantics

Following [12, 21], the operational semantics is defined via two relations: a structural congruence and a reduction relation.

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

5.1 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.

5.2 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, or replicated input.

5.3 Lemma: For any process P, there is some Q in normal form such that $P \equiv Q$.

Proof: By the associativity and commutativity of $\mid$, 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 the same as the standard one[12, 21], except that it is annotated with special tags (as in [9]), to state the subject reduction property, etc. The reduction relation is of the form $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. Intuitively, the type environment Γ represents capability which can be used in P, while Γ' is the type environment obtained from Γ by removing the capability used in the reduction. Note that for well typed processes, the reduction relation defined below coincides with the standard reduction relation for untyped process calculi.

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

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

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

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

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

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 $\Gamma \vdash P \not\longrightarrow$ if there is no Q such that $\Gamma \vdash P \longrightarrow Q$.

Partial Confluence

As claimed in [9], communication on linear or replicated input channels enjoys the confluence property. It implies that communication on those channels essentially preserves equivalence class of processes. We do not show any proof here since it is the same as that in [9].

5.4 Theorem [partial confluence]: Suppose that $\Gamma \vdash P$ and $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$ for $m = 1$ or r. 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 $\Delta \vdash Q \xrightarrow{m} Q' \dashv \Delta'$.

5.2 Partial Deadlock Freedom

The subject reduction below ensures that well-typedness of process expressions is kept during reduction. The difference to the ordinary subject reduction is that the process expression after reduction 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.[5]

5.5 Theorem [subject reduction]: If $\Gamma \vdash P$ and $\Gamma \vdash P \longrightarrow P' \dashv \Gamma'$, then $\Gamma' \vdash P'$.

We need several lemmas to prove Theorem 5.5.

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

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 "+" (Lemma 3.19). 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 $\Gamma \vdash (\nu x:T)\, P_1 \mid P_2$. Then there are some Γ_1, Γ_2 such that $\Gamma_1 \vdash (\nu x:T)\, P_1$ and $\Gamma_2 \vdash P_2$, with $\Gamma_1 + \Gamma_2 = \Gamma$. By (T-New) (the only rule by which $\Gamma_1 \vdash (\nu x:T)\, P_1$ can be derived), $\Gamma_1, x:T \vdash P_1$. By (T-Par), $(\Gamma_1, x:T) + \Gamma_2 \vdash P_1 \mid P_2$, that is, $\Gamma, x:T \vdash P_1 \mid P_2$ (because we can assume that $x \notin dom(\Gamma_2)$ by appropriate renaming). By (T-New), $\Gamma \vdash (\nu x:T)\,(P_1 \mid P_2)$.

On the other hand, suppose that $\Gamma \vdash (\nu x:T)\,(P_1 \mid P_2)$. Then $\Gamma, x:T \vdash P_1 \mid P_2$. By (T-Par), there are some Γ_1, Γ_2 such that $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$, with $\Gamma_1 + \Gamma_2 = \Gamma, x:T$. From the fact that x is not free in P_2 and Lemmas 4.3 and 4.2, we can assume that $\Gamma_1(x) = T, x \notin dom(\Gamma_2)$. By (T-New) and (T-Par), we obtain $\Gamma_1 \backslash x + \Gamma_2 \vdash (\nu x{:}T)\, P_1 \mid P_2$, that is, $\Gamma \vdash (\nu x{:}T)\, P_1 \mid P_2$. □

5.7 Lemma: If $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$ is derivable by n steps of inferences, and $\Gamma + \Delta$ is well defined, then $\Gamma + \Delta \vdash P \xrightarrow{m} P' \dashv \Gamma' + \Delta$ is also derivable by n steps of inferences.

[5] In the presence of conditional expressions or choice expressions, we need to add the following rule to strengthen the ordering:

$$\frac{(X, \leq, \psi) \vdash P \qquad \leq \subseteq \leq'}{(X, \leq', \psi) \vdash P}$$

to the typing rules in order to make the subject reduction valid, provided that the typing rule for choice looks like:

$$\frac{(X, \leq_1, \psi) \vdash P \qquad (X, \leq_2, \psi) \vdash Q}{(X, (\leq_1 \cup \leq_2)^*, \psi) \vdash P + Q}$$

Proof: Induction on the number of steps of inferences for $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. The base cases (cases for the rules (R-Com) and (R-RCom)) follow from the fact that if $\updownarrow^m[\tilde{T}] + S$ is well defined, then $(\updownarrow^m[\tilde{T}] + S)^- = (\updownarrow^m[\tilde{T}])^- + S$. Induction steps are straightforward. □

5.8 Lemma: If $\Gamma_1 + \Gamma_2 \vdash P \xrightarrow{m} P' \dashv \Gamma'$ is derivable by n steps of inferences, and $\Gamma_1 \vdash P$, then $\Gamma_1 \vdash P \xrightarrow{m} P' \dashv \Gamma_1'$ is also derivable by n steps of inferences and $\Gamma' = \Gamma_1' + \Gamma_2$ for some Γ_1'.

Proof: Induction on the number n of steps of inferences for $\Gamma_1 + \Gamma_2 \vdash P \xrightarrow{m} P' \dashv \Gamma'$, with case analysis on the last rule used. 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 $\Gamma_1 + \Gamma_2 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma'$ is derivable by $n-1$ steps. From $\Gamma_1 \vdash P_1 \mid P_2$, we have

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

Since $\Gamma_1 + \Gamma_2 = \Gamma_3 + (\Gamma_4 + \Gamma_2)$, $\Gamma_3 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_3'$ is derivable by $n-1$ steps and $\Gamma_3' + (\Gamma_4 + \Gamma_2) = \Gamma'$ for some Γ_3' by induction hypothesis. By Lemma 5.7, $\Gamma_3 + \Gamma_4 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_3' + \Gamma_4$ is also derivable by $n-1$ steps. Let $\Gamma_1' = \Gamma_3' + \Gamma_4$. Then, $\Gamma_1 \vdash P_1 \mid P_2 \xrightarrow{m} P' \dashv \Gamma_1'$ with $\Gamma_1' + \Gamma_2 = \Gamma'$ is derivable by n steps. □

Proof of Theorem 5.5: Suppose $\Gamma \vdash P$ and $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$. We show $\Gamma' \vdash P'$ by induction on the number of steps of inferences in the proof of $\Gamma \vdash P \xrightarrow{m} P' \dashv \Gamma'$, with case analysis on the last rule used.

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

  $$\Gamma_1 \text{ is unlimited.}$$
  $$\Gamma_2 \vdash x?[\tilde{z}].Q$$
  $$(\Gamma_1 + (x : !^m[\tilde{T}]; (y_1 : T_1 + \cdots + y_n : T_n))) + \Gamma_2 = \Gamma$$
  $$\Gamma' = \Gamma - x$$

  for some Γ_1, Γ_2. Without loss of generality (by Lemma 4.3), we can assume that polarities of all bindings in Γ_1 are $\mid$. Case analysis on $m (\in \{1, \omega, M\})$.

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

    $$\Gamma_2 = x : ?^m[\tilde{T}]; \Gamma_3$$
    $$\Gamma_3, \tilde{z}{:}\tilde{T} \vdash Q$$

    for some Γ_3. From the fact $\Gamma' = \Gamma - x = ((\Gamma_1 + (x : !^m[\tilde{T}]; (y_1 : T_1 + \cdots + y_n : T_n))) + \Gamma_2) - x$ and Lemma 3.19, we have

    $$\begin{aligned} \Gamma' &= ((\Gamma_1 + (x : !^m[\tilde{T}]; (y_1 : T_1 + \cdots + y_n : T_n))) + (x : ?^m[\tilde{T}]; \Gamma_3)) - x \\ &= (\Gamma_1 + (x : \updownarrow^m[\tilde{T}]; (y_1 : T_1 + \cdots + y_n : T_n + \Gamma_3))) - x \\ &= (x : \updownarrow^m[\tilde{T}] + (\Gamma_3 + y_1 : T_1 + \cdots + y_n : T_n)) - x \\ &= x : (\updownarrow^m[\tilde{T}])^- + (\Gamma_3 + y_1 : T_1 + \cdots + y_n : T_n). \end{aligned}$$

    The above equation also implies that $\Gamma_3 + y_1 : T_1 + \cdots + y_n : T_n$ is well defined. Therefore, by Lemma 4.4 and $\Gamma_3, \tilde{z}{:}\tilde{T} \vdash Q$, we have

    $$\Gamma_3 + y_1 : T_1 + \cdots + y_n : T_n \vdash [\tilde{y}/\tilde{z}]Q$$

    By weakening (Lemma 4.3),

$$\Gamma_3 + y_1 : T_1 + \cdots + y_n : T_n + x : (\updownarrow^m[\tilde{T}])^- \vdash [\tilde{y}/\tilde{z}]Q$$

That is,

$$\Gamma' \vdash P'$$

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

$$\Gamma_2 = x : ?^M[\tilde{T}];\Gamma_3$$
$$\Gamma_3, x : p^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash Q$$
$$o \in p$$

for some Γ_3. From the fact $\Gamma' = \Gamma - x = \Gamma = ((\Gamma_1 + (x : !^M[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n))) + \Gamma_2)$ and Lemma 3.19, we have

$$\begin{aligned}\Gamma' &= (\Gamma_1 + (x : !^M[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n))) + (x : ?^M[\tilde{T}];\Gamma_3)\\ &= \Gamma_1 + (x : \updownarrow^M[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n + \Gamma_3))\\ &= x : \updownarrow^M[\tilde{T}] + (\Gamma_3 + y_1 : T_1 + \cdots + y_n : T_n)\end{aligned}$$

The above equation also implies that $(\Gamma_3, x : p^M[\tilde{T}]) + y_1 : T_1 + \cdots + y_n : T_n$ is well defined. Therefore, by Lemma 4.4 and $\Gamma_3, x : p^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash Q$, we have

$$\Gamma_3 + x : p^M[\tilde{T}] + y_1 : T_1 + \cdots + y_n : T_n \vdash [\tilde{y}/\tilde{z}]Q$$

By weakening (Lemma 4.3),

$$\Gamma_3 + x : \updownarrow^M[\tilde{T}] + y_1 : T_1 + \cdots + y_n : T_n \vdash [\tilde{y}/\tilde{z}]Q$$

That is,

$$\Gamma' \vdash P'$$

• Case (R-RCOM)
In this case, $P = x![\tilde{y}] \mid x?^*[\tilde{z}].Q$, and $P' = [\tilde{y}/\tilde{z}]Q \mid x?^*[\tilde{z}].Q$. By (T-PAR), (T-OUT) and (T-RIN),

$$\Gamma_2, \tilde{z}{:}\tilde{T} \vdash Q$$
$$(\Gamma_1 + (x : !^r[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n))) + (x : ?^r[\tilde{T}];\Gamma_2) = \Gamma$$
$$\Gamma' = \Gamma - x$$

for some unlimited Γ_1, Γ_2. Without loss of generality (by Lemma 4.3), we can assume that polarities of all bindings in Γ_1 are $\downarrow$. From the fact $\Gamma' = \Gamma - x = \Gamma = ((\Gamma_1 + (x : !^r[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n))) + (x : ?^r[\tilde{T}];\Gamma_2))$ and Lemma 3.19, we have

$$\begin{aligned}\Gamma' &= (\Gamma_1 + (x : !^r[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n))) + (x : ?^r[\tilde{T}];\Gamma_2)\\ &= \Gamma_1 + (x : \updownarrow^r[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n + \Gamma_2))\\ &= x : \updownarrow^r[\tilde{T}] + (\Gamma_2 + y_1 : T_1 + \cdots + y_n : T_n).\end{aligned}$$

The above equation also implies that $\Gamma_2 + y_1 : T_1 + \cdots + y_n : T_n$ is well defined. Therefore, by Lemma 4.4 and $\Gamma_2, \tilde{z}{:}\tilde{T} \vdash Q$, we have

$$\Gamma_2 + y_1 : T_1 + \cdots + y_n : T_n \vdash [\tilde{y}/\tilde{z}]Q$$

By weakening (Lemma 4.3) and $x : ?^r[\tilde{T}];\Gamma_2 \vdash x?^*[\tilde{z}].Q$, we have

$$(\Gamma_2 + y_1 : T_1 + \cdots + y_n : T_n) + (x : ?^r[\tilde{T}];\Gamma_2) + x : !^r[\tilde{T}] \vdash [\tilde{y}/\tilde{z}]Q \mid x?^*[\tilde{z}].Q$$

That is,

$$\Gamma' \vdash P'.$$

(Note that $\Gamma_2 + \Gamma_2 = \Gamma_2$ because Γ_2 is unlimited.)

- 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,

  $$\Gamma_1 \vdash P_1$$
  $$\Gamma_2 \vdash P_2$$
  $$\Gamma_1 + \Gamma_2 = \Gamma$$

  for some Γ_1, Γ_2. By Lemma 5.8, for some Γ_1', $\Gamma_1 \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma_1'$ is derivable by the same number of steps as that for $\Gamma \vdash P_1 \xrightarrow{m} P_1' \dashv \Gamma'$, and $\Gamma' = \Gamma_1' + \Gamma_2$ holds. By induction hypothesis, $\Gamma_1' \vdash P_1'$, which implies $\Gamma_1' + \Gamma_2 \vdash P_1' \mid P_2$. That is, $\Gamma' \vdash P'$.

- Case (R-New) is trivial.

- Case (R-Cong) follows immediately from Lemma 5.6.

□

The following theorem ensures lack of immediate deadlock on linear, replicated input, and mutex channels. We use the shorthand $\{\mid Q\}$ for either nothing or a parallel composition with the process Q. 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 channel or linear channels whose only input or output end is available in P (in the first case below), or (2)all computation has been completed and only processes which correspond to process definitions (replicated input expressions) or stores (i.e., messages put on mutex channels) are left (in the second case below).

5.9 Theorem [lack of immediate deadlock]: Suppose $\Gamma \vdash P$. If $\Gamma \vdash P \not\longrightarrow$, then one of the following conditions holds.

1. either $P \equiv (\nu \tilde{w}{:}\tilde{U})\,(x?[\tilde{z}].Q\{\mid R\})$, or $P \equiv (\nu \tilde{w}{:}\tilde{U})\,(x![\tilde{z}]\{\mid R\})$; and the binding of x (in Γ or $\tilde{w}{:}\tilde{U}$) has the unreliable mode or has odd polarity.
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 $x?^*[\tilde{y}].Q$ (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 5.6, $\Gamma \vdash (\nu \tilde{w}{:}\tilde{U})\,(P_1 \mid \cdots \mid P_n)$. By (T-New) and (T-Par), there exist $\Gamma_1, \ldots, \Gamma_n$ such that

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

Suppose that neither of the above two conditions holds. There are three cases:

- Case 1: $P_i = x![\tilde{y}]$ for some i and $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^r[\tilde{T}]$.
- Case 2: $P_i = x?[\tilde{y}].Q$ for some i and $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^M[\tilde{T}]$.
- Case 3: $P_i = x![\tilde{y}]$ or $P_i = x?[\tilde{y}].Q$ for some i, and $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^1[\tilde{T}]$.

We show that none of the above cases can happen.

- Case 1:
  Note that $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^r[\tilde{T}]$, and $\Gamma_1 + \cdots + \Gamma_n = \Gamma, \tilde{w}{:}\tilde{U}$. From the syntax of communicatable value types and the typing rules, the input end $?^r[\tilde{T}]$ only come from the rule (T-RIN). Therefore, for some $j(1 \leq j \leq n)$, P_j has a subexpression $x?^*[\tilde{z}].Q$. By the side conditions in (T-IN1), (T-IN2) and (T-RIN), it cannot be guarded by any input prefix. Therefore, P_j must be $x?^*[\tilde{z}].Q$, which contradicts the assumption that $\Gamma \vdash P \not\longrightarrow$.

- Case 2:
  Note that $(\Gamma, \tilde{w}{:}\tilde{U})(x) = \updownarrow^M[\tilde{T}]$, and $\Gamma_1 + \cdots + \Gamma_n = \Gamma, \tilde{w}{:}\tilde{U}$. From the syntax of communicatable value types and the typing rules, the output end $!^M[\tilde{T}]$ only come from the rule (T-OUT). Therefore, for some $j(1 \leq j \leq n)$, P_j has a subexpression $x![\tilde{z}]$. By the side conditions in (T-IN1), (T-IN2) and (T-RIN), it cannot be guarded by any input prefix. Therefore, P_j must be $x![\tilde{z}]$, which contradicts the assumption that $\Gamma \vdash P \not\longrightarrow$.

- Case 3:
  Since case 1 and case 2 have already been denied, we can assume that, without loss of generality, $P_1, \ldots, P_k$ are output or input on linear channels, and $P_{k+1}, \ldots, P_n$ are either output on mutex channels: $u![\tilde{z}]$ or input on replicated input channels: $u?^*[\tilde{z}].R$. Let us define the set S by:

  $$\begin{array}{rcl} S & = & \{x \mid P_i = x![\tilde{y}] \text{ for some } i(1 \leq i \leq k)\} \\ & \cup & \{x \mid P_i = x?[\tilde{y}].Q \text{ for some } i(1 \leq i \leq k)\} \end{array}$$

  Let us define the relation $\mathcal{R}(\subseteq S \times S)$ by:

  $$\mathcal{R} \;=\; \{(x,y) \mid x, y \in S \wedge x \leq_{\Gamma_i} y\} \text{ for some } i(1 \leq i \leq k)\}$$

  Then, since $\Gamma_1 + \cdots + \Gamma_k$ is well defined, $\mathcal{R}^*$ is a partial order, that is, there is no $x, y(x \neq y)$ such that $(x,y),(y,x) \in \mathcal{R}^*$. Therefore, there is $x \in S$ such that there is no $y(\neq x)$ such that $(y,x) \in \mathcal{R}$. By $x \in S$, $P_i = x![\tilde{y}]$ or $x?[\tilde{y}].Q$ for some $i(1 \leq i \leq k)$. Since the other case is similar, we show only the case where $P_i = x![\tilde{y}]$. From the fact that $(\Gamma_1 + \cdots + \Gamma_n)(x) = \updownarrow^1[\tilde{T}]$, for some j such that $j(j \neq i, 1 \leq j \leq k)$, $\Gamma_j(x) = ?^1[\tilde{T}]$ (Note here that $\Gamma_l(x) = |^1[\tilde{T}]$ for $l > k$ because of the syntactic restriction of types that ensures the output end of linear channels cannot be passed along mutex channels, and the side condition of (T-RIN)). Because there is no $y(\neq x)$ such that $(y,x) \in \mathcal{R}$, it must be that $P_j = x?[\tilde{z}].Q$, which contradicts $\Gamma \vdash P \not\longrightarrow$.

□

The subject reduction property (Theorem 5.5) together with the lack of immediate deadlock (Theorem 5.9) ensures the following *partial deadlock freedom*. Note that, because multiplicities and even/odd-ness of channel types do not change during reductions, we can judge whether a given typed program falls into deadlock or not, and where it may, by just looking at the initial type environment and type bindings in ν-prefixes.

5.10 Theorem: Suppose that $\Gamma \vdash P$. 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 the binding of x (in Γ or $\tilde{w}{:}\tilde{U}$) has the unreliable mode or has odd polarity.

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 5.5 and Theorem 5.9. Note that the binding of x in Γ' has the unreliable mode or odd polarity if and only if the binding of x in Γ has. □

5.11 Corollary: Suppose that $\Gamma \vdash P$ and that no binding in Γ, P is unlimited or has odd polarity. 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 $x![\tilde{y}]$ where x has the multiplicity M or $x?^*[\tilde{y}].Q$; or
2. $\Delta \vdash Q \longrightarrow Q' \dashv \Delta'$ for some Δ', Q'.

Proof: A special case of Theorem 5.10. □

6 Applications

We can apply the results in the previous section to reasoning about behaviors of concurrent programs. As a simple application, this section shows that we can reason that functional programs realized in terms of concurrency primitives will eventually return the result or fall into an infinite loop *only from the type information.* Therefore, our typed process calculus can capture behavior of typed λ-calculus in the level of the process calculi much better than the untyped calculus[11] and previous typed calculi[16, 9].

6.1 Encoding Call-by-value λ-calculus

We show encoding of call-by-value, simply-typed λ-calculus into our process calculus.[6]

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.

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

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

[6] Though we do not show in this paper, recursion can also be encoded and the same result holds for the extended fragment. More advanced type systems for λ-calculus could also be captured by appropriately extending our process calculus with subtyping, polymorphism, etc.

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

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 [11, 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 the function of type $\tau_1 \to \tau_2$ can be viewed, from the caller of the function, as the output end of a replicated input channel to send a pair consisting of an argument applied to the function and the output end of the channel to get the result.

6.4 Definition: Types are encoded by the following function:

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

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 λ-terms. Then, $[\![\tau]\!]$ is an unlimited type.

Proof: Trivial from the definition of $[\![\tau]\!]$. □

Now we give an encoding of λ-terms. For each λ-term e, $(\!|e|\!)^p_{\mathcal{T}}$ represents a process in our calculus, which evaluates e and returns the result to p. We use a linear channel for the result channel p. First, a variable just returns its name to the return address.

$$(\!|x|\!)^p_{\mathcal{T}} = p![x]$$

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

$$(\!|\lambda x^{\tau_1}.e|\!)^p_{\mathcal{T}} = (\nu y : \updownarrow^r[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]])\,(p![y] \mid y?^*[x, q].\ (\!|\, e \,|\!)^q_{\mathcal{T}, x:\tau_1})$$

where $\tau_2 = (\mathcal{T}, x : \tau_1)(e)$. The function application $e_1 e_2$ first spawns processes to evaluate the function e_1 and the argument e_2, and then applies the argument to the function.[7]

$$(\!|e_1 e_2|\!)^p_{\mathcal{T}} = (\nu y : \updownarrow^1[[\![\tau_1 \to \tau_2]\!]])\,(\nu z : \updownarrow^1[[\![\tau_1]\!]])\,((\!|e_1|\!)^y_{\mathcal{T}} \mid (\!|\, e_2 \,|\!)^z_{\mathcal{T}} \mid y?[f].z?[v].f![v, p])$$

where $\tau_1 = \mathcal{T}(e_2), \tau_2 = \mathcal{T}(e_1 e_2)$.

The following lemma essentially says that well-typed λ-terms are encoded into processes well-typed under the corresponding type environment.

[7] The encoding presented here evaluates a function and its argument in parallel. Another encoding, which evaluates a function and its argument sequentially, is of course possible, and satisfies the same property below.

6.6 Lemma: Let $\mathcal{T} \vdash e : \tau$. Then, $[\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]] \vdash [\![e]\!]^p_{\mathcal{T}}$.

Proof: Induction on the proof of $\mathcal{T} \vdash e : \tau$, with case analysis on the last rule used.

- Case (λT-Var): In this case, $\mathcal{T} = \mathcal{T}', x : \tau$ and $e = x$. Since $[\![\mathcal{T}']\!]$ is unlimited, we have $[\![\mathcal{T}']\!] + (p : !^1[[\![\tau]\!]]; x : [\![\tau]\!]) \vdash p![x]$. Because $[\![\tau]\!]$ is unlimited, we have

$$\begin{array}{ll} & [\![\mathcal{T}']\!] + (p : !^1[[\![\tau]\!]]; x : [\![\tau]\!]) \\ = & [\![\mathcal{T}']\!], x : [\![\tau]\!], p : !^1[[\![\tau]\!]] \\ = & [\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]]. \end{array}$$

- Case (λT-App): In this case, $e = e_1 e_2$, $\mathcal{T} \vdash e_1 : \tau' \to \tau$, and $\mathcal{T} \vdash e_2 : \tau'$. By induction hypothesis,

$$\begin{array}{l} [\![\mathcal{T}]\!], y : !^1[!^r[[\![\tau']\!], !^1[[\![\tau]\!]]]] \vdash [\![e_1]\!]^y_{\mathcal{T}} \\ [\![\mathcal{T}]\!], z : !^1[[\![\tau']\!]] \vdash [\![e_2]\!]^z_{\mathcal{T}}, \end{array}$$

  From the rules (T-In) and (T-Out), we have

$$y : ?^1[!^r[[\![\tau']\!], !^1[[\![\tau]\!]]]]; z : ?^1[[\![\tau']\!]]; p : !^1[[\![\tau]\!]] \vdash y?[f].z?[v].f![v, p].$$

  From the above facts and $[\![\tau' \to \tau]\!] = !^r[[\![\tau']\!], !^1[[\![\tau]\!]]]$,

$$\begin{array}{l} ([\![\mathcal{T}]\!], y : !^1[[\![\tau' \to \tau]\!]]) + ([\![\mathcal{T}]\!], z : !^1[[\![\tau']\!]]) \\ \quad + (y : ?^1[[\![\tau' \to \tau]\!]]; z : ?^1[[\![\tau']\!]]; p : !^1[[\![\tau]\!]]) \\ \qquad \vdash [\![e_1]\!]^y_{\mathcal{T}} \mid [\![e_2]\!]^z_{\mathcal{T}} \mid y?[f].z?[v].f![v, p]. \end{array}$$

  That is,

$$[\![\mathcal{T}]\!], y : \updownarrow^1[[\![\tau' \to \tau]\!]], z : \updownarrow^1[[\![\tau']\!]], p : !^1[[\![\tau]\!]] \vdash [\![e_1]\!]^y_{\mathcal{T}} \mid [\![e_2]\!]^z_{\mathcal{T}} \mid y?[f].z?[v].f![v, p].$$

  By (T-New), we have

$$[\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]] \vdash [\![e_1 e_2]\!]^p_{\mathcal{T}}.$$

- Case (λT-Abs): In this case, $e = \lambda x^{\tau_1}.e'$, $\tau = \tau_1 \to \tau_2$, and $\mathcal{T}, x : \tau_1 \vdash e' : \tau_2$. By induction hypothesis,

$$[\![\mathcal{T}]\!], x : [\![\tau_1]\!], q : !^1[[\![\tau_2]\!]] \vdash [\![e']\!]^q_{\mathcal{T}, x:\tau_1}$$

  By (T-RIn), we have

$$[\![\mathcal{T}]\!], y : ?^r[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]] \vdash y?^*[x, q].\ [\![e']\!]^q_{\mathcal{T}, x:\tau_1}$$

  for fresh y. By the fact $[\![\tau]\!] = [\![\tau_1 \to \tau_2]\!] = !^r[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]]$, we also have

$$p : !^1[[\![\tau]\!]]; y : !^r[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]] \vdash p![y].$$

  From (T-Par) and (T-New), we obtain

$$[\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]] \vdash (\nu y : \updownarrow^r[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]])\,(p![y] \mid y?^*[x, q].\ [\![e]\!]^q_{\mathcal{T}, x:\tau_1})$$

  That is, $[\![\mathcal{T}]\!], p : !^1[[\![\tau]\!]] \vdash [\![\lambda x^{\tau_1}.e]\!]^p_{\mathcal{T}}$.

□

We obtain the following theorem by Theorem 5.10 and the above lemma. 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 can also know from Theorem 5.4 that the returned result is unique up to the appropriate typed process congruence[9].

6.7 Theorem: Let e be a closed $\lambda^{\rightarrow}$-term (i.e., $\emptyset \vdash e : \tau$), and suppose that $p : !^1[[\![\tau]\!]] \vdash [\![e]\!]^p_{\emptyset} \longrightarrow^* P \dashv \Delta$ 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: By the definition of the reduction relation, $\Delta = p : !^1[[\![\tau]\!]]$. Suppose that $\Delta \vdash P \not\longrightarrow$. Since all bindings bound by ν-prefixes in $[\![e]\!]^p_{\emptyset}$ are either linear or replicated input, we have either

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

by Theorem 5.10. By the subject reduction theorem (Theorem 5.5), $p : !^1[[\![\tau]\!]] \vdash P$. Therefore, the second case cannot happen, because the typing rules require that the type environment of P must be unlimited. □

6.2 Reference Cells

As an example of the use of mutex channels, we show an encoding of reference cells as an instance of concurrent objects. Because concurrent objects are in general realized by a combination of linear, replicated input, and general mutex channels, we can ensure partial deadlock-freedom for concurrent object-oriented programs if we extend our type system to relax conditions for mutex channels. Such an extension and encoding of more general concurrent objects are described in [7]. As is shown in [18], reference cells accessed by multiple processes can be represented as a parallel combination of a channel, which keeps the current value, and two processes, each of which serves read/write requests. The following process receives requests from the channel new, creates a new reference cell with the initial value $init$, and returns two channels to send read/write requests:

$$
\begin{array}{l}
new?^*[init, r]. \\
\quad ((\nu l : \updownarrow^M[T])\,(\nu read : \updownarrow^r[!^1[T]])\,(\nu write : \updownarrow^r[T, !^1[]]) \\
\qquad (l![init] \qquad\qquad (* \text{ stores the initial value } *) \\
\qquad \,|\, r![read, write] \qquad (* \text{ returns channels to send requests } *) \\
\qquad \,|\, read?^*[rep].l?[x].(l![x] \,|\, rep![x]) \\
\qquad\qquad\quad (* \text{ if read request arrives, then this reads the current value } *) \\
\qquad\qquad\quad (* \text{ and sends it back to } rep *) \\
\qquad \,|\, write?^*[v, rep].l?[x].(l![v] \,|\, rep![]) \\
\qquad\qquad\quad (* \text{ if write request arrives, then this replaces the current value } *) \\
\qquad\qquad\quad (* \text{ and sends back an acknowledgment to } rep *) \\
\qquad) \\
\quad)
\end{array}
$$

Thus, simply-type λ-calculus extended with reference cells is also encoded into deadlock free fragment. Therefore, we can again reason from the type system, for example, that encoding of a closed term of the extended calculus will eventually return a result or fall into an infinite loop.

7 Type Check

This section presents a type check algorithm for our type system. The algorithm is fairly straightforward. Consider the relation $\Gamma_1 \vdash P \overset{\Gamma_2}{\Rightarrow} \Gamma_3$ where Γ_1 is the input type environment, Γ_2 is the environment used in P, and Γ_3 is the environment obtained by removing the used bindings from Γ_1 and adding constraints on ordering of channel usage paused by P. For example, the relation: $x : \updownarrow^1[\,], y : \updownarrow^1[\,] \vdash x?[\,].y![\,] \overset{x:?^1[\,];y:!^1[\,]}{\Rightarrow} x : !^1[\,]; y : ?^1[\,]$ expresses the fact that, given the type environment $x : \updownarrow^1[\,], y : \updownarrow^1[\,]$, $x?[\,].y![\,]$ consumes $x : ?^1[\,]; y : !^1[\,]$ and outputs the rest of the environment: $x : !^1[\,]; y : ?^1[\,]$. The type check algorithm takes Γ_1 and P as inputs, and outputs Γ_2 and Γ_3 such that $\Gamma_1 \vdash P \overset{\Gamma_2}{\Rightarrow} \Gamma_3$. For example, the rule for parallel composition is expressed as follows:

$$\frac{\Gamma_1 \vdash P_1 \overset{\Gamma_2}{\Rightarrow} \Gamma_4 \quad \Gamma_4 \vdash P_2 \overset{\Gamma_5}{\Rightarrow} \Gamma_3}{\Gamma_1 \vdash P_1 \mid P_2 \overset{\Gamma_2 + \Gamma_5}{\Rightarrow} \Gamma_3} \qquad \text{(TCHECK-PAR)}$$

The rule says that, given Γ_1 and $P_1 \mid P_2$, first check P_1 and obtain Γ_2, Γ_4 such that $\Gamma_1 \vdash P_1 \overset{\Gamma_2}{\Rightarrow} \Gamma_4$. Then, by giving Γ_4 and P_2 as inputs, we can obtain Γ_5, Γ_3.

We need several definitions to present the whole set of rules. The type $T_1 - T_2$ represents the resulting type hold by a value of type T_1 after it is used as a value of type T_2. For example, if x initially had the type $\updownarrow^1[\tilde{T}]$ and if it has been used as a value of the type $?^1[\tilde{T}]$ (i.e., if it has been used once in an input prefix), then the type of x is now $\updownarrow^1[\tilde{T}] - ?^1[\tilde{T}] = !^1[\tilde{T}]$.

7.1 Definition: The subtraction of two types, written $T_1 - T_2$, is defined as follows:

$$\begin{array}{lcl}
b - b & = & b \\
p^\omega[\tilde{T}] - q^\omega[\tilde{T}] & = & p^\omega[\tilde{T}] \text{ if } p \supseteq q \\
p^1[\tilde{T}] - q^1[\tilde{T}] & = & (p \backslash q)^1[\tilde{T}] \text{ if } p \supseteq q \\
p^r[\tilde{T}] - q^r[\tilde{T}] & = & ((p \backslash q) \cup (p \cap \{o\}))^r[\tilde{T}] \text{ if } p \supseteq q \\
p^M[\tilde{T}] - q^M[\tilde{T}] & = & ((p \backslash q) \cup (p \cap \{i\}))^M[\tilde{T}] \text{ if } p \supseteq q
\end{array}$$

7.2 Definition [restriction]: Let $\Gamma = (X, \leq, \psi)$ be a type environment. The restriction $\Gamma \backslash x = (X', \leq', \psi')$, is defined as follows:

$$\begin{array}{l}
X' = X \backslash \{x\} \\
\leq' = \leq \cap X \times X \\
\psi'(y) = \psi(y) \text{ for all } y \in X'
\end{array}$$

We often write $\Gamma \backslash \tilde{x}$ for $\Gamma \backslash x_1 \cdots \backslash x_n$.

The following operator $\Gamma_1 - \Gamma_2$ removes the used capabilities Γ_2 from Γ_1.

7.3 Definition: Let $\Gamma_1 = (X_1, \leq_1, \psi_1)$ and $\Gamma_2 = (X_2, \leq_2, \psi_2)$ be canonical type environments, and suppose that $X_1 \supseteq X_2$. We define $\Gamma_1 - \Gamma_2 = \mathcal{N}(X, \leq, \psi)$ by

$$\begin{array}{l}
X = X_1 \\
\leq = (\leq_1 \cup \leq_2)^* \text{ (if } (\leq_1 \cup \leq_2)^* \text{ is not a partial-order, then } \Gamma_1 - \Gamma_2 \text{ is undefined)} \\
\psi(x) = \left\{ \begin{array}{ll} \psi_1(x) - \psi_2(x) & \text{if } x \in X_1 \cap X_2 \\ \psi_1(x) & \text{if } x \in X_1 \backslash X_2 \end{array} \right.
\end{array}$$

7.4 Example:

1. $(x:\updownarrow^1[\tilde{T}], y:\updownarrow^1[\tilde{T}], z:?^1[\tilde{T}]) - (x:?^1[\tilde{T}]; y:!^1[\tilde{T}]) = (x:!^1[\tilde{T}]; y:?^1[\tilde{T}]), z:?^1[\tilde{T}]$
2. $(x:?^1[\tilde{T}]; y:!^1[\tilde{T}]; z:!^1[\tilde{T}]) - y:!^1[\tilde{T}] = y:|^1[\tilde{T}], (x:?^1[\tilde{T}]; z:!^1[\tilde{T}])$

7.5 Lemma: Let $\Gamma_1, \Gamma_2, \Gamma_3$ be type environments. Then,

1. $\Gamma_1 - \Gamma_2 - \Gamma_3 = \Gamma_1 - (\Gamma_2 + \Gamma_3)$.
2. if $\Gamma_1 - \Gamma_1$ is well defined, then $\Gamma_1 - \Gamma_1$ is unlimited
3. if $\Gamma_1 - \Gamma_1$ is well defined, then $\Gamma_1 - \Gamma_1 + \Gamma_1 = \Gamma_1$.
4. if $x:T_1;\Gamma_1 - x:T_2;\Gamma_2$ is well defined, then $(x:T_1;\Gamma_1) - (x:T_2;\Gamma_2) = x:(T_1 - T_2);(\Gamma_1 - \Gamma_2)$.

Proof: Trivial from the definition and the fact that $T_1 - T_2 - T_3 = T_1 - (T_2 + T_3)$, etc. □

The whole set of rules is given below. Each rule should be read in a bottom-up manner. For example, the rule (TCHECK-IN) should be read "In order to check $x?[\tilde{z}].P$ under the initial environment Γ, augument Γ with the bindings $\tilde{z}{:}\tilde{T}$ ($\tilde{T}$ can be determined by the type $\Gamma(x)$) and check P under the augmented environment. Let Γ_1 and $\Gamma_2, \tilde{z}{:}\tilde{T}'$ be outputs of type checking of P. Check side conditions that $\tilde{T}'$ must be unlimited, etc. Then, outputs $x:?^m[\tilde{T}];(\Gamma_1\backslash\tilde{z})$ and $\Gamma - x:?^m[\tilde{T}];(\Gamma_1\backslash\tilde{z})$ as results of type checking of $x?[\tilde{z}].P$." Note that the side condition "$\forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_1).(u = z_i \vee \neg(u \leq_{\Gamma_1} z_i \vee z_i \leq_{\Gamma_1} u)))$" in the input and output prefix rules means that there must be no ordering between uses of z and uses of other variables in P.

$$\frac{\Gamma_1 \vdash e_1 \overset{\Gamma_2}{\Rightarrow} \Gamma_4 \quad \Gamma_4 \vdash e_2 \overset{\Gamma_5}{\Rightarrow} \Gamma_3}{\Gamma_1 \vdash e_1 \,|\, e_2 \overset{\Gamma_2+\Gamma_5}{\Rightarrow} \Gamma_3} \qquad \text{(TCHECK-PAR)}$$

$$\Gamma \vdash x![y_1, \ldots, y_n] \overset{x:!^m[\tilde{T}];(y_1:T_1+\cdots+y_n:T_n)}{\Rightarrow} \Gamma - (x:!^m[\tilde{T}];(y_1:T_1+\cdots+y_n:T_n)) \qquad \text{(TCHECK-OUT)}$$

$$\frac{\begin{array}{c}\Gamma, \tilde{z}{:}\tilde{T} \vdash P \overset{\Gamma_1}{\Rightarrow} \Gamma_2, \tilde{z}{:}\tilde{T}' \\ \tilde{T}' \text{ are unlimited} \\ \forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_1).(u = z_i \vee \neg(u \leq_{\Gamma_1} z_i \vee z_i \leq_{\Gamma_1} u))) \\ m = 1 \text{ or } \omega \\ \text{There is no } y \text{ such that } \Gamma_1(y) = p^M[\tilde{U}](o \in p), \text{ or } p^r[\tilde{U}](i \in p)\end{array}}{\Gamma \vdash x?[\tilde{z}].P \overset{x:?^m[\tilde{T}];(\Gamma_1\backslash\tilde{z})}{\Rightarrow} \Gamma - (x:?^m[\tilde{T}];(\Gamma_1\backslash\tilde{z}))} \qquad \text{(TCHECK-IN)}$$

$$\frac{\begin{array}{c}\Gamma\backslash x, x:\updownarrow^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash P \overset{\Gamma_1, x:p^M[\tilde{T}]}{\Rightarrow} \Gamma_2, \tilde{z}{:}\tilde{T}' \qquad o \in p \\ \tilde{T}' \text{ are unlimited} \\ \forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_1).(u = z_i \vee \neg(u \leq_{\Gamma_1} z_i \vee z_i \leq_{\Gamma_1} u))) \\ \text{There is no } y \text{ such that } \Gamma_1(y) = p^M[\tilde{U}](o \in p), \text{ or } p^r[\tilde{U}](i \in p)\end{array}}{\Gamma \vdash x?[\tilde{z}].P \overset{\Gamma_1\backslash\tilde{z}, x:?^M[\tilde{T}]}{\Rightarrow} \Gamma - (\Gamma_1\backslash\tilde{z}, x:?^M[\tilde{T}])} \qquad \text{(TCHECK-MIN)}$$

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P \stackrel{\Gamma_1}{\Rightarrow} \Gamma_2, \tilde{z}{:}\tilde{T}' \qquad \Gamma_1 \backslash \tilde{z}, \tilde{T}' \text{ are unlimited} \quad \forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_1).(u = z_i \vee \neg(u \leq_{\Gamma_1} z_i \vee z_i \leq_{\Gamma_1} u)))}{\Gamma \vdash x?^*[\tilde{z}].\, P \stackrel{x:?^r[\tilde{T}];(\Gamma_1\backslash\tilde{z})}{\Rightarrow} \Gamma - (x : ?^r[\tilde{T}];(\Gamma_1\backslash\tilde{z}))} \quad \text{(TCHECK-RIN)}$$

$$\frac{\Gamma, x : p^m[\tilde{T}] \vdash P \stackrel{\Gamma_1}{\Rightarrow} \Gamma_2, x : U \qquad p \text{ is either } \updownarrow \text{ or } | \quad U \text{ is unlimited}}{\Gamma \vdash (\nu x : p^m[\tilde{T}])\, P \stackrel{\Gamma_1\backslash x}{\Rightarrow} \Gamma_2} \quad \text{(TCHECK-NEW)}$$

Using this algorithm, whether $\Gamma \vdash P$ holds or not can be checked by giving the pair of Γ and P as inputs, and checking whether the rest of the environment is unlimited or not (and other subtle conditions). Correspondence between type checking rules and typing rules is formally stated as follows.

7.6 Theorem [Correctness of Type Checking]:

1. If $\Gamma \vdash P$, then $\Gamma \vdash P \stackrel{\Gamma_1}{\Rightarrow} \Gamma_2$ and $\Gamma = \Gamma_1 + \Gamma_2$ for some type environment Γ_1 and unlimtied type environment Γ_2.
2. If $\Gamma \vdash P \stackrel{\Gamma_1}{\Rightarrow} \Gamma_2$, Γ_2 is unlimited, and if $\Gamma = \Gamma_1 + \Gamma_2$, then $\Gamma \vdash P$.

We need several lemmas to prove the theorem.

7.7 Lemma: If $\Gamma_1 \vdash P \stackrel{\Gamma_2}{\Rightarrow} \Gamma_3$, then

1. $\Gamma_2 \vdash P$
2. $\Gamma_1 - \Gamma_2 = \Gamma_3$

Proof: Induction on the proof of $\Gamma_1 \vdash P \stackrel{\Gamma_2}{\Rightarrow} \Gamma_3$, with case analysis on the last rule.

- Case (TCHECK-OUT) is trivial.
- Case (TCHECK-PAR): In this case, $P = P_1 \,|\, P_2$, $\Gamma_1 \vdash P_1 \stackrel{\Gamma_4}{\Rightarrow} \Gamma_5$, $\Gamma_5 \vdash P_2 \stackrel{\Gamma_6}{\Rightarrow} \Gamma_3$, and $\Gamma_2 = \Gamma_4 + \Gamma_6$ for some $\Gamma_4, \Gamma_5, \Gamma_6$. By induction hypothesis,

  $\Gamma_4 \vdash P_1$
  $\Gamma_6 \vdash P_2$
  $\Gamma_5 = \Gamma_1 - \Gamma_4$
  $\Gamma_3 = \Gamma_5 - \Gamma_6$

  By (T-PAR), we obtain $\Gamma_2 (= \Gamma_4 + \Gamma_6) \vdash P_1 \,|\, P_2$. $\Gamma_1 - \Gamma_2 = \Gamma_1 - (\Gamma_4 + \Gamma_6) = \Gamma_1 - \Gamma_4 - \Gamma_6 = \Gamma_3$ by Lemma 7.5.
- Case (TCHECK-IN): In this case,

  $P = x?[\tilde{z}].\, P_1$
  $\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1 \stackrel{\Gamma_4}{\Rightarrow} \Gamma_5, \tilde{z}{:}\tilde{T}'$
  $\tilde{T}'$ are unlimited
  $\Gamma_2 = x : ?^m[\tilde{T}];(\Gamma_4\backslash\tilde{z})$
  $\Gamma_3 = \Gamma_1 - (x : ?^m[\tilde{T}];(\Gamma_4\backslash\tilde{z}))$

$\Gamma_1 - \Gamma_2 = \Gamma_3$ is trivial from the last two conditions above. By induction hypothesis,

$$\Gamma_4 \vdash P_1$$
$$(\Gamma_1, \tilde{z}{:}\tilde{T}) - \Gamma_4 = \Gamma_5, \tilde{z}{:}\tilde{T}'$$

From the side condition:

$$\forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_4).(u = z_i \vee \neg(u \leq_{\Gamma_4} z_i \vee z_i \leq_{\Gamma_1} u))),$$

we have $\Gamma_4 + \tilde{z}{:}\tilde{T}' = \Gamma_4 \backslash \tilde{z}, \tilde{z}{:}\tilde{T}$. Since $\tilde{T}'$ is unlimited, we have $\Gamma_4 \backslash \tilde{z}, \tilde{z}{:}\tilde{T} \vdash P_1$ by weakening (Lemma 4.3). By (T-In), $x : ?^m[\tilde{T}]; (\Gamma_4 \backslash \tilde{z}) \vdash x?[\tilde{z}].P_1$, that is, $\Gamma_2 \vdash P$.

- Case (TCHECK-MIn): In this case,

$$P = x?[\tilde{z}].P_1$$
$$\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1 \overset{\Gamma_4, x:p^M[\tilde{T}]}{\Rightarrow} \Gamma_5, \tilde{z}{:}\tilde{T}'$$
$$o \in p$$
$$\tilde{T}' \text{ are unlimited}$$
$$\Gamma_2 = \Gamma_4 \backslash \tilde{z}, x : ?^M[\tilde{T}]$$
$$\Gamma_3 = \Gamma_1 - (\Gamma_4 \backslash \tilde{z}, x : ?^M[\tilde{T}])$$

$\Gamma_1 - \Gamma_2 = \Gamma_3$ is trivial from the last two conditions above. By induction hypothesis,

$$\Gamma_4, x : p^M[\tilde{T}] \vdash P_1$$
$$(\Gamma_1, \tilde{z}{:}\tilde{T}) - (\Gamma_4, x : p^M[\tilde{T}]) = \Gamma_5, \tilde{z}{:}\tilde{T}'$$

From the side condition:

$$\forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_4).(u = z_i \vee \neg(u \leq_{\Gamma_4} z_i \vee z_i \leq_{\Gamma_1} u))),$$

we have $(\Gamma_4, x : p^M[\tilde{T}]) + \tilde{z}{:}\tilde{T}' = \Gamma_4 \backslash \tilde{z}, x : p^M[\tilde{T}], \tilde{z}{:}\tilde{T}$. Since $\tilde{T}'$ is unlimited, we have $\Gamma_4 \backslash \tilde{z}, x : p^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash P_1$ by weakening (Lemma 4.3). By (T-MIn), $x : ?^M[\tilde{T}], (\Gamma_4 \backslash \tilde{z}) \vdash x?[\tilde{z}].P_1$, that is, $\Gamma_2 \vdash P$.

- Case (TCHECK-RIn): In this case,

$$P = x?^*[\tilde{z}].P_1$$
$$\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1 \overset{\Gamma_4}{\Rightarrow} \Gamma_5, \tilde{z}{:}\tilde{T}'$$
$$\Gamma_1, \tilde{T}' \text{ are unlimited}$$
$$\Gamma_2 = x : ?^r[\tilde{T}]; (\Gamma_4 \backslash \tilde{z})$$
$$\Gamma_3 = \Gamma_1 - (x : ?^r[\tilde{T}]; (\Gamma_4 \backslash \tilde{z}))$$

$\Gamma_1 - \Gamma_2 = \Gamma_3$ is trivial from the last two conditions above. By induction hypothesis,

$$\Gamma_4 \vdash P_1$$
$$(\Gamma_1, \tilde{z}{:}\tilde{T}) - \Gamma_4 = \Gamma_5, \tilde{z}{:}\tilde{T}'$$

From the side condition:

$$\forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_4).(u = z_i \vee \neg(u \leq_{\Gamma_4} z_i \vee z_i \leq_{\Gamma_1} u))),$$

we have $\Gamma_4 + \tilde{z}{:}\tilde{T}' = \Gamma_4 \backslash \tilde{z}, \tilde{z}{:}\tilde{T}$. Since $\tilde{T}'$ is unlimited, we have $\Gamma_4 \backslash \tilde{z}, \tilde{z}{:}\tilde{T} \vdash P_1$ by weakening (Lemma 4.3). By (T-RIn), $x : ?^r[\tilde{T}]; (\Gamma_4 \backslash \tilde{z}) \vdash x?^*[\tilde{z}].P_1$, that is, $\Gamma_2 \vdash P$.

- Case (TCHECK-New): In this case,

$P = (\nu x : p^m[\tilde{T}])\, P_1$
$\Gamma_1, x : p^m[\tilde{T}] \vdash P_1 \stackrel{\Gamma_4}{\Rightarrow} \Gamma_3, x : U$
U is unlimited
$\Gamma_2 = \Gamma_4 \backslash x$

By induction hypothesis,

$\Gamma_4 \vdash P_1$
$(\Gamma_1, x : p^m[\tilde{T}]) - \Gamma_4 = \Gamma_3, x : U$

Note that $\Gamma_1 - \Gamma_2 = \Gamma_1 - \Gamma_4 \backslash x = \Gamma_3$. Because U is unlimited, $\Gamma_4 + x : U = \Gamma_4 \backslash x, x : p^m[\tilde{T}]$. Therefore, by weakening (Lemma 4.3, we have $\Gamma_4 \backslash x, x : p^m[\tilde{T}] \vdash P_1$. By (T-NEW), $\Gamma_4 \backslash x \vdash (\nu x : p^m[\tilde{T}])\, P_1$, that is, $\Gamma_2 \vdash P$

□

7.8 Lemma: 1. If $\Gamma_1 \vdash P \stackrel{\Gamma_2}{\Rightarrow} \Gamma_3$ and if $\Gamma_1 + \Gamma_4$ is well defined, then $\Gamma_1 + \Gamma_4 \vdash P \stackrel{\Gamma_2}{\Rightarrow} \Gamma_3 + \Gamma_4$

2. If $\Gamma_1, \Gamma_4 \vdash P \stackrel{\Gamma_2}{\Rightarrow} \Gamma_3, \Gamma_5, dom(\Gamma_4 = \Gamma_5)$, and if $(\Gamma_6; \Gamma_1), \Gamma_4$ is well defined, then $(\Gamma_6; \Gamma_1), \Gamma_4 \vdash P \stackrel{\Gamma_2}{\Rightarrow} (\Gamma_6; \Gamma_3), \Gamma_5$

Proof: Straightforward induction. □

Proof of Theorem 7.6: 1: Induction on the proof of $\Gamma \vdash P$, with case analysis on the last rule used.

– Case (T-PAR)
In this case,

$P = P_1 \mid P_2$
$\Gamma = \Gamma_1 + \Gamma_2$
$\Gamma_1 \vdash P_1$
$\Gamma_2 \vdash P_2$

By induction hypothesis,

$\Gamma_1 \vdash P_1 \stackrel{\Gamma_3}{\Rightarrow} \Gamma_4$
$\Gamma_2 \vdash P_2 \stackrel{\Gamma_5}{\Rightarrow} \Gamma_6$
$\Gamma_1 = \Gamma_3 + \Gamma_4$
$\Gamma_2 = \Gamma_5 + \Gamma_6$
Γ_4, Γ_6 are unlimited

for some $\Gamma_3, \Gamma_4, \Gamma_5, \Gamma_6$. By Lemma 7.8,

$\Gamma_1 + \Gamma_2 \vdash P_1 \stackrel{\Gamma_3}{\Rightarrow} \Gamma_4 + \Gamma_2$
$\Gamma_2 + \Gamma_4 \vdash P_2 \stackrel{\Gamma_5}{\Rightarrow} \Gamma_6 + \Gamma_4$

By (TCHECK-PAR), we have $\Gamma \vdash P \stackrel{\Gamma_3 + \Gamma_5}{\Rightarrow} \Gamma_6 + \Gamma_4$ and $(\Gamma_3 + \Gamma_5) + (\Gamma_6 + \Gamma_4) = \Gamma$.

– Case (T-OUT)
In this case,

$P = x![\tilde{y}]$
$\Gamma = \Gamma_1 + (x : !^m[\tilde{T}]; (y_1 : T_1 + \cdots + y_n : T_n))$

Let $\Gamma_2 = x : !^m[\tilde{T}];(y_1 : T_1 + \cdots + y_n : T_n), \Gamma_3 = \Gamma_2 - \Gamma_2$. By Lemma 7.5, Γ_3 is unlimited and $\Gamma_2 + \Gamma_3 = \Gamma_2$. By (TCHECK-OUT), $\Gamma_2 \vdash P \overset{\Gamma_2}{\Rightarrow} \Gamma_3$. By Lemma 7.8, $\Gamma \vdash P \overset{\Gamma_2}{\Rightarrow} \Gamma_1 + \Gamma_3$, and $\Gamma_2 + (\Gamma_1 + \Gamma_3) = \Gamma_1 + (\Gamma_2 + \Gamma_3) = \Gamma$.

– Case (T-IN)
In this case,

$P = x?[\tilde{z}].P_1$
$\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1$
$\Gamma = x : ?^m[\tilde{T}];\Gamma_1$

By induction hypothesis,

$\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1 \overset{\Gamma_2}{\Rightarrow} \Gamma_3, \tilde{z}{:}\tilde{T}'$
$\Gamma_1, \tilde{z}{:}\tilde{T} = \Gamma_2 + (\Gamma_3, \tilde{z}{:}\tilde{T}')$
$\Gamma_3, \tilde{T}'$ are unlimited

for some Γ_2, Γ_3. By Lemma 7.8, $(x : ?^m[\tilde{T}];\Gamma_1), \tilde{z}{:}\tilde{T} \vdash P_1 \overset{\Gamma_2}{\Rightarrow} (x : ?^m[\tilde{T}];\Gamma_3), \tilde{z}{:}\tilde{T}'$
The second and the third conditions imply the condition:

$$\forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_2).(u = z_i \vee \neg(u \leq_{\Gamma_1} z_i \vee z_i \leq_{\Gamma_2} u)))$$

Therefore, by (TCHECK-IN) (because other side conditions also hold),

$$x : ?^m[\tilde{T}];\Gamma_1 \vdash P_1 \overset{x:?^m[\tilde{T}];(\Gamma_2\backslash\tilde{z})}{\Rightarrow} (x : ?^m[\tilde{T}];\Gamma_1) - (x : ?^m[\tilde{T}];(\Gamma_2\backslash\tilde{z}))$$

Let $\Gamma_4 = x : ?^m[\tilde{T}];(\Gamma_2\backslash\tilde{z}), \Gamma_5 = (x : ?^m[\tilde{T}];\Gamma_1) - (x : ?^m[\tilde{T}];(\Gamma_2\backslash\tilde{z}))$. Then, the fact that Γ_5 is unlimited follows from the following equations:

$$\begin{aligned}\Gamma_5 &= (x : ?^m[\tilde{T}];\Gamma_1) - (x : ?^m[\tilde{T}];(\Gamma_2\backslash\tilde{z})) \\ &= x : (?^m[\tilde{T}] - ?^m[\tilde{T}]);(\Gamma_1 - \Gamma_2\backslash\tilde{z}) \\ &= x : (?^m[\tilde{T}] - ?^m[\tilde{T}]);\Gamma_3.\end{aligned}$$

Moreover,

$$\begin{aligned}\Gamma_4 + \Gamma_5 &= x : ?^m[\tilde{T}];(\Gamma_2\backslash\tilde{z}) + (x : (?^m[\tilde{T}] - ?^m[\tilde{T}]);\Gamma_3) \\ &= x : (?^m[\tilde{T}] + (?^m[\tilde{T}] - ?^m[\tilde{T}]));(\Gamma_2\backslash\tilde{z} + \Gamma_3) \\ &= x : ?^m[\tilde{T}];\Gamma_1 \\ &= \Gamma\end{aligned}$$

– Case (T-MIN)
In this case,

$P = x?[\tilde{z}].P_1$
$\Gamma_1, x : p^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash P_1$
$o \in p$
$\Gamma = \Gamma_1, x : ?^M[\tilde{T}]$

By the second condition and Lemma 4.3, we have $\Gamma_1, x : \updownarrow^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash P_1$. By induction hypothesis,

$\Gamma_1, x : \updownarrow^M[\tilde{T}], \tilde{z}{:}\tilde{T} \vdash P_1 \overset{\Gamma_2}{\Rightarrow} \Gamma_3, \tilde{z}{:}\tilde{T}'$
$\Gamma_1, x : \updownarrow^M[\tilde{T}], \tilde{z}{:}\tilde{T} = \Gamma_2 + (\Gamma_3, \tilde{z}{:}\tilde{T}')$
$\Gamma_3, \tilde{T}'$ are unlimited

Because $\Gamma_3, \tilde{z}{:}\tilde{T}' = (\Gamma_1, x : \updownarrow^M[\tilde{T}], \tilde{z}{:}\tilde{T}) - \Gamma_2$ is unlimited, Γ_2 can be expressed as $\Gamma_4, x : q^M[\tilde{T}]$ for $o \in q$. Let $\Gamma_5 = \Gamma_4\backslash\tilde{z}, x : ?^M[\tilde{T}]$. By (TCHECK-MIN), we have $\Gamma \vdash P \overset{\Gamma_5}{\Rightarrow} \Gamma - \Gamma_5$. $\Gamma - \Gamma_5$ is unlimited from the following equations:

$$\begin{aligned}\Gamma - \Gamma_5 &= (\Gamma_1, x : ?^M[\tilde{T}]) - (\Gamma_4 \backslash \tilde{z}, x : ?^M[\tilde{T}]) \\ &= (\Gamma_1 - \Gamma_4 \backslash \tilde{z}), x : ?^M[\tilde{T}] \\ &= \Gamma_3 \backslash x, x : ?^M[\tilde{T}]\end{aligned}$$

Moreover,

$$\begin{aligned}\Gamma_5 + (\Gamma - \Gamma_5) &= (\Gamma_4 \backslash \tilde{z}, x : ?^M[\tilde{T}]) + (\Gamma_3 \backslash x, x : ?^M[\tilde{T}]) \\ &= (\Gamma_4 \backslash \tilde{z} + \Gamma_3 \backslash x), x : ?^M[\tilde{T}] \\ &= \Gamma_1, x : ?^M[\tilde{T}] \\ &= \Gamma\end{aligned}$$

– Case (T-RIN)
In this case,

$P = x?^*[\tilde{z}].P_1$
$\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1$
$\Gamma = x : ?^r[\tilde{T}] + \Gamma_1$
Γ_1 is unlimited

By induction hypothesis,

$\Gamma_1, \tilde{z}{:}\tilde{T} \vdash P_1 \stackrel{\Gamma_2}{\Rightarrow} \Gamma_3, \tilde{z}{:}\tilde{T'}$
$\Gamma_1, \tilde{z}{:}\tilde{T} = \Gamma_2 + (\Gamma_3, \tilde{z}{:}\tilde{T'})$
$\Gamma_3, \tilde{T'}$ are unlimited

for some Γ_2, Γ_3. By the condition "$\Gamma_1, \tilde{z}{:}\tilde{T} = \Gamma_2 + (\Gamma_3, \tilde{z}{:}\tilde{T'})$" and the fact that Γ_1 is unlimited, $\Gamma_2 \backslash \tilde{z}$ is also unlimited.

By Lemma 7.8, $x : ?^r[\tilde{T}] + (\Gamma_1, \tilde{z}{:}\tilde{T}) \vdash P_1 \stackrel{\Gamma_2}{\Rightarrow} x : ?^r[\tilde{T}] + (\Gamma_3, \tilde{z}{:}\tilde{T'})$
The second and the third conditions above imply the condition:

$$\forall z_i \in \{\tilde{z}\}.(\forall u \in dom(\Gamma_2).(u = z_i \vee \neg(u \leq_{\Gamma_1} z_i \vee z_i \leq_{\Gamma_2} u)))$$

Therefore, by (TCHECK-RIN),

$$x : ?^r[\tilde{T}] + \Gamma_1 \vdash x?^*[\tilde{z}].P_1 \stackrel{x:?^r[\tilde{T}];\Gamma_2 \backslash \tilde{z}}{\Rightarrow} x : ?^r[\tilde{T}] + \Gamma_1 - (x : ?^r[\tilde{T}]; \Gamma_2 \backslash \tilde{z})$$

Let $\Gamma_4 = x : ?^r[\tilde{T}]; (\Gamma_2 \backslash \tilde{z})$,$\Gamma_5 = (x : ?^r[\tilde{T}] + \Gamma_1) - (x : ?^r[\tilde{T}]; (\Gamma_2 \backslash \tilde{z}))$. Then, the fact that Γ_5 is unlimited follows from the equations below:

$$\begin{aligned}\Gamma_5 &= (x : ?^r[\tilde{T}] + \Gamma_1) - (x : ?^r[\tilde{T}]; (\Gamma_2 \backslash \tilde{z})) \\ &= x : (?^r[\tilde{T}] - ?^r[\tilde{T}]) + (\Gamma_1 - \Gamma_2 \backslash \tilde{z}) \\ &= x : |^r[\tilde{T}] + \Gamma_3.\end{aligned}$$

Moreover,

$$\begin{aligned}\Gamma_4 + \Gamma_5 &= (x : ?^r[\tilde{T}]; (\Gamma_2 \backslash \tilde{z})) + (x : |^r[\tilde{T}] + \Gamma_3) \\ &= x : ?^r[\tilde{T}] + (\Gamma_2 \backslash \tilde{z} + \Gamma_3) \\ &= x : ?^r[\tilde{T}] + \Gamma_1 \\ &= \Gamma\end{aligned}$$

– Case (T-NEW)
In this case,

$P = (\nu x : p^m[\tilde{T}])\, P_1$
$\Gamma, x : p^m[\tilde{T}] \vdash P_1$

By induction hypothesis,

$\Gamma, x : p^m[\tilde{T}] \vdash P_1 \stackrel{\Gamma_2}{\Rightarrow} \Gamma_2, x : U$
$\Gamma_1 + (\Gamma_2, x : U) = \Gamma, x : p^m[\tilde{T}]$
Γ_2, U are unlimited

By (TCHECK-NEW), we have $\Gamma \vdash P \stackrel{\Gamma_1 \backslash x}{\Rightarrow} \Gamma_2$ and $\Gamma_1 \backslash x + \Gamma_2 = (\Gamma, x : p^m[\tilde{T}]) \backslash x = \Gamma$.

2: Suppose that $\Gamma \vdash P \stackrel{\Gamma_1}{\Rightarrow} \Gamma_2$, Γ_2 is unlimited, and that $\Gamma = \Gamma_1 + \Gamma_2$. By Lemma 7.7, $\Gamma_1 \vdash P$. By weakening (Lemma 4.3), we obtain $\Gamma \vdash P$.

□

8 Discussions

8.1 Extensions and Variants

The type system presented in this paper is simple but very restricted. In fact, mutex channels allowed in the type system are too restricted. For example, the present type system does not allow the following fragment of a program:

$$(\nu x : \updownarrow^1[?^1[\,], !^1[\,]])\,(x![y,z] \,|\, x?[u,v].\, u?[\,].\, v![\,])$$

because there is an ordering between u and v, which violates the rule (T-IN). Because of such restrictions, general concurrent objects cannot be realized in the deadlock free fragment. One way to extend the type system is to attach time tags to channel type constructors, and to keep ordering between them. For example, in the above program, the type of x can be expressed as $\updownarrow^{1,t_1}[?^{1,t_2}[\,], !^{1,t_3}[\,]]$ where $t_2 \leq t_3$. A type environment can be represented as a pair consisting of a set of type bindings and poset on time tags. Along this line of the extension, we can also relax conditions for mutex channels. The extended system is described in [7]. Another (theoretically) interesting variant would be the one obtained by further restricting usage of replicated input channels. Recall, in Section 6, that a function type $\tau_1 \to \tau_2$ was encoded into $!^r[[\![\tau_1]\!], !^1[[\![\tau_2]\!]]]$. However, we cannot conclude that the invocation of a process listening to a channel of that type is not blocked, without information that the callee process does not use any channels of the type $p^\omega[\tilde{T}]$.[8] It is easy to restrict the present type system to obtain the precise encoding of function types. For example, we can consider restricted versions of replicated input channel types $p^{r'}[\tilde{T}]$ and serialized linear channel types $p^{1'}[\tilde{T}]$, and add the side conditions saying "body of the restricted replicated input prefix can only use channels of types $p^{r'}[\tilde{T}]$ or $p^{1'}[\tilde{T}]$," etc. It is easy to add such side conditions so that a receiver listening to a channel of the type $!^{r'}[[\![\tau_1]\!], !^{1'}[[\![\tau_2]\!]]]$ will exactly behave in a functional manner (i.e., eventually return a reply or fall into an infinite loop).

8.2 Type Reconstruction

Type reconstruction seems much harder than type check. In fact, in our present type system, some expression does not have most general typing (i.e., principal typing). For example, consider the expression $P = x?[\,].\, y![\,] \,|\, y?[\,].\, x![\,]$. Both of the type judgments:

$$x : \updownarrow^1[\,], y : \updownarrow^\omega[\,] \vdash P$$
$$x : \updownarrow^\omega[\,], y : \updownarrow^1[\,] \vdash P$$

are valid (the first typing judges that the source of deadlock is an unreliable channel y, while the second one judges that the source of deadlock is an unreliable channel y), but there is not a more general, valid typing. One solution to this problem would be to let programmers

[8] Note that our typed calculus still has advantages over untyped or other typed calculus because we need not look into the code precisely; we just need to know what types of channels are used in the code.

declare which channels should be treated as reliable channels (i.e., linear, replicated, and mutex channels) by the type system (instead of having them declare all types). Then, we think that we can reconstruct other type information by modifying Igarashi and Kobayashi's algorithm[6] to detect linear channels.

Even without any reconstruction algorithm, our typed calculus would also be useful as an intermediate implementation language for higher-level concurrent programming languages (note that type annotations can automatically be attached during translations of high-level structures into the calculus), because it keeps high-level structures useful for implementation and debugging by means of typing.

9 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 setting of name passing process calculi such as π-calculus. 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 presented in [2] ensures deadlock freedom, but the calculus is so restricted that it cannot express the mobility (i.e., it does not allow processes to communicate channels, hence the communication topology is completely static), and also that there can be only one sender and receiver for each channel.

Very recently, Yoshida[26] independently studied a type system for process calculi where types can be represented by graph structures. One of weak points of her type system would be, unlike ours, that there seems no reasonable type check algorithm. Although the purpose of their type system is different from ours, it might be interesting to study relationships between their type system and ours (one described here and the extension in [7]) to obtain more general type systems for process calculi.

Nielson and Nielson[14] proposed a technique to extract terms of process algebra from CML programs. The extracted terms help programmers understand behaviors of CML programs to some extent. However, while CML has channels as first-class values, their process algebra treats channels as (essentially) non-first-class values as in CCS, and as a result, multiple channels in CML programs must be mapped to the same channel (which they call a region) in the extracted process term. Therefore, we cannot use their technique to reason about deadlock, etc. To see why we cannot reason about deadlock, consider a 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 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 where the original program may fall into deadlock, but its approximation does not. On the other hand, consider a 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.

10 Conclusion

We have proposed a novel type system for process calculus to ensure partial deadlock freedom. The type system not only improves the reliability of concurrent programs, but also will potentially enable several optimizations. Future work includes extensions of the type system to ensure deadlock freedom of a larger part of concurrent programs, and applications of the type system to 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. The idea of this work has been inspired during discussions with them.

References

[1] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. *Software Trends*, 1:61–80, 1993.

[2] Simon Gay and Rajagopal Nagarajan. A typed calculus of synchronous processes. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1995.

[3] Kohei Honda. Types for dydadic interaction. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523, 1993.

[4] Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 348–360, 1994.

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

[6] 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, 1996. to appear.

[7] Naoki Kobayashi. A Partially Deadlock-free Typed Process Calculus (II) – An Advanced System and Deadlock-free Concurrent Objects –. forthcoming technical report from Department of Information Science, University of Tokyo, 1996.

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

[9] 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.

[10] Naoki Kobayashi and Akinori Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer-Verlag, 1995.

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

[12] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i, ii. *Information and Computation*, 100:1–40, September 1992.

[14] 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.

[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 Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 1995. To appear. A summary was presented at LICS '93.

[18] 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.

[19] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1995.

[20] 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.

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

[22] Vijay Saraswat and Patrick Lincoln. Higher-order, linear, concurrent constraint programming, July 1992. draft.

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

[24] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, 1995.

[25] Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, 1987. 282pages.

[26] Nobuko Yoshida. Graph types for monadic processes. presented in UK/Japan Workshop, Manchester, April 1996.