A New Type System for Deadlock-Free Processes

Naoki Kobayashi

Graduate School of Information Sciences, Tohoku University

Abstract. We extend a previous type system for the π-calculus that guarantees deadlock-freedom. The previous type systems for deadlock-freedom either lacked a reasonable type inference algorithm or were not strong enough to ensure deadlock-freedom of processes using recursion. Although the extension is fairly simple, the new type system admits type inference and is much more expressive than the previous type systems that admit type inference. In fact, we show that the simply-typed λ-calculus with recursion can be encoded into the deadlock-free fragment of our typed π-calculus. To enable analysis of realistic programs, we also present an extension of the type system to handle recursive data structures like lists. Both extensions have already been incorporated into the recent release of `TyPiCal`, a type-based analyzer for the π-calculus.

1 Introduction

Various type systems for the π-calculus have been proposed, some of which can guarantee that processes are deadlock-free in the sense that certain communications will eventually succeed unless the process diverges [3, 5, 6, 8, 10, 15]. (Some of them guarantee even a stronger property.) Earlier type systems for deadlock-freedom [5, 6, 14, 15] required explicit type annotations, so that they were not suitable for automatic analysis of deadlock-freedom. Kobayashi et al. [8, 10] later modified the type systems so that the resulting type systems have a type inference algorithm, and deadlock-freedom of processes can be automatically analyzed through type inference.

Based on the type system of [8], Kobayashi has implemented the first version of `TyPiCal` (ver. 1.0), a type-based analyzer for the π-calculus. Figure 1 shows a sample input and output of the deadlock analysis of `TyPiCal`. The first line in the input program runs two servers, one of which waits for a request on channel `server1` and sends 1 back to the reply channel `r`, and the other of which waits for a request on channel `server2` and may or may not send a reply, depending on the value of `b`. (Here, ?, !, and | represent an input action, an output action, and parallel composition respectively. `O` represents an inaction.) The second line runs a client process, which creates a fresh communication channel `r1` for receiving a reply, sends a request on `server1`, and waits for a reply. The client process on the third line behaves similarly, except that it sends a request on `server2`. Given that program, `TyPiCal`'s deadlock analyzer automatically finds input and output operations that are guaranteed to succeed if they are ever executed and if the whole process does not diverge, and mark them with `??` and `!!`. The output shown in the figure indicates that the first client can eventually receive a reply (note that `r1?x` has been replaced by `r1??x`), while the second client may not be able to receive a reply (`r2?x` remains the same).

Input program:

```
  *(server1?r.r!1) | *(server2?r.if b then r!1 else O)  /* Servers */
| new r1 in server1!r1.r1?x  /* A client for the first server */
| new r2 in server2!r2.r2?x  /* A client for the second server */
```

Output:

```
  *(server1?r.r!!1) | *(server2?r.if b then r!!1 else O)
| new r1 in server1!!r1.r1??x | new r2 in server2!!r2.r2?x
```

Fig. 1. A sample input and output of the deadlock analysis of `TyPiCal`

To enable type inference, however, we have traded the strength of the type system [8, 10]. In particular, the previous type systems for deadlock-freedom equipped with type inference algorithms cannot well handle recursive processes. For example, consider the following function server, which computes the factorial:

```
*fact?(n,r).if n=0 then r!1
            else new r1 in (fact!(n-1,r1) | r1?x.r!(x*n))
```

The server is deadlock-free in the sense that given a request, it will eventually returns a result unless the process diverges (actually, the process does not diverge, but the termination analysis is out of scope of this paper), but the previous type systems fail to conclude that. Even the simply-typed λ-calculus (without recursion) could not be encoded into the deadlock-free fragment of the previous type systems [8, 10]. On the other hand, an earlier type system of Kobayashi [5] could handle the above recursive process, but it was so complicated that a type inference algorithm could not be developed.

In this paper, we introduce a simple extension of the type system for deadlock-freedom [8, 10], which allows us to handle recursive processes like above, while keeping the existence of a type inference algorithm. Unlike the previous type systems which deal with pure polyadic π-calculus, we also extend the target language with data structures like pairs and lists. We have already incorporated those extensions into the recent version of `TyPiCal`.

The rest of this paper is structured as follows. Section 2 introduces our target language (with only pairs as data structures). Section 3 introduces our new type system for deadlock-freedom, and shows its soundness. To demonstrate the strength of our type system, Section 4 shows that the simply-typed λ-calculus with recursion can be encoded into the deadlock-free fragment of our typed calculus. Section 5 informally explains how to deal with list data structures.

2 Target Language

This section introduces the target language of our deadlock analysis, which is a subset of π-calculus [12] extended with booleans, pairs, and conditionals.

2.1 Syntax

Definition 21 (processes) *The set of processes, ranged over by P, is defined by:*

$$\begin{array}{rl} P ::= & \mathbf{0} \mid x!^t v.\, P \mid x?^t y.\, P \\ & \mid (P \,|\, Q) \mid *P \mid (\nu x)\, P \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \mid \mathbf{let}\ x = e\ \mathbf{in}\ P \\ e\ ::= & \mathit{true} \mid \mathit{false} \mid x \mid \langle e_1, e_2 \rangle \mid \mathit{proj}_1(e) \mid \mathit{proj}_2(e) \\ v\ ::= & \mathit{true} \mid \mathit{false} \mid x \mid \langle v_1, v_2 \rangle \end{array}$$

Here, x and y range over a countably infinite set **Var** *of variables. t ranges over* $\mathbf{Nat} \cup \{\infty\}$.

Notation 21 *The prefix $x?y$ binds variables y and (νx) binds x. As usual, we identify processes up to α-conversions (renaming of bound variables), and assume that α-conversions are implicitly applied so that bound variables are always different from each other and from free variables. We write $[x \mapsto v]P$ for the process obtained by replacing all the free occurrences of x in P with v. We often omit $\mathbf{0}$ and write $x!v$ and $x?y$ for $x!v.\,\mathbf{0}$ and $x?y.\,\mathbf{0}$ respectively.*

We assume that prefixes ($x!v$, $x?y$, (νx), and $*$) bind tighter than the parallel composition operator $|$, so that $x!y.\, P \,|\, Q$ means $(x!y.\, P) \,|\, Q$, not $x!y.\, (P \,|\, Q)$. We often write $x?(y, z).\, P$ for $x?p.\, \mathbf{let}\ y = \mathit{proj}_1(p)\ \mathbf{in}\ \mathbf{let}\ z = \mathit{proj}_2(p)\ \mathbf{in}\ P$ (where we assume p does not appear in P).

Process $\mathbf{0}$ does nothing. Process $x!^t v.\, P$ sends v on x, and then (after v is received by some process) the process behaves like P. The label t indicates whether the output operation is deadlock-free: If $t \neq \infty$, then the output is deadlock-free, i.e., if it is ever executed, v will eventually be received by some process or the whole process diverges. The exact value of t can be ignored at this moment; it will only be used in the type system. We call t a *capability annotation*. Note that programmers actually need not supply capability annotations; They are automatically inferred through type inference. We often omit t when it is unimportant. Process $x?^t y.\, P$ waits to receive a value v on x and then behaves like $[y \mapsto v]P$. The label t indicates whether the input operation is deadlock-free: If $t \neq \infty$, then the input is deadlock-free, i.e., if it is ever executed, the process will eventually be able to receive a message on x or the whole process diverges. $P \,|\, Q$ represents concurrent execution of P and Q. $*P$ represents infinitely many copies of the process P running in parallel, and $(\nu x)\, P$ denotes a process that creates a fresh communication channel x and then behaves like P. $\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ behaves like P if v is *true*, and behaves like Q if v is *false*. $\mathbf{let}\ x = e\ \mathbf{in}\ P$ evaluates e to some value v, binds x to it, and then behaves like P.

2.2 Operational Semantics

As usual, we define the operational semantics using a structural relation $P \preceq Q$, and a reduction relation $P \longrightarrow Q$. The former relation means that P can be restructured to Q by using the commutativity and associativity laws on $|$, etc. The latter relation means that P is reduced to Q by one communication on a channel. The formal definition of the relations are given in the full paper [**?**]. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

3 Type System

We introduce the new type system for deadlock-freedom in this section.

3.1 Overview

We first review the idea of previous type systems for deadlock-freedom [8, 10], identify the weakness of them, and then explain how to get rid of the weakness.

Ideas of previous type systems for deadlock-freedom The main idea of previous type systems for deadlock-freedom was to extend channel types with the following information:

- Channel-wise usage information, which describes how often and in which order each channel is used for input and output.
- Capability and obligation of each input/output action, which captures certain inter-channel dependency information.

We express channel-wise usage information by using a small, CCS-like process calculus, which has two primitive actions ? and !. For example, usage of x in the process $x?y \,|\, x!1 \,|\, x!2$ is expressed by $? \,|\, ! \,|\, !$, which means that x is used once for input and twice for output possibly in parallel. The usage of x in $x?y.\, x!y$ is expressed by ?.!, which means that x is first used for input, and then used for output. The usage conveys some information about whether each action succeeds or not. For example, x having usage $? \,|\, ! \,|\, !$ indicates that at least one of the two outputs fails to succeed. Similarly, x having usage ?.! (in the whole process) indicates that neither an input action nor an output action succeeds, since the input and output do not occur in parallel.

Channel-wise usage information alone is not sufficient for the analysis of deadlock. For example, it cannot distinguish between a deadlocked process $x?z.\, y!z \,|\, y?z.\, x!1$ and a non-deadlocked process $x?z.\, y!z \,|\, x!1.\, y?z.$. To control the dependency between communications on different channels, we have introduced the notion of *capabilities* and *obligations* [6, 8]. Let us explain why $x?z.\, y!z \,|\, y?z.\, x!1$ deadlocks in terms of *capabilities* (to successfully receive or send a message) and *obligations* (to wait for or to send a message). In order for the left sub-process $x?z.\, y!z$ to succeed in receiving a message on x, some process has to fulfill an *obligation* to send a message on x. The right sub-process, however, tries to exercise a *capability* to receive a message on y before fulfilling the obligation. In order for the right sub-process to be able to exercise a capability, the left process must fulfill an obligation to send a message on y, but the left process tries to exercise a capability to receive a message on x before fulfilling the obligation. Thus, the capability/obligation dependency is circular, so that no communication can succeed. To avoid such circular dependency, each action (? or !) in the channel-wise usage is associated with the *levels* of obligations and capabilities, which range over $\{0, 1, 2, \ldots\} \cup \{\infty\}$. The capability and obligation levels impose the following rules on the behavior of a process and its environment.

A. An obligation of level $n(\neq \infty)$ must be fulfilled by using only capabilities of level less than n. For example, suppose that x has usage $?^0_0$ and y has usage $!^1_1$, where the subscript of an action describes its capability level and the superscript describes its obligation level. Then, $x?z.\, y!z$ and $x?z \mid y!1$ are valid, but $y!1.\, x?z$ is invalid: the last process tries to exercise a capability of level 1 before fulfilling the obligation of lower level.

B. For an action of capability level $n(\neq \infty)$, there must exist a co-action of obligation level less than or equal to n (so as to guarantee that the capability can be eventually exercised).

Therefore, the obligation level describes a requirement for the process being concerned, while the capability level describes an assumption about the environment of the process being concerned. The two rules above ensure that there is no cyclic dependency between capabilities and obligations of finite levels; thus, deadlock-freedom is ensured for any action of a finite capability level.

Let us come back to the deadlocked process $x?z.\, y!z \mid y?z.\, x!1$. Suppose that the usages of x and y are $?^{o_{x1}}_{c_{x1}} \mid !^{o_{x2}}_{c_{x2}}$ and $?^{o_{y1}}_{c_{y1}} \mid !^{o_{y2}}_{c_{y2}}$, where c_{x1} and c_{y1} are finite. Rule **A** above implies that $c_{x1} < o_{y2}$ and $c_{y1} < o_{x2}$, while rule B implies that $o_{x2} \leq c_{x1}$, $o_{x1} \leq c_{x2}$, $o_{y2} \leq c_{y1}$, and $o_{y1} \leq c_{y2}$. So, we get $c_{x1} < o_{y2} \leq c_{y1} < o_{x2} \leq c_{x1}$, a contradiction.

Remark 1. A reader may wonder why a simpler approach of assigning a single level to each name does not work. For example, one may be tempted to assign levels l_x and l_y with $l_x < l_y$ to x and y to conclude that $x?z.\, y!z \mid y?z.\, x!1$ is wrong (since the lefthand process violates the order). This naive approach breaks down when one tries to deal with more complicated examples. For example, consider $x?z.\, y?z.\, (x!z. \mid y!z)$, where x and y are lock channels [5, 8]. The process is innocent (it only tries to lock x and y in this order, and then release both of them), but with the single level for each channel, we cannot assign a proper order between l_x and l_y, as the process performs a communication on x first, then on y, and again on x.

Weakness of previous type systems The main weakness of the previous type systems based on the idea above was that they cannot handle recursive processes well. Consider the following function server computing the factorial:

$$*\mathit{fact}?(n, r).\, \mathbf{if}\ n = 0\ \mathbf{then}\ r!1\ \mathbf{else}\ (\nu r')\, (\mathit{fact}!(n-1, r') \mid r'?m.\, r!(m \times n))$$

The second argument r of *fact* is assigned a type of the form $\mathbf{chan}(int, !^{t_o}_{t_c})$, which says that the channel is used for sending an integer, and the levels of the obligation and capability to do so are t_o and t_c respectively. Since r' is sent on *fact*, it is also assigned the type $\mathbf{chan}(int, !^{t_o}_{t_c})$. Then, because of rule **B**, however, the capability level of the input action on r' in $r'?m. \cdots$ must be greater than t_o. So, the sub-process $r'?m. r!(m \times n)$ violates rule **A** (if t_o is not ∞). The same problem arises even in handling a process simulating a term of the simply-typed λ-calculus (without recursion). One way to overcome the problem above is to use dependent types, so that the obligation level of the second argument r can depend on the value of the first argument n [6]. The resulting type system would, however, require heavy type annotations.

The idea of the extension To get rid of the weakness mentioned above, we weaken rule **A** as follows:

A′. An obligation of level n on a channel x must be fulfilled by using only capabilities of level less than or *equal to* n, and if the capability level is n, the capability must be on a channel which has been created more recently than x.

For example, in the factorial server above, the level of an obligation to return a value on r and that of a capability to receive a value on r' are the same, but since r' has been created more recently, $r'?m. r!(m \times n)$ conforms to rule **A′**. Rule **A′** is sufficient to prevent deadlock by avoiding circular dependency between different channels. Since information about which channels has been created more recently is dynamic, a static analysis is required to estimate the information. In this paper, we use a simple syntactic analysis, which concludes that, in the process $(\nu x)\, P$, x has been created more recently than any other free channel of P. Fortunately, that turns out to be sufficient for handling recursive processes like the factorial server and processes simulating λ-terms.

In the formal operational semantics, a channel x being created more recently than another channel y corresponds to the condition that the prefix (νx) is inside the scope of the prefix (νy). Note that our operational semantics disallows the usual structural rule $(\nu x)\,(\nu y)\, P \equiv (\nu y)\,(\nu x)\, P$. The condition in **A′** could be the other way around; we could require that the capability must be on a channel which has been created *less* recently than x. We, however, found the condition above more useful than this alternative requirement. That is because one of the common channel creation patterns is $(\nu x)\,(P \,|\, x?y. Q)$, where P performs some sub-computation and sends the result on x.

3.2 Usages

This subsection introduces the syntax and semantics of usages more formally. They are almost identical to those of the previous type system [8].

Definition 31 (usages) *The set* $\mathcal{U}$ *of* usages*, ranged over by* U*, is given by:*

$$U ::= \mathbf{0} \mid \alpha^{t_1}_{t_2}.U \mid (U_1 \mid U_2) \mid *U \mid \uparrow^t U \mid U_1 \mathbin{\&} U_2 \mid \rho \mid \mu\rho.U$$
$$\alpha ::= ? \mid !$$

Here, t *ranges over* $\mathbf{Nat} \cup \{\infty\}$ *(where* $\mathbf{Nat}$ *is the set of natural numbers).*

We often omit $\mathbf{0}$ and write $\alpha^{t_1}_{t_2}$ for $\alpha^{t_1}_{t_2}.\mathbf{0}$. We extend the usual binary relation $\leq$ on $\mathbf{Nat}$ to that on $\mathbf{Nat} \cup \{\infty\}$ by $\forall t \in \mathbf{Nat} \cup \{\infty\}.t \leq \infty$. We also extend $+$ by $\infty + t = t + \infty = \infty$. We write $\mathbf{min}(x_1, \ldots, x_n)$ for the least element of $\{x_1, \ldots, x_n\}$ (∞ if $n = 0$) with respect to $\leq$ and write $\mathbf{max}(x_1, \ldots, x_n)$ for the greatest element of $\{x_1, \ldots, x_n\}$ (0 if $n = 0$). We assume that $\mu\rho$ binds ρ. We write $[\rho \mapsto U_1]U_2$ for the usage obtained by replacing the free occurrences of ρ in U_2 with U_1. We write $FV(U)$ for the set of free usage variables. A usage is *closed* if $FV(U) = \emptyset$.

Usages	Interpretation
$\mathbf{0}$	Cannot be used at all
$?^{t_o}_{t_c}.U$	Used once for input, and then used according to U
$!^{t_o}_{t_c}.U$	Used once for output, and then used according to U
$U_1 \mid U_2$	Used according to U_1 and U_2, possibly in parallel
$*U$	Used according to U by infinitely many processes
$\uparrow^t U$	The same as U, except that input and output obligation levels are lifted to t.
$U_1 \mathbin{\&} U_2$	Used according to either U_1 or U_2
ρ	Usage variable (used in combination with recursive usages below)
$\mu\rho.U$	Recursively used according to $[\rho \mapsto \mu\rho.U]U$.

Table 1. Meaning of Usage Expressions

Intuitive meaning of usages is summarized in Table 1. If t_o is finite, a channel of usage $\alpha^{t_o}_{t_c}.U$ must be used for the action α, while if t_o is ∞, the action need not be performed. When t_c is finite, the action will eventually succeed if it is ever executed and the whole process does not diverge. If t_c is ∞, there is no such guarantee. Note that a channel of usage $\alpha^{t_o}_{t_c}.U$ must be used according to U only if it has been used for the action α and the action succeeds. For example, a channel of usage $?^{\infty}_{0}.!^{0}_{\infty}$ can be used for input (but need not be used), and if it has been used for input and the input has succeeded, it *must* be used for output. That is similar to the usage of a lock: a lock may be acquired (but need not be acquired), and after the lock has been acquired, the lock must be released. In fact, a lock can be expressed as a channel of such usage: see Example 1. Usage $\uparrow^t U$ lifts the obligation levels occurring in U (except for those guarded by ? or !) so that the input obligations and output obligations become greater than or equal to t. For example, $\uparrow^1(?^0_0.!^0_\infty)$ is the same as $?^1_0.!^0_\infty$.

We give a higher precedence to prefixes ($\alpha^{t_o}_{t_c}$ and $*$) than to $\mid$. We write $\overline{\alpha}$ for the co-action of α ($\overline{?} =!$ and $\overline{!} =?$).

Example 1. Linear channels [9] are given a usage of the form $?^{n_1}_{n_2} \,|\, !^{n_3}_{n_4}$. Affine channels, which can be used *at most once*, are given a usage $?^{\infty}_{\infty} \,|\, !^{\infty}_{\infty}$. A reference cell can be implemented as a channel holding the current value as a message. Then, the read operation is expressed as $x?y.\,(x!y \,|\, \cdots)$, while the write operation is expressed as $x?y.\,(x!v \,|\, \cdots)$. The usage of a reference cell is thus represented as $!^{0}_{\infty} \,|\, *?^{\infty}_{0}.!^{0}_{\infty}$. Similarly, a binary semaphore can be expressed as a channel holding at most one message. The semaphore can be acquired by receiving the message, and released by sending the message back to the channel. Thus, the usage of a semaphore is represented as $!^{0}_{\infty} \,|\, *?^{\infty}_{n}.!^{n}_{\infty}$. Here, the level n controls which locks should be acquired first when multiple locks need to be acquired.

Next, we define capability/obligation levels of a usage.

Definition 32 (capabilities) *The* input and output capability levels *of usage U, written $cap_?(U)$ and $cap_!(U)$, are defined by:*

$$
\begin{array}{ll}
cap_\alpha(\mathbf{0}) = cap_\alpha(\overline{\alpha}^{t_o}_{t_c}.U) = cap_\alpha(\rho) = \infty & cap_\alpha(\alpha^{t_o}_{t_c}.U) = t_c \\
cap_\alpha(*U) = cap_\alpha(\uparrow^t U) = cap_\alpha(\mu\rho.U) = cap_\alpha(U) & \\
cap_\alpha(U_1 \,|\, U_2) = cap_\alpha(U_1 \,\&\, U_2) = \mathbf{min}(cap_\alpha(U_1), cap_\alpha(U_2)) &
\end{array}
$$

Definition 33 (obligations) *The* input and output obligation levels *of a closed usage U, written $ob_?(U)$ and $ob_!(U)$, are defined by:*

$$
\begin{array}{ll}
ob_\alpha(\mathbf{0}) = ob_\alpha(\overline{\alpha}^{t_o}_{t_c}.U) = \infty & ob_\alpha(\rho) = 0 \\
ob_\alpha(\alpha^{t_o}_{t_c}.U) = t_o & ob_\alpha(U_1 \,|\, U_2) = \mathbf{min}(ob_\alpha(U_1), ob_\alpha(U_2)) \\
ob_\alpha(\uparrow^t U) = \mathbf{max}(t, ob_\alpha(U)) & ob_\alpha(U_1 \,\&\, U_2) = \mathbf{max}(ob_\alpha(U_1), ob_\alpha(U_2)) \\
ob_\alpha(*U) = ob_\alpha(\mu\rho.U) = ob_\alpha(U) &
\end{array}
$$

We write $ob(U)$ for $\mathbf{max}(ob_?(U), ob_!(U))$.

We next introduce the usage reduction relation $U \longrightarrow U'$. Intuitively, $U \longrightarrow U'$ means that if a channel of usage U has been used for a communication, then it should be used according to U' afterwards. For example, $!^{0}_{\infty} \,|\, ?^{\infty}_{0}.!^{0}_{\infty} \longrightarrow !^{0}_{\infty}$ holds. The formal definition of the relation is given in the full paper [**?**].

Relations and operations on usages As described in rule **B** in Subsection 3.1, if some action has a capability of level n, the obligation level of its co-action should be at most n. The relation $rel(U)$ defined below ensures that condition.

Definition 34 (reliability) *We write $con_\alpha(U)$ when $ob_{\overline{\alpha}}(U) \leq cap_\alpha(U)$. We write $con(U)$ when both $con_?(U)$ and $con_!(U)$ hold. A usage U is* reliable*, written $rel(U)$, if $con(U')$ holds for any U' such that $U \longrightarrow^* U'$.*

The subusage relation $U_1 \leq U_2$ defined below means that U_1 expresses more liberal usage of channels than U_2, so that a channel of usage U_1 may be used as that of usage U_2. The first and second conditions require that the subusage relation is closed under contexts and reduction. The third and fourth conditions allow capabilities to be weakened and obligations to be strengthened.

Definition 35 (subusage) *The* subusage relation $\leq$ *on closed usages is the largest binary relation on usages such that the following conditions hold whenever* $U_1 \leq U_2$.

1. $[\rho \mapsto U_1]U \leq [\rho \mapsto U_2]U$ *for any usage* U *such that* $FV(U) = \{\rho\}$.
2. *If* $U_2 \longrightarrow U_2'$, *then there exists* U_1' *such that* $U_1 \longrightarrow U_1'$ *and* $U_1' \leq U_2'$.
3. *For each* $\alpha \in \{?, !\}$, $cap_\alpha(U_1) \leq cap_\alpha(U_2)$ *holds.*
4. *For each* $\alpha \in \{?, !\}$, *if* $con_{\overline{\alpha}}(U_1)$, *then* $ob_\alpha(U_1) \geq ob_\alpha(U_2)$.

3.3 Types

Definition 36 (types) *The set of* types *is given by:*

$$\tau \ \textit{(types)} ::= \mathbf{bool} \mid \tau_1 \times \tau_2 \mid \mathbf{chan}(\tau, U)$$

Type **bool** is the type of booleans. The type $\tau_1 \times \tau_2$ describes pairs consisting of a value of type τ_1 and a value of type τ_2. The type $\mathbf{chan}(\tau, U)$ describes channels that should be used according to U for transmitting values of type τ.

We extend relations and operations on usages to those on types.

Definition 37 (subtyping) *A subtyping relation* $\leq$ *is the least reflexive relation closed under the following rule:*

$$\frac{U \leq U'}{\mathbf{chan}(\tau, U) \leq \mathbf{chan}(\tau, U')} \qquad \frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

Definition 38 *The* obligation level *of type* τ, *written* $ob(\tau)$, *is defined by:* $ob(\mathbf{bool}) = \infty$, $ob(\tau_1 \times \tau_2) = \mathbf{min}(ob(\tau_1), ob(\tau_2))$, *and* $ob(\mathbf{chan}(\tau, U)) = ob(U)$.

Definition 39 *Unary operations* $*$ *and* $\uparrow^t$ *on types is defined by:* $*\mathbf{bool} = \uparrow^t\mathbf{bool} = \mathbf{bool}$, $*(\tau_1 \times \tau_2) = (*\tau_1) \times (*\tau_2)$, $\uparrow^t(\tau_1 \times \tau_2) = (\uparrow^t\tau_1) \times (\uparrow^t\tau_2)$, $*(\mathbf{chan}(\tau, U)) = \mathbf{chan}(\tau, *U)$, *and* $\uparrow^t(\mathbf{chan}(\tau, U)) = \mathbf{chan}(\tau, \uparrow^t U)$,

Definition 310 *A (partial) binary operation* $\mid$ *on types is defined by:* $\mathbf{bool} \,|\, \mathbf{bool} = \mathbf{bool}$, $(\tau_{11} \times \tau_{12}) \,|\, (\tau_{21} \times \tau_{22}) = (\tau_{11} \,|\, \tau_{21}) \times (\tau_{12} \,|\, \tau_{22})$, *and* $(\mathbf{chan}(\tau, U_1)) \,|\, (\mathbf{chan}(\tau, U_2)) = \mathbf{chan}(\tau, (U_1 \,|\, U_2))$. $\tau_1 \,|\, \tau_2$ *is undefined if it does not match any of the above rules.*

3.4 Type Environment

A type environment is a mapping from a finite set of variables to types. We use metavariables Γ and Δ for type environments. We write $\emptyset$ for the type environment whose domain is empty. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Γ' such that $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for all $y \in dom(\Gamma)$.

The operations and relations on types are pointwise extended to those on type environments below.

For expressions

$$\frac{}{x:\tau \vdash x:\tau} \ (\text{Tv-Var}) \qquad \frac{\Gamma \vdash e:\tau_1 \times \tau_2 \quad i \in \{1,2\} \quad ob(\tau_{3-i}) = \infty}{\Gamma \vdash proj_i(e):\tau_i} \ (\text{Tv-Proj})$$

$$\frac{b \in \{true, false\}}{\emptyset \vdash b:\mathbf{bool}} \ (\text{Tv-Bool})$$

$$\frac{\Gamma_1 \vdash e_1:\tau_1 \quad \Gamma_2 \vdash e_2:\tau_2}{\Gamma_1 \mid \Gamma_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \ (\text{Tv-Pair}) \qquad \frac{\Gamma \vdash e:\tau \quad \Gamma' \leq \Gamma}{\Gamma' \vdash e:\tau} \ (\text{Tv-Weak})$$

For Processes

$$\frac{\Gamma, x:\mathbf{chan}(\tau, U) \vdash_{\prec \cup \{(x,y) \mid y \in FV(P) \setminus \{x\}\}} P \quad rel(U)}{\Gamma \vdash_{\prec} (\nu x)\, P} \ (\text{T-New})$$

$$\frac{}{\emptyset \vdash_{\prec} \mathbf{0}} \ (\text{T-Zero}) \qquad \frac{\Gamma' \vdash_{\prec} P \quad \Gamma \leq \Gamma'}{\Gamma \vdash_{\prec} P} \ (\text{T-Weak})$$

$$\frac{\Gamma_1 \vdash_{\prec} P_1 \quad \Gamma_2 \vdash_{\prec} P_2}{\Gamma_1 \mid \Gamma_2 \vdash_{\prec} P_1 \mid P_2} \ (\text{T-Par}) \qquad \frac{\Gamma \vdash_{\prec} P}{*\Gamma \vdash_{\prec} *P} \ (\text{T-Rep})$$

$$\frac{\Gamma_1 \vdash_{\prec} P \quad \Gamma_2 \vdash v:\tau}{x:\mathbf{chan}(\tau, !^{0}_{t_c});_{\prec} (\Gamma_1 \mid \Gamma_2) \vdash_{\prec} x!^{t_c} v.\, P} \ (\text{T-Out}) \qquad \frac{\Gamma, y:\tau \vdash_{\prec} P}{x:\mathbf{chan}(\tau, ?^{0}_{t_c});_{\prec} \Gamma \vdash_{\prec} x?^{t_c} y.\, P} \ (\text{T-In})$$

$$\frac{\Gamma_1 \vdash e:\tau \quad \Gamma_2, x:\tau \vdash_{\prec} P}{\Gamma_1 \mid \Gamma_2 \vdash_{\prec} \mathbf{let}\ x = e\ \mathbf{in}\ P} \ (\text{T-Let}) \qquad \frac{\Gamma_1 \vdash v:\mathbf{bool} \quad \Gamma_2 \vdash_{\prec} P \quad \Gamma_2 \vdash_{\prec} Q}{\Gamma_1 \mid \Gamma_2 \vdash_{\prec} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q} \ (\text{T-If})$$

Fig. 2. Typing Rules

Definition 311 *A binary relation $\leq$ on type environments is defined by: $\Gamma_1 \leq \Gamma_2$ if and only if (i) $dom(\Gamma_1) \supseteq dom(\Gamma_2)$, (ii) $\Gamma_1(x) \leq \Gamma_2(x)$ for each $x \in dom(\Gamma_2)$, and (iii) $ob(\Gamma_1(x)) = \infty$ for each $x \in dom(\Gamma_1) \setminus dom(\Gamma_2)$.*

Definition 312 *The operations $\mid$ and $*$ on type environments are defined by:*

$$(\Gamma_1 \mid \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \mid \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \setminus dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \setminus dom(\Gamma_1) \end{cases}$$
$$(*\Gamma)(x) = *(\Gamma(x))$$

3.5 Typing Rules

We have two kinds of judgments: $\Gamma \vdash e : \tau$ for expressions, and $\Gamma \vdash_{\prec} P$ for processes. The latter means that P uses free variables as specified by Γ. $\prec$ is a partial order that statically estimates the order between the times when channels are created. $x \prec y$ means that x must have been created more recently than y. Because of rule $\mathbf{A}'$, $x : \mathbf{chan}(\mathbf{bool}, ?^0_1), y : \mathbf{chan}(\mathbf{bool}, !^1_\infty) \vdash_{\{(x,y)\}} x?z.\, y!z$ and $x : \mathbf{chan}(\mathbf{bool}, ?^0_0), y : \mathbf{chan}(\mathbf{bool}, !^1_\infty) \vdash_\emptyset x?z.\, y!z$ are valid judgments, while $x : \mathbf{chan}(\mathbf{bool}, ?^0_1), y : \mathbf{chan}(\mathbf{bool}, !^1_\infty) \vdash_\emptyset x?z.\, y!z$ is invalid.

We assume that α-conversion is implicitly applied so that the variables in Γ and $\prec$ are always different from the bound variables in P. The typing rules for deriving valid type judgments are given in Figure 2.

We explain some key rules. In T-New, $\prec$ is extended with the assumption that x has been created more recently than any other free channels in P.

In T-Out and T-In, we use the operation $x : \mathbf{chan}(\tau, \alpha^{t_o}_{t_c});_\prec \Gamma$ on type environments. It represents the type environment Δ defined by:

$$
\begin{aligned}
&dom(\Delta) = \{x\} \cup dom(\Gamma)\\
&\Delta(x) = \begin{cases} \mathbf{chan}(\tau, \alpha^{t_o}_{t_c}.U) & \text{if } \Gamma(x) = \mathbf{chan}(\tau, U)\\ \mathbf{chan}(\tau, \alpha^{t_o}_{t_c}) & \text{if } x \notin dom(\Gamma)\end{cases}\\
&\Delta(y) = \begin{cases} \uparrow^{t_c} \Gamma(y) & \text{if } y \neq x \wedge x \prec y\\ \uparrow^{t_c+1} \Gamma(y) & \text{if } y \neq x \wedge x \not\prec y\end{cases}
\end{aligned}
$$

For example, $x : \mathbf{chan}(\tau, ?^0_2);_{\{(x,y)\}}(x : \mathbf{chan}(\tau, !^0_0), y : \mathbf{chan}(\tau_1, !^0_0), z : \mathbf{chan}(\tau_2, !^0_0))$ is $x : \mathbf{chan}(\tau, ?^0_2.!^0_0), y : \mathbf{chan}(\tau_1, !^2_0), z : \mathbf{chan}(\tau_2, !^3_0))$.

Intuitively, the environment $x : \mathbf{chan}(\tau, \alpha^{t_o}_{t_c});_\prec \Gamma$ means that x may be first used for the action α, and then communications can be performed according to Γ. Since the capability of level t_c is exercised before fulfilling obligations in Γ, the level of each obligation in Γ are lifted either to t_c or $t_c + 1$, depending on $\prec$.

In rule T-In, the premise means that P performs communications according to Γ. Since $x?^{t_c}y.\, P$ tries to exercise a capability of level t_c to receive a value on x, the process is well-typed under $x : \mathbf{chan}(\tau, ?^0_{t_c});_\prec \Gamma$.

Example 2. Let us consider the following process P:

$$*f?r.\, (\mathbf{if}\ b\ \mathbf{then}\ r!\mathit{true}\ \mathbf{else}\ (\nu r')\,(f!r' \mid r'?x.\, r!x)).$$

It is typed as follows.

$$
\dfrac{\dfrac{\dfrac{\Gamma \vdash_\emptyset r!\mathit{true} \quad \Gamma \vdash_\emptyset (\nu r') \cdots}{\Gamma \vdash_\emptyset \mathbf{if}\ b\ \mathbf{then}\ r!\mathit{true}\ \mathbf{else}\ \cdots}}{f : \mathbf{chan}(\mathbf{chan}(\mathbf{bool}, !^1_\infty), ?^0_\infty.!^\infty_0), b : \mathbf{bool} \vdash_\emptyset f?r.\, \cdots}}{f : \mathbf{chan}(\mathbf{chan}(\mathbf{bool}, !^1_\infty), *?^0_\infty.!^\infty_0), b : \mathbf{bool} \vdash_\emptyset P}
$$

Here, Γ is $f : \mathbf{chan}(\mathbf{chan}(\mathbf{bool}, !^1_\infty), !^\infty_0), b : \mathbf{bool}, r : \mathbf{chan}(\mathbf{bool}, !^1_\infty)$, and $\Gamma \vdash_\emptyset (\nu r') \cdots$ is derived by:

$$
\dfrac{\dfrac{\Gamma_1 \vdash_{\{(r',r)\}} f!r' \quad r : \mathbf{chan}(\mathbf{bool}, !^1_\infty), r' : \mathbf{chan}(\mathbf{bool}, ?^0_1) \vdash_{\{(r',r)\}} r'?x.\, r!x}{\Gamma, r' : \mathbf{chan}(\mathbf{bool}, !^1_\infty \mid ?^0_1) \vdash_{\{(r',r)\}} f!r' \mid r'?x.\, r!x}}{\Gamma \vdash_\emptyset (\nu r') \cdots}
$$

Here, $\Gamma_1 = f:\mathbf{chan}(\mathbf{chan}(\mathbf{bool}, !^1_\infty), !^\infty_0), r':\mathbf{chan}(\mathbf{bool}, !^1_\infty)$. Note that if $r' \prec r$ did not hold, we could only obtain $r:\mathbf{chan}(\mathbf{bool}, !^2_\infty),\ r':\mathbf{chan}(\mathbf{bool}, ?^0_1) \vdash_\emptyset r'?x.\, r!x$, so that $f:\mathbf{chan}(\mathbf{chan}(\mathbf{bool}, !^1_\infty), *?^0_\infty.!^\infty_0),\ b:\mathbf{bool} \vdash_\emptyset P$ were not derivable.

3.6 Type Soundness

The following theorems imply that if a process is well-typed in our type system, an input or output process that is annotated with a finite capability level is deadlock-free, in the sense that if the process is ready (i.e., it appears at the top-level, without being guarded by any other input or output prefix), the whole process can be reduced further.

We write $\Gamma \longrightarrow \Gamma'$ when $\Gamma = \Gamma_1, x:\mathbf{chan}(\tau, U)$ and $\Gamma' = \Gamma_1, x:\mathbf{chan}(\tau, U')$ with $U \longrightarrow U'$ for some Γ_1, x, τ, U, and U'.

Theorem 1 (type preservation). *If $\Gamma \vdash_\prec P$ and $P \longrightarrow Q$, then $\Gamma' \vdash_\prec Q$ for some Γ' such that $\Gamma' = \Gamma$ or $\Gamma \longrightarrow \Gamma'$.*

Theorem 2. *If $\emptyset \vdash_\prec P$ and either $P \preceq (\nu\widetilde{x})\,(x!^n v.\, Q_1 \mid Q_2)$ or $P \preceq (\nu\widetilde{x})\,(x?^n y.\, Q_1 \mid Q_2)$ with $n \in \mathbf{Nat}$, then $P \longrightarrow R$ for some R.*

Corollary 1. *Suppose $\emptyset \vdash_\prec P$. If $P \longrightarrow^* Q$, and either $Q \preceq (\nu\widetilde{x})\,(x!^n v.\, Q_1 \mid Q_2)$ or $Q \preceq (\nu\widetilde{x})\,(x?^n y.\, Q_1 \mid Q_2)$ with $n \in \mathbf{Nat}$, then $Q \longrightarrow R$ for some R.*

3.7 Type Inference

Given a closed process P (without any capability annotations on input and output processes), there is a complete algorithm to decide whether there exists P' such that $\emptyset \vdash_\emptyset P'$ holds and P and P' coincide except for capability annotations. Moreover, such an algorithm tries to infer the least capability for each input/output process. Since the algorithm is almost the same as that of the previous type system [8], we do not re-describe the algorithm here; The algorithm first extract constraints on types, reduce them step by step to obtain constraints of the form $rel(U)$, and then solve $rel(U)$ by reduction to Petri net reachability problems [8]. The only extra work compared with the previous one is to expand the relation $\prec$ when the algorithm encounters the ν-prefix. We have already implemented the algorithm in `TyPiCal` [4].

4 Encoding of λ-calculus

To demonstrate the power of the new type system, we show that the call-by-value simply-typed λ-calculus with recursion can be encoded into the deadlock-free fragment. Concurrent objects can also be encoded as in our previous paper [5].

Definition 41 *The sets of types and terms of $\lambda^{\rightarrow,\mathit{fix}}$ are given by the following syntax:*

$$\begin{aligned} &\theta \ \textit{(types)} \quad ::= \mathbf{bool} \mid \theta_1 \rightarrow \theta_2 \\ &M \ \textit{(terms)} ::= x \mid \boldsymbol{\mathit{fix}}(f, x, M) \mid M_1 M_2 \end{aligned}$$

Here, $\mathbf{fix}(f, x, M)$ represents a recursive function f defined by $f(x) \stackrel{\triangle}{=} M$. If f does not appear in M, it is the same as the usual λ-abstraction $\lambda x.M$.

Typing rules are given as follows.

$$\frac{}{\mathcal{T}, x : \theta \vdash x : \theta} \quad \text{(TL-VAR)}$$

$$\frac{\mathcal{T}, f : \theta_1 \to \theta_2, x : \theta_1 \vdash M : \theta_2}{\mathcal{T} \vdash \mathbf{fix}(f, x, M) : \theta_1 \to \theta_2} \quad \text{(TL-FIX)}$$

$$\frac{\mathcal{T} \vdash M_1 : \theta_1 \to \theta_2 \qquad \mathcal{T} \vdash M_2 : \theta_1}{\mathcal{T} \vdash M_1 M_2 : \theta_2} \quad \text{(TL-APP)}$$

We encode terms, types, and type environments into our typed π-calculus as follows, in a standard manner [5, 11, 13].

$$\begin{aligned}
[\![x]\!]^r &= r!x \\
[\![\mathbf{fix}(f, x, M)]\!]^r &= (\nu y)\,(r!y \mid *y?(x, r').\, [\![M]\!]^{r'}) \\
[\![M_1 M_2]\!]^r &= (\nu r_1)\,(\nu r_2)\,([\![M_1]\!]^{r_1} \mid [\![M_2]\!]^{r_2} \mid r_1?f.\, r_2?x.\, f!(x, r))
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{bool}]\!] &= \mathbf{bool} \\
[\![\theta_1 \to \theta_2]\!] &= \mathbf{chan}([\![\theta_1]\!] \times \mathbf{chan}([\![\theta_2]\!], !^1_\infty), *!^\infty_0) \\
[\![x_1 : \theta_1, \ldots, x_n : \theta_n]\!] &= x_1 : [\![\theta_1]\!], \ldots, x_n : [\![\theta_n]\!]
\end{aligned}$$

Intuitively, a term M is encoded into $[\![M]\!]^r$ which evaluates M and sends the result on channel r. The usage $*!^\infty_0$ in the encoding of function types means that a function can be invoked an arbitrary number of times, and the usage $!^1_\infty$ means that the function will eventually returns a result (or diverge).

It is easy to check that the typing is preserved by encoding.

Lemma 1. *If $\mathcal{T} \vdash M : \theta$, then $[\![\mathcal{T}]\!], r : \mathbf{chan}([\![\theta]\!], !^1_\infty) \vdash_\emptyset [\![M]\!]^r$.*

The following is an immediate corollary of the above lemma, which means that a process that simulates functional computation does not get deadlocked before returning a result.

Corollary 2. *If $\emptyset \vdash M : \theta$ and $[\![M]\!]^r \longrightarrow^* P$, then $P \longrightarrow Q$ for some Q or $P \preceq (\nu \widetilde{x})\,(r!v.\, Q_1 \mid Q_2)$ for some v, Q_1, Q_2.*

Proof. Suppose $\emptyset \vdash M : \theta$ and $[\![M]\!]^r \longrightarrow^* P$. By Lemma 1, $r : \mathbf{chan}([\![\theta]\!], !^1_\infty) \vdash_\emptyset [\![M]\!]^r$. By Theorem 1, we have $r : \mathbf{chan}([\![\theta]\!], !^1_\infty) \vdash_\emptyset P$. Let $R = (\nu r)\,(P \mid r?^1 x.\, \mathbf{0})$. Then, $\emptyset \vdash_\emptyset R$. By Theorem 2, we have $R \longrightarrow R'$, which implies either $P \longrightarrow Q$ or $P \preceq (\nu \widetilde{x})\,(r!v.\, Q_1 \mid Q_2)$.

5 Extension for Recursive Data Structures

The language discussed so far is the π-calculus extended with pairs. We briefly discuss a subtle point that arises when dealing with recursive data structures, using the list data structure as an example.

Let us consider the following process, which waits to receive a list l of channels, and sends *true* to all the channels in the list.

$$\begin{array}{l} *broadcast?l.\,\mathbf{if}\ \ null(l)\ \mathbf{then\ 0\ else} \\ \qquad\qquad (\mathbf{let}\ x = \mathbf{hd}(l)\ \mathbf{in}\ (x!true \mid broadcast!\mathbf{tl}(l))) \end{array}$$

Here, $\mathbf{hd}(l)$ is the first element of the list l, and $\mathbf{tl}(l)$ is the rest.

A naive way to handle lists is to introduce list types of the form $\mathbf{list}(\tau)$, which describes lists whose elements are of type τ, and the following typing rules:

$$\frac{\Gamma \vdash e : \mathbf{list}(\tau)}{\Gamma \vdash \mathbf{hd}(e) : \tau} \qquad\qquad \frac{\Gamma \vdash e : \mathbf{list}(\tau)}{\Gamma \vdash \mathbf{tl}(e) : \mathbf{list}(\tau)}$$

However, we have to add the condition that $ob(\tau) = \infty$ in both rules (just like we had to impose the condition $ob(\tau_{3-i}) = \infty$ in the rule for projections), since $\mathbf{hd}(e)$ throws away the elements other than the head, and $\mathbf{tl}(e)$ throws away the head. Thus, we can only assign $\mathbf{list}(\mathbf{chan}(\mathbf{bool}, !^{\infty}_{t}))$ to l in the above example, failing to infer that the server eventually sends messages to all the elements in the list.

To overcome the problem above, we represent list types as $\mathbf{list}(\tau_1, \tau_2)$, where τ_1 is the type of the first element, and τ_2 is the type of the rest of the elements, and use the following types:

$$\frac{\Gamma \vdash e : \mathbf{list}(\tau_1, \tau_2) \qquad ob(\tau_2) = \infty}{\Gamma \vdash \mathbf{hd}(e) : \tau_1} \qquad\qquad \frac{\Gamma \vdash e : \mathbf{list}(\tau_1, \tau_2) \qquad ob(\tau_1) = \infty}{\Gamma \vdash \mathbf{tl}(e) : \mathbf{list}(\tau_2, \tau_2)}$$

With these rules, we can assign $\mathbf{list}(\mathbf{chan}(\mathbf{bool}, !^{1}_{\infty}), \mathbf{chan}(\mathbf{bool}, !^{1}_{\infty}))$ to l in the example above, so that we can infer that the server eventually sends messages to all the elements in the list.

The replacement of $\mathbf{list}(\tau)$ with $\mathbf{list}(\tau_1, \tau_2)$ corresponds to the unfolding of the recursive type $\mu\alpha.(1 + (\tau \times \alpha))$ to $1 + \tau \times \mu\alpha.(1 + (\tau \times \alpha))$. As in the case of lists above, unfolding of recursive types in general seems to be useful to make our type system for deadlock-freedom more robust.

6 Related Work

As already mentioned in Section 1, earlier type systems that can guarantee deadlock-freedom [5, 14, 15] required explicit type annotations, having no reasonable type inference algorithm. We have later modified the type systems to make type inference tractable [8, 10], with the sacrifice of some expressive power.

The type system proposed in this paper can be considered a reunion of the earlier type systems [5, 14] and recent ones [8, 10].

Some type systems [6, 8] can guarantee a stronger property that certain communications will eventually succeed no matter whether the process diverges. There are also type systems that guarantee the termination of processes [1, 16]. Unfortunately, the idea proposed in the present paper does not work for guaranteeing those stronger properties.

There are some studies of abstract interpretation for the π-calculus [2]. To the best of our knowledge, deadlock-freedom analysis has not been studied in that context. Our type-based analysis relies on a syntactic analysis of the order in which channels are created. Abstract interpretation [2] might be useful for obtaining more precise information about the order of channel creation.

7 Conclusion

We have proposed a new type system for deadlock-freedom of π-calculus processes. The new type system admits type inference, while it is strictly more expressive than the previous type systems that admit type inference. We have also extended the type system to handle data structures like pairs and lists.

References

1. Y. Deng and D. Sangiorgi. Ensuring termination by typability. In *Proceedings of IFIP TCS 2004*, pages 619–632, 2004.
2. J. Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63(1), 2005.
3. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. of POPL*, pages 81–92, 2002.
4. N. Kobayashi. TyPiCal: A type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.
5. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Prog. Lang. Syst.*, 20(2):436–482, 1998.
6. N. Kobayashi. A type system for lock-free processes. *Info. Comput.*, 177:122–159, 2002.
7. N. Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 20th Anniversary Colloquium*, volume 2757 of *LNCS*, pages 439–453. Springer-Verlag, 2003.
8. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
9. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Prog. Lang. Syst.*, 21(5):914–947, 1999.
10. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. Technical Report TR00-01, Dept. Info. Sci., Univ. of Tokyo, January 2000. A summary has appeared in Proceedings of CONCUR 2000, Springer LNCS1877, pp.489-503, 2000.
11. R. Milner. Function as processes. In *Automata, Language and Programming*, volume 443 of *LNCS*, pages 167–180. Springer-Verlag, 1990.

12. R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.
13. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proc. of LICS*, pages 376–385, 1993.
14. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, volume 16(3) of *ENTCS*, pages 55–77, 1998.
15. N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, volume 1180 of *LNCS*, pages 371–387. Springer-Verlag, 1996.
16. N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi-calculus. *Info. Comput.*, 191(2):145–202, 2004.

A Reduction semantics of processes

Definition A1 *The evaluation relation $e \Downarrow v$ is the least relation closed under the following rules:*

$$x \Downarrow x$$

$$\frac{b \in \{\mathit{true}, \mathit{false}\}}{b \Downarrow b}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\langle e_1, e_2 \rangle \Downarrow \langle v_1, v_2 \rangle}$$

$$\frac{e \Downarrow \langle v_1, v_2 \rangle \qquad i \in \{1, 2\}}{\mathit{proj}_i(e) \Downarrow v_i}$$

Definition A2 *The* structural preorder $\preceq$ *is the least reflexive and transitive relation closed under the following rules ($P \equiv Q$ denotes $(P \preceq Q) \wedge (Q \preceq P)$):*

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

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

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

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

$$\textbf{if } \mathit{true} \textbf{ then } P \textbf{ else } Q \preceq P \qquad \text{(S-IfT)}$$

$$\textbf{if } \mathit{false} \textbf{ then } P \textbf{ else } Q \preceq Q \qquad \text{(S-IfF)}$$

$$\frac{e \Downarrow v}{\textbf{let } x = e \textbf{ in } P \preceq [x \mapsto v]P} \qquad \text{(S-Let)}$$

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

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

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

Definition A3 *The reduction relation* $\longrightarrow$ *is the least relation closed under the following rules:*

$$x!v.\,P \,|\, x?y.\,Q \longrightarrow P \,|\, [y \mapsto v]Q \qquad \text{(R-Com)}$$

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

$$\frac{P \longrightarrow Q}{(\nu x)\, P \longrightarrow (\nu x)\, Q} \qquad \text{(R-New)}$$

$$\frac{P \preceq P' \qquad P' \longrightarrow Q' \qquad Q' \preceq Q}{P \longrightarrow Q} \qquad \text{(R-SP)}$$

B Usage Reduction Semantics

Definition B1 $\preceq$ *is the least reflexive and transitive relation on usages satisfying the following rules:*

$$U_1 \,|\, U_2 \preceq U_2 \,|\, U_1$$

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

$$\frac{U_1 \preceq U_1' \qquad U_2 \preceq U_2'}{U_1 \,|\, U_2 \preceq U_1' \,|\, U_2'}$$

$$*U \preceq *U \,|\, U$$

$$\uparrow^t \alpha_{t_2}^{t_1}.U \preceq \alpha_{t_2}^{\mathbf{max}(t_1,t)}.U$$

$$\uparrow^t (U_1 \,|\, U_2) \preceq (\uparrow^t U_1) \,|\, (\uparrow^t U_2)$$

$$U_1 \,\&\, U_2 \preceq U_i \;\; (i \in \{1,2\})$$

$$\mu\rho.U \preceq [\rho \mapsto \mu\rho.U]U$$

$$\frac{U \preceq U'}{\uparrow^t U \preceq \uparrow^t U'}$$

Definition B2 (usage reduction) *The binary relation* $\longrightarrow$ *on usages is the least relation closed under the following rules:*

$$?^{t_o}_{t_c}.U_1 \,|\, !^{t_o'}_{t_c'}.U_2 \longrightarrow U_1 \,|\, U_2$$

$$\frac{U_1 \longrightarrow U_1'}{U_1 \,|\, U_2 \longrightarrow U_1' \,|\, U_2}$$

$$\frac{U_1 \preceq U_1' \qquad U_1' \longrightarrow U_2' \qquad U_2' \preceq U_2}{U_1 \longrightarrow U_2}$$

C Proofs of Main Lemmas and Theorems

C.1 Proof of Theorem 1

The proof of the type preservation theorem is almost the same as that for the previous type systems [6, 8].

We first note that the properties of the subusage relation in the previous work [8] still hold for the subusage relation in the present paper, since they are essentially the same except that some usage constructor ($\uparrow$) has been omitted in the present paper. In particular, the following law on the subusage relation plays an important role.

Lemma 2. $\alpha^0_n.U_1 \,|\, \uparrow^{t+1}U_2 \leq \alpha^0_n.(U_1 \,|\, U_2)$.

We prepare the substitution lemma. A little care is required for the condition on $\prec$.

Lemma 3 (substitution). *If* $\Gamma_1, x : \tau \vdash_\prec P$ *and* $\Gamma_2 \vdash v : \tau$ *with* $(\{x\} \times \boldsymbol{Vars}) \cap \prec = (\boldsymbol{Vars} \times \{x\}) \cap \prec = \emptyset$, *then* $\Gamma_1 \,|\, \Gamma_2 \vdash_\prec P$.

Proof. The proof proceeds by induction on derivation of $\Gamma_1, x : \tau \vdash_\prec P$, with case analysis on the last rule used. We do not describe the detail since the proof is boring and almost identical to the proof of the corresponding theorem. We only mention the case for T-In, where the condition on $\prec$ is crucial. Suppose $\Gamma_1, x : \tau \vdash_\prec y?z.\,P_1$. There are two cases to consider: the case $x = y$ and the case $x \neq y$. We discuss only the first case, since the second case is simpler.

- Case $x = y$: In this case, we have:

  $$\Gamma_3, z : \tau_1 \vdash_\prec P_1$$
  $$\Gamma_1, x : \tau \leq x : \mathbf{chan}(\tau_1, ?^0_t);_\prec \Gamma_3$$

  We can assume without loss of generality that $\Gamma_3 = \Gamma_3', v : \mathbf{chan}(\tau_1, U_v), x : \mathbf{chan}(\tau_1, U_x)$ for some U_x, U_v, and Γ_3'. So, we have

  $$\Gamma_1, x : \tau \leq x : \mathbf{chan}(\tau_1, ?^0_t);_\prec \Gamma_3 = \uparrow^{t+1}\Gamma_3', x : \mathbf{chan}(\tau_1, ?^0_t.U_x), v : \mathbf{chan}(\tau_1, \uparrow^{t+1}U_v).$$

  This implies

  $$\Gamma_1 \,|\, v : \tau \leq \uparrow^{t+1}\Gamma_3', v : \mathbf{chan}(\tau_1, \uparrow^{t+1}U_v \,|\, ?^0_t.U_x)$$

  Note that the equality comes from the condition that $x(= y)$ is not related by $\prec$. (This is the very place where we need the condition on $\prec$.)
  By the assumption $\Gamma_2 \vdash v : \tau$, it must be the case that v is a variable. So, by applying the induction hypothesis to $\Gamma_3', x : \mathbf{chan}(\tau_1, U_x), v : \mathbf{chan}(\tau_1, U_v), z : \tau_1 \vdash_\prec P_1$, we obtain: $(\Gamma_3', v : \mathbf{chan}(\tau_1, U_x \,|\, U_v)), z : \tau_1 \vdash_\prec [x \mapsto v]P_1$. From this, we obtain

  $$v : \mathbf{chan}(\tau_1, ?^0_t);_\prec(\Gamma_3' \,|\, v : \mathbf{chan}(\tau_1, U_x \,|\, U_v)) \vdash_\prec v?z.\,[x \mapsto v]P_1.$$

By applying T-Weak, we further obtain

$$\uparrow^{t+1}\Gamma_3', v:\mathbf{chan}(\tau_1, ?^0_t.(U_x \,|\, U_v) \vdash_\prec v?z.\,[x \mapsto v]P_1.$$

By using Lemma 2 and T-Weak, we get:

$$\uparrow^{t+1}\Gamma_3', v:\mathbf{chan}(\tau_1, \uparrow^{t+1}U_v \,|\, ?^0_t.U) \vdash_\prec v?z.\,[x \mapsto v]P_1.$$

Using the fact $\Gamma_1 \,|\, v:\tau \leq \uparrow^{t+1}\Gamma_3', v:\mathbf{chan}(\tau_1, \uparrow^{t+1}U_v \,|\, ?^0_t.U_x)$, we can further apply T-Weak to obtain

$$\Gamma_1 \,|\, v:\tau \vdash_\prec v?z.\,[x \mapsto v]P_1.$$

Since $\Gamma_2 \leq v:\tau$, we finally obtain

$$\Gamma_1 \,|\, \Gamma_2 \vdash_\prec [x \mapsto v](x?z.\,P_1).$$

– Case $x \neq y$: In this case, we have:

$$\begin{array}{l}\Gamma_3, x:\tau', z:\sigma \vdash_\prec P_1\\ \Gamma_1, x:\tau \leq y:\mathbf{chan}(\sigma, ?^0_n);_\prec(\Gamma_3, x:\tau')\end{array}$$

From the second condition, it must be the case that $\tau \leq \uparrow^{n+1}\tau'$. With $\tau \leq \uparrow^{n+1}\tau'$ and $\Gamma_2 \vdash v:\tau$, we can easily show (by induction on derivation of $\Gamma_2 \vdash v:\tau$) that there exists Γ_2' such that $\Gamma_2' \vdash v:\tau'$ and $\Gamma_2 \leq \uparrow^{n+1}\Gamma_2'$. So, by applying the induction hypothesis to $\Gamma_3, x:\tau', z:\sigma \vdash_\prec P_1$, we get

$$\Gamma_2' \,|\, \Gamma_3, z:\sigma \vdash_\prec [x \mapsto v]P_1.$$

By applying T-In, we obtain

$$y:\mathbf{chan}(\sigma, ?^0_n);_\prec(\Gamma_2' \,|\, \Gamma_3) \vdash_\prec y?z.\,[x \mapsto v]P_1.$$

By Lemma 2 and $\Gamma_1, x:\tau \leq y:\mathbf{chan}(\sigma, ?^0_n);_\prec(\Gamma_3, x:\tau')$, we have

$$\begin{aligned}\Gamma_1 \,|\, \Gamma_2 &\leq (y:\mathbf{chan}(\sigma, ?^0_n);_\prec\Gamma_3) \,|\, \Gamma_2\\ &\leq y:\mathbf{chan}(\sigma, ?^0_n);_\prec(\Gamma_2' \,|\, \Gamma_3)\end{aligned}$$

Therefore, by applying T-Weak, we obtain

$$\Gamma_1 \,|\, \Gamma_2 \vdash_\prec y?z.\,[x \mapsto v]P_1$$

as required.

The next lemma says that typing is preserved by the structural relation.

Lemma 4. *If $\Gamma \vdash_\prec P$ and $P \preceq Q$, then $\Gamma \vdash_\prec Q$.*

Proof. Straightforward induction on derivation of $P \preceq Q$.

Lemma 5. *If $\Gamma \leq \Gamma' \longrightarrow \Delta'$, then there exists Δ such that $\Gamma \longrightarrow \Delta$ and $\Delta \leq \Delta'$.*

Proof. This follows immediately from the definition of $\Gamma \longrightarrow \Delta$ and the subusage relation $U \longrightarrow U'$.

Once we have prepared the above lemmas, we can prove the type preservation in a standard manner.

Proof of Theorem 1 The proof proceeds by induction on derivation of $P \longrightarrow Q$, with case analysis on the last rule used. We show only the case for R-COM, which is the only non-trivial case. Suppose $\Gamma \vdash_{\prec} x!^{t_1} v.\, P_1 \,|\, x?^{t_2} y.\, P_2$. Then, by the typing rules, we must have:

$$\begin{array}{l} \Gamma_1 \vdash_{\prec} P_1 \\ \Gamma_2 \vdash v : \tau \\ \Gamma_3, y : \tau \vdash_{\prec} P_2 \\ \Gamma \leq (x : \mathbf{chan}(\tau, !^0_{t_1});_{\prec} (\Gamma_1 \,|\, \Gamma_2)) \,|\, (x : \mathbf{chan}(\tau, ?^0_{t_1});_{\prec} \Gamma_3) \end{array}$$

By the assumption on bound variables, we can assume without loss of generality that y does not appear in $\prec$. Therefore, by applying the substitution lemma (Lemma 3), we obtain $\Gamma_2 \,|\, \Gamma_3 \vdash_{\prec} [y \mapsto v]P_2$. So, we obtain $\Gamma_1 \,|\, \Gamma_2 \,|\, \Gamma_3 \vdash_{\prec} P_1 \,|\, [x \mapsto v]P_2$. Since

$$\Gamma \leq (x : \mathbf{chan}(\tau, !^0_{t_1});_{\prec} (\Gamma_1 \,|\, \Gamma_2)) \,|\, (x : \mathbf{chan}(\tau, ?^0_{t_1});_{\prec} \Gamma_3) \longrightarrow_{\leq} \Gamma_1 \,|\, \Gamma_2 \,|\, \Gamma_3$$

holds, by using Lemma 5, we obtain Δ such that $\Gamma \longrightarrow \Delta$ and $\Delta \leq \Gamma_1 \,|\, \Gamma_2 \,|\, \Gamma_3$. Thus, by using T-WEAK, we get $\Delta \vdash_{\prec} P_1 \,|\, [x \mapsto v]P_2$ as required. □

C.2 Proof of Theorem 2

We need some preliminary definitions and lemmas.

We first introduce the notion of normal forms.

Definition C1 *A process N is* in normal form *if all the if-expressions and let-expressions are guarded by input or output prefixes, i.e., if it is generated by the following syntax.*

$$N ::= \mathbf{0} \mid x!^t v.\, P \mid x?^t y.\, P \mid (N_1 \,|\, N_2) \mid *N \mid (\nu x)\, N$$

Here, P ranges over the set of (arbitrary) processes.

Definition C2 *The extended structural relation $\preceq'$ is the least relation closed under the rules for $\preceq$ (with $\preceq$ being replaced by $\preceq'$), plus the following rule.*

$$\frac{P \preceq' Q}{*P \preceq' *Q}$$

The following lemma means that the replacement of $\preceq$ with $\preceq'$ does not change (one-step) reducibility of a process.

Lemma 6. *If $P \preceq' P' \longrightarrow Q'$, then there exists Q such that $P \longrightarrow Q$.*

The following lemma means that a closed, well-typed process can be always converted to a process in normal form.

Lemma 7. *If $\emptyset \vdash_{\prec} P$, then there exists a normal form process N such that $P \preceq' N$, and $\emptyset \vdash_{\prec} N$ holds.*

In the proof of Theorem 2, we need to identify certain bound variables. Since α-conversion can be implicitly performed in type derivation, however, we need to introduce a unique identifier of a bound variable. $\mathit{Index}(x, P)$ defined below serves as such a unique identifier. In the definition below, α-conversion is forbidden.

Definition C3 *The* index *of x in P, $\mathit{Index}(x, P)$ is defined by:*

$$
\begin{array}{l}
\mathit{Index}(x, (\nu x)\, P) = \epsilon \\
\mathit{Index}(x, (\nu y)\, P) = 0.\mathit{Index}(x, P) \\
\mathit{Index}(x, P_1 \mid P_2) = \begin{cases} 1.\mathit{Index}(x, P_1) & \mathit{if}\ (\nu x)\ \mathit{appears\ in}\ P_1 \\ 2.\mathit{Index}(x, P_2) & \mathit{if}\ (\nu x)\ \mathit{appears\ in}\ P_2 \\ \mathit{undefined} & \mathit{otherwise} \end{cases} \\
\mathit{Index}(x, *P) = \mathit{Index}(x, P)
\end{array}
$$

If (νx) does not appear in P or is guarded by input or output prefixes, $\mathit{Index}(x, P)$ is undefined.

We are now ready to prove the main theorem.

Proof of Theorem 2 Suppose that $\emptyset \vdash_{\prec} P$ and either $P \preceq' Q = (\nu \widetilde{x})\,(x!^n v.\, Q_1 \mid Q_2)$ or $P \preceq' Q = (\nu \widetilde{x})\,(x?^n y.\, Q_1 \mid Q_2)$ with $n \in \mathbf{Nat}$. Since P is closed, we can assume without loss of generality that $\prec = \emptyset$. We can also assume without loss of generality that Q is in normal form. For a sub-process N in normal form, we define the set $\mathit{Waiting}_Q(N)$ inductively as follows.

$$
\begin{array}{l}
\mathit{Waiting}_Q(\mathbf{0}) = \emptyset \\
\mathit{Waiting}_Q(x!^m v.\, R) = \{(n, \mathit{Index}(x, Q), x!^m v.\, R)\} \\
\mathit{Waiting}_Q(x?^m y.\, R) = \{(n, \mathit{Index}(x, Q), x?^m y.\, R)\} \\
\mathit{Waiting}_Q(N_1 \mid N_2) = \mathit{Waiting}_Q(N_1) \cup \mathit{Waiting}_Q(N_2) \\
\mathit{Waiting}_Q(*N) = \mathit{Waiting}_Q(N) \\
\mathit{Waiting}_Q((\nu x)\, N) = \mathit{Waiting}_Q(N)
\end{array}
$$

Let (m, π, N) be a minimum element of $\mathit{Waiting}_Q(Q)$ with respect to the following order $<$:

$$(m, \pi, N) < (m', \pi', N') \iff m < m' \text{ or } (m = m' \text{ and } \pi' \text{ is a prefix of } \pi)$$

Note that m is finite (i.e., $m \in \mathbf{Nat}$), since Q is $(\nu \widetilde{x})\,(x!^n v.\, Q_1 \mid Q_2)$ or $(\nu \widetilde{x})\,(x?^n y.\, Q_1 \mid Q_2)$ with $n \in \mathbf{Nat}$.

Assume that N is of the form $x!^m v.\, R$. (The case where N is $x?^m y.\, P$ is similar.) By the assumptions, Q contains a process of the form $(\nu x)\, Q_1$, and Q_1 contains N as a sub-expression. Since $\emptyset \vdash_{\emptyset} Q$, we have $\Gamma, x : \mathbf{chan}(\tau, U) \vdash_{\prec'} Q_1$, where $\mathit{rel}(U)$. Moreover, $x_1 \prec' x_2$ holds only if $\mathit{Index}(x_2, Q)$ is a prefix of $\mathit{Index}(x_1, Q)$. Since N is unguarded by any input or output prefix in Q_1, it must be the case that $\mathit{cap}_!(U) \le m$. Therefore, since $\mathit{rel}(U)$, it must be the case that $\mathit{ob}_?(U) \le \mathit{cap}_!(U) \le m$. That implies Q_1 must contain an input or output process that is typed under a context of the form $\Gamma', x : \mathbf{chan}(\tau, U_1)$, where $\mathit{ob}_?(U_1) \le m$.

Moreover, since (m, π, N) is the minimum element of $\mathit{Waiting}_Q(Q)$, such an input or output process must be of the form $x?^t y.\, R$. (Notice that if the outermost input or output prefix were on another channel w, the capability annotation of the guard must have been either smaller than m, or equal to m and $\mathit{Index}(x, Q)$ must be a prefix of $\mathit{Index}(w, Q)$. That contradicts with the assumption that (m, π, N) is the minimum element.) Thus, $Q_1 \longrightarrow Q_1'$, which implies $P \preceq' Q \longrightarrow Q'$. By Lemma 6, we have $P \longrightarrow P'$ for some P' as required. □

C.3 Proof of Lemma 1

Lemma 8. *If $\Gamma \vdash_{\prec'} P$ and $\prec' \subseteq \prec$, then $\Gamma \vdash_{\prec} P$.*

Proof. Straightforward induction on derivation of $\Gamma \vdash_{\prec'} P$.

Proof of Lemma 1 The proof proceeds by induction on derivation of $\mathcal{T} \vdash M : \theta$, with case analysis on the last rule used.

- Case TL-Var: In this case, $M = x$ and $\mathcal{T} = \mathcal{T}', x : \theta$. Using rule T-Out and T-Zero, we get
$$x : \uparrow^{\infty} [\![\theta]\!], r : \mathbf{chan}([\![\theta]\!], !^{0}_{\infty}) \vdash_{\emptyset} r!x$$
By using T-Weak, we obtain
$$[\![\mathcal{T}]\!], r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty}) \vdash_{\emptyset} r!x$$
as required.
- Case TL-Fix: In this case, $M = \mathbf{fix}(f, x, M_1)$ and $\theta = \theta_1 \rightarrow \theta_2$, with
$$\mathcal{T}, f : \theta_1 \rightarrow \theta_2, x : \theta_1 \vdash M_1 : \theta_2.$$
By the induction hypothesis, we have
$$[\![\mathcal{T}]\!], f : \mathbf{chan}([\![\theta_1]\!] \times \mathbf{chan}([\![\theta_2]\!], !^{1}_{\infty}), *!^{\infty}_{0}), x : [\![\theta_1]\!], r' : \mathbf{chan}([\![\theta_2]\!], !^{1}_{\infty}) \vdash_{\emptyset} [\![M_1]\!]^{r'}.$$
Using T-In, T-Let and T-Rep, we obtain:
$$*\uparrow^{\infty} [\![\mathcal{T}]\!], f : \mathbf{chan}([\![\theta_1]\!] \times \mathbf{chan}([\![\theta_2]\!], !^{1}_{\infty}), *?^{0}_{\infty}.*!^{\infty}_{0}) \vdash_{\emptyset} *f?x, r'.\, [\![M_1]\!]^{r'}.$$
Since $[\![\mathcal{T}]\!] \leq *\uparrow^{\infty} [\![\mathcal{T}]\!]$, we obtain
$$[\![\mathcal{T}]\!], f : \mathbf{chan}([\![\theta_1]\!] \times \mathbf{chan}([\![\theta_2]\!], !^{1}_{\infty}), *?^{0}_{\infty}.*!^{\infty}_{0}) \vdash_{\emptyset} *f?x, r'.\, [\![M_1]\!]^{r'}$$
by using T-Weak. Using T-Out, T-Zero, we also obtain
$$f : \uparrow^{\infty} [\![\theta_1 \rightarrow \theta_2]\!], r : \mathbf{chan}([\![\theta_1 \rightarrow \theta_2]\!], !^{1}_{\infty}) \vdash_{\emptyset} r!f.$$
So, we get
$$\begin{array}{c}[\![\mathcal{T}]\!], f : \mathbf{chan}([\![\theta_1]\!] \times \mathbf{chan}([\![\theta_2]\!], !^{1}_{\infty}), *!^{\infty}_{0} \mid *?^{0}_{\infty}.*!^{\infty}_{0}), r : \mathbf{chan}([\![\theta_1 \rightarrow \theta_2]\!], !^{1}_{\infty}) \\ \vdash_{\emptyset} r!f \mid *f?x, r'.\, [\![M_1]\!]^{r'}\end{array}$$

Since $rel(*!^{\infty}_{0} \mid *?^{0}_{\infty}.*!^{\infty}_{0})$ holds, we can apply T-New to obtain

$$[\![\mathcal{T}]\!], r : \mathbf{chan}([\![\theta_1 \to \theta_2]\!], !^{1}_{\infty}) \vdash_{\emptyset} [\![\mathbf{fix}(f, x, M_1)]\!]^{r}$$

as required.

– Case T-App: In this case, $M = M_1 M_2$ with: $\mathcal{T} \vdash M_1 : \theta_1 \to \theta$ and $\mathcal{T} \vdash M_2 : \theta_1$. By the induction hypothesis, we have:

$$[\![\mathcal{T}]\!], r_1 : \mathbf{chan}([\![\theta_1 \to \theta]\!], !^{1}_{\infty}) \vdash_{\emptyset} [\![M_1]\!]^{r_1}$$
$$[\![\mathcal{T}]\!], r_2 : \mathbf{chan}([\![\theta_1]\!], !^{1}_{\infty}) \vdash_{\emptyset} [\![M_2]\!]^{r_2}$$

Let $\prec = \{(r_1, x) \mid x \in \{r\} \cup FV(M)\} \cup \{(r_2, x) \mid x \in \{r, r_1\} \cup FV(M)\}$. Then, by using T-Out, T-Zero, and T-Weak, we get

$$f : [\![\theta_1 \to \theta]\!], x : [\![\theta_1]\!], r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty}) \vdash_{\prec} f!(x, r).$$

By using T-In, we get

$$r_2 : \mathbf{chan}([\![\theta_1]\!], ?^{\infty}_{1}) ;_{\prec} (f : [\![\theta_1 \to \theta]\!], r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty})) \vdash_{\prec} r_2?^{*1}[x].\, f!(x, r).$$

That is,

$$r_2 : \mathbf{chan}([\![\theta_1]\!], ?^{\infty}_{1}), f : [\![\theta_1 \to \theta]\!], r : \uparrow^{1}\mathbf{chan}([\![\theta]\!], !^{1}_{\infty})) \vdash_{\prec} r_2?^{*1}[x].\, f!(x, r).$$

(Note that this is the very place where the relation $r_2 \prec r$ is important. By using T-Weak, we obtain

$$r_2 : \mathbf{chan}([\![\theta_1]\!], ?^{\infty}_{1}), f : [\![\theta_1 \to \theta]\!], r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty})) \vdash_{\prec} r_2?^{*1}[x].\, f!(x, r).$$

By applying T-In and T-Weak again, we further obtain:

$$r_2 : \mathbf{chan}([\![\theta_1]\!], ?^{\infty}_{1}), r_1 : \mathbf{chan}([\![\theta_1 \to \theta]\!], ?^{\infty}_{1}), r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty})) \vdash_{\prec} r_1?^{*1}[f].\, r_2?^{*1}[x].\, f!(x, r).$$

So, we obtain

$$\begin{array}{l}[\![\mathcal{T}]\!], r_1 : \mathbf{chan}([\![\theta_1 \to \theta]\!], ?^{\infty}_{1} \mid !^{1}_{\infty}), r_2 : \mathbf{chan}([\![\theta_1]\!], ?^{\infty}_{1} \mid !^{1}_{\infty}), r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty})) \\ \qquad \vdash_{\prec} [\![M_1]\!]^{r_1} \mid [\![M_2]\!]^{r_2} \mid r_1?^{*1}[f].\, r_2?^{*1}[x].\, f!(x, r).\end{array}$$

Since $rel(?^{\infty}_{1} \mid !^{1}_{\infty})$ holds, by applying T-New twice, we obtain:

$$[\![\mathcal{T}]\!], r : \mathbf{chan}([\![\theta]\!], !^{1}_{\infty})) \vdash_{\prec} [\![M]\!]^{r}$$

as required.

□