A Generic Type System for the Pi-Calculus

Atsushi Igarashi
Department of Graphics and Computer Science
Graduate School of Arts and Sciences
University of Tokyo
email:igarashi@graco.c.u-tokyo.ac.jp

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

Abstract

We propose a general, powerful framework of type systems for the π-calculus, and show that we can obtain as its instances a variety of type systems guaranteeing non-trivial properties like deadlock-freedom and race-freedom. A key idea is to express types and type environments as abstract processes: We can check various properties of a process by checking the corresponding properties of its type environment. The framework clarifies the essence of recent complex type systems, and it also enables sharing of a large amount of work such as a proof of type preservation, making it easy to develop new type systems.

1 Introduction

1.1 Motivation

Static guarantee of the correctness of concurrent programs is important: Because concurrent programs are more complex than sequential programs (due to non-determinism, deadlock, etc.), it is hard for programmers to debug concurrent programs or reason about their behavior.

A number of advanced type systems have recently been proposed to analyze various properties of concurrent programs, such as input/output modes [27], multiplicities (how often each channel is used) [17], race conditions [4, 6], deadlock [14, 18, 30, 38], livelock [16], and information flow [8, 10].

Unfortunately, however, there has been no satisfactorily general framework of type systems for concurrent programming languages: Most type systems have been designed in a rather ad hoc manner for guaranteeing certain specific properties. The lack of a general framework kept it difficult to compare, integrate, or extend the existing type systems. Also, a lot of tasks (such as proving type soundness) had to be repeated for each type system. This situation stands in contrast with that of type systems for functional programming languages, where a number of useful analyses (such as side-effect analysis, region inference [36], and exception analysis [3, 26]) can be obtained as instances of the effect analysis [34, 35].

To appear in Proceedings of The 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2001). ©ACM 2000

The goal of this paper is therefore to establish a general framework of type systems for concurrent processes, so that various advanced type systems can be derived as its instances. As in many other type systems, we use the π-calculus as a target language: It is simple yet expressive enough to model modern concurrent/distributed programming languages.

1.2 Main Ideas

A main idea of the present work is to express types and type environments as *abstract processes*. So, a type judgment $\Gamma \triangleright P$, which is normally read as "The process P is well-typed under the type environment Γ," means that the abstract process Γ is a correct abstraction of the process P, in the sense that P satisfies a certain property (like race-freedom and deadlock-freedom) if its abstraction Γ satisfies the corresponding property. (In this sense, our type system may be regarded as a kind of abstract interpretation [2].) We define such a relation $\Gamma \triangleright P$ by using typing rules. Because we use a much simpler process calculus to express type environments than the π-calculus, it is easier to check properties of Γ than to check those of P directly.

To see how type environments can be expressed as abstract processes, let us review the ideas of our previous type systems for deadlock/livelock-freedom [16, 18, 33]. Let $x![y_1, \ldots, y_n].P$ be a process that sends the tuple $[y_1, \ldots, y_n]$ along the channel x and then behaves like P, and $x?[y_1, \ldots, y_n].Q$ be a process that receives a tuple $[z_1, \ldots, z_n]$ along x, binds $y_1, \ldots, y_n$ to $z_1, \ldots, z_n$, and then behaves like Q. Let us write $P \,|\, Q$ for a parallel execution of P and Q, and $\mathbf{0}$ for inaction. In our previous type systems [18, 33], the process $P = x![z] \,|\, x?[y].\, y![\,] \,|\, z?[\,].\, z?[\,].\mathbf{0}$ is roughly typed as follows:

$$x : [[\,]/O]/(O|I), z : [\,]/(O|I.I) \triangleright P$$

Types of the form $[\tau_1, \ldots, \tau_n]/U$ are channel types: The part $[\tau_1, \ldots, \tau_n]$ means that the channel is used for communicating a tuple of values of types $\tau_1, \ldots, \tau_n$, and the part U (called a usage) expresses how channels are used for input/output. For example, the part $O|I.I$ of the type of z means that z is used for output (denoted by O) and for two successive inputs (denoted by $I.I$) in parallel. By focusing on the usage parts, we can view the above type environment as a collection of abstract processes, each of which performs a pair of co-actions I and O on each channel. Indeed, we can reduce the type environment by canceling I and O of the usage of x and obtain $x : [[\,]/O]/0, z : [\,]/(O|I.I)$, which is a

type environment of the process $z![] \mid z?[].\, z?[].\, \mathbf{0}$, obtained by reducing P. By further reducing the type environment, we obtain $x : [[]/O]/0, z : []/I$, which indicates that an input on z may remain after P is fully reduced. Based on this idea, we developed type systems for deadlock/livelock-freedom [16, 18, 33].

We push the above "type environments as abstract processes" view further, and express type environments as CCS-like processes (unlike in CCS [22], however, we have no operator for hiding or creating channels). The type environment of the above process P is expressed as $x![\tau].\, z![\tau'] \mid x?[\tau].\, \mathbf{0} \mid z?[\tau'].\, z?[\tau'].\, \mathbf{0}$. It represents not only how each channel is used, but also the order of communications on different channels, such as the fact that an output on z occurs only after an output on x succeeds (as indicated by the part $x![\tau].\, z![\tau']$). The parts enclosed by square brackets abstract the usage of values transmitted through channels. Thanks to this generalization, we can reason about not only deadlock-freedom but also other properties such as race conditions within a single framework. The new type system can also guarantee deadlock-freedom of more processes, such as that of concurrent objects with non-uniform service availability [30, 31].

1.3 Contributions

Contributions of this paper are summarized as follows:

- We develop a general framework of type systems, which we call a *generic type system* — just as a generic function is parametrized by types and can be instantiated to functions on various arguments by changing the types, the generic type system is parameterized by a subtyping relation and a consistency condition of types and it can be instantiated to a variety of type systems by changing the subtyping relation and the consistency condition.
- We prove that the general type system satisfies several important properties (such as subject reduction), independently of a choice of the subtyping relation and the consistency condition. Therefore, there is no need to prove them for each type system.
- We show that a variety of non-trivial type systems (such as those ensuring deadlock-freedom and race-freedom) can indeed be derived as instances of the general type system, and prove their correctness.

1.4 The Rest of This Paper

Section 2 introduces the syntax and the operational semantics of our target process calculus. Section 3 presents our generic type system and shows its properties. As its instances, Section 4 derives a variety of type systems and proves their correctness. To further demonstrate the strength of our framework, Section 5 shows that deadlock and race conditions of concurrent objects can also be analyzed within our generic type system. Section 6 describes preliminary results of our ongoing studies on the power of our generic type system. Section 7 discusses limitations and extensions of our generic type system. Section 8 discusses related work and Section 9 concludes this paper. For the space restriction, we omit some technical details in this extended abstract. They are found in the full paper [11].[1]

2 Target Language

2.1 Syntax

Our calculus is basically a subset of the polyadic π-calculus [23]. To state properties of a process, we annotate each input or output operation with a label.

Definition 2.1.1 [processes]: The set of processes is defined by the following syntax.

$$\begin{array}{lll} P \text{ (processes)} & ::= & \mathbf{0} \mid G_1 + \cdots + G_n \mid (P \mid Q) \\ & & \mid (\nu x_1, \ldots, x_n)\, P \mid *P \\ G \text{ (guarded processes)} & & \\ & ::= & x!^{\mathbf{t}}[y_1, \ldots, y_n].\, P \mid x?^{\mathbf{t}}[y_1, \ldots, y_n].\, P \end{array}$$

Here, x, y, and z range over a countably infinite set $\mathbf{Var}$ of variables. $\mathbf{t}$ ranges over a countably infinite set $\mathbf{T}$ of labels called *events*. We assume that $\mathbf{Var} \cap \mathbf{T} = \emptyset$.

Notation 2.1.2: We write $\tilde{x}$ for a (possibly empty) sequence $x_1, \ldots, x_n$, and $\|\tilde{x}\|$ for the length n of the sequence $\tilde{x}$. $(\nu \tilde{x_1}) \cdots (\nu \tilde{x_n})\, P$ is abbreviated to $(\widetilde{\nu x}_{1..n})\, P$ or $(\widetilde{\nu x})\, P$. As usual, $\tilde{y}$ in $x?[\tilde{y}].\, P$ and $\tilde{x}$ in $(\nu \tilde{x})\, P$ are called bound variables. The other variables are called free variables. We assume that α-conversions are implicitly applied so that bound variables are always different from each other and from free variables. The expression $[z_1/x_1, \ldots, z_n/x_n]P$, abbreviated to $[\tilde{z}/\tilde{x}]P$, denotes a process obtained from P by replacing all free occurrences of $x_1, \ldots, x_n$ with $z_1, \ldots, z_n$. We often omit the inaction $\mathbf{0}$ and write $x!^{\mathbf{t}}[\tilde{y}]$ for $x!^{\mathbf{t}}[\tilde{y}].\, \mathbf{0}$. When events are not important, we omit them and just write $x![\tilde{y}].\, P$ and $x?[\tilde{y}].\, P$ for $x!^{\mathbf{t}}[\tilde{y}].\, P$ and $x?^{\mathbf{t}}[\tilde{y}].\, P$ respectively. We give a higher precedence to $+$ and $(\nu \tilde{x})$ than to $\mid$.

The meanings of $\mathbf{0}$, $x!^{\mathbf{t}}[\tilde{y}].\, P$, $x?^{\mathbf{t}}[\tilde{y}].\, P$, and $P \mid Q$ have already been explained. $G_1 + \cdots + G_n$ (where $G_1, \ldots, G_n$ are input or output processes) denotes an external choice: It behaves like one of $G_1, \ldots, G_n$ depending on enabled communications. $(\nu \tilde{x})\, P$ creates fresh channels $\tilde{x}$ and then executes P.[2] $*P$ denotes infinitely many copies of P running in parallel.

2.2 Operational Semantics

As usual [23], we define a reduction semantics by using a structural relation and a reduction relation. For technical convenience, we do not require the structural relation to be symmetric. The reduction relation $P \longrightarrow Q$ is annotated with a term of the form $x^{\mathbf{t},\mathbf{t}'}$ or $\epsilon^{\mathbf{t},\mathbf{t}'}$. The term records on which channel and events the reduction is performed: It is used to state properties of a process in Section 4.

Definition 2.2.1: The *structural preorder* $\preceq$ is the least reflexive and transitive relation closed under the rules in Figure 1 ($P \equiv Q$ denotes $(P \preceq Q) \wedge (Q \preceq P)$).

Definition 2.2.2: The reduction relation $\xrightarrow{l}$ is the least relation closed under the rules in Figure 2.

[1] Available from `http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html`.

[2] This is operationally the same as $(\nu x_1) \cdots (\nu x_n)\, P$, but we distinguish them in the type system given in Section 3.

$$P \mid \mathbf{0} \equiv P \quad \text{(SPCong-Zero)}$$

$$P \mid Q \equiv Q \mid P \quad \text{(SPCong-Commut)}$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad \text{(SPCong-Assoc)}$$

$$*P \preceq *P \mid P \quad \text{(SPCong-Rep)}$$

$$(\nu\tilde{x})\, P \mid Q \preceq (\nu\tilde{x})\, (P \mid Q) \text{(if } \tilde{x} \text{ are not free in } Q) \quad \text{(SPCong-New)}$$

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

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

Figure 1: Structural Preorder

$$\cdots + x!^{\mathbf{t}}[\tilde{z}].\, P + \cdots \mid \cdots + x?^{\mathbf{t}'}[\tilde{y}].\, Q + \cdots \xrightarrow{x^{\mathbf{t},\mathbf{t}'}} P \mid [\tilde{z}/\tilde{y}]Q \quad \text{(R-Com)}$$

$$\frac{P \xrightarrow{l} Q}{P \mid R \xrightarrow{l} Q \mid R} \quad \text{(R-Par)} \qquad \frac{P \xrightarrow{l} Q \qquad (l = \epsilon^{\mathbf{t},\mathbf{t}'}) \vee (l = y^{\mathbf{t},\mathbf{t}'} \wedge y \notin \{\tilde{x}\})}{(\nu\tilde{x})\, P \xrightarrow{l} (\nu\tilde{x})\, Q} \quad \text{(R-New2)}$$

$$\frac{P \xrightarrow{y^{\mathbf{t},\mathbf{t}'}} Q \qquad y \in \{\tilde{x}\}}{(\nu\tilde{x})\, P \xrightarrow{\epsilon^{\mathbf{t},\mathbf{t}'}} (\nu\tilde{x})\, Q} \quad \text{(R-New1)} \qquad \frac{P \preceq P' \qquad P' \xrightarrow{l} Q' \qquad Q' \preceq Q}{P \xrightarrow{l} Q} \quad \text{(R-SPCong)}$$

Figure 2: Reduction Relation

Notation 2.2.3: We write $P \longrightarrow Q$ if $P \xrightarrow{l} Q$ for some l.

Notation 2.2.4: When $\mathcal{R}_1, \mathcal{R}_2$ are binary relations on a set S, we write $\mathcal{R}_1^*$ for the reflexive and transitive closure of $\mathcal{R}_1$, and $\mathcal{R}_1\mathcal{R}_2$ for the composition of $\mathcal{R}_1$ and $\mathcal{R}_2$.

3 Generic Type System

3.1 Types

As explained in Section 1, we extend ordinary type environments and express them as abstract processes. In the rest of this paper, we call them *process types* (or, just types in short). For most of the process constructors introduced in the previous section, there are the corresponding constructors for process types. We use the same symbols for them to clarify the correspondence.

Definition 3.1.1 [types]: The sets of tuple types and process types are defined by the following syntax.

$$\begin{array}{l} \tau \text{ (tuple types)} \ ::= (x_1, \ldots, x_n)\Gamma \\ \Gamma \text{ (process types)} \\ \qquad ::= \mathbf{0} \mid \alpha \mid \gamma_1 + \cdots + \gamma_n \mid (\Gamma_1 \mid \Gamma_2) \mid \Gamma_1 \& \Gamma_2 \mid \mu\alpha.\Gamma \\ \gamma ::= x!^{\mathbf{t}}[\tau].\, \Gamma \mid x?^{\mathbf{t}}[\tau].\, \Gamma \mid \mathbf{t}.\Gamma \end{array}$$

Here, the metavariable α ranges over the set of type variables.

Notation 3.1.2: The tuple type $(\tilde{x})\Gamma$ binds the variables $\tilde{x}$ in Γ. We assume that α-conversions are implicitly applied so that bound variables are always different from each other and free variables. In this paper, we restrict the syntax of tuple types so that $(\tilde{x})\Gamma$ does not contain any free variables or free type variables. (It is possible to remove this restriction: See Section 7.) We write $[\tilde{y}/\tilde{x}]\Gamma$ for a process type obtained by substituting $\tilde{y}$ for all free occurrences of $\tilde{x}$ in Γ. We write $*\Gamma$ for $\mu\alpha.(\Gamma \mid \alpha)$. We often omit $\mathbf{0}$ and write $x!^{\mathbf{t}}[\tau]$ and $x?^{\mathbf{t}}[\tau]$ for $x!^{\mathbf{t}}[\tau].\, \mathbf{0}$ and $x?^{\mathbf{t}}[\tau].\, \mathbf{0}$ respectively. We also abbreviate $x!^{\mathbf{t}}[()\mathbf{0}].\, \Gamma$ and $x?^{\mathbf{t}}[()\mathbf{0}].\, \Gamma$ to $x!^{\mathbf{t}}[].\, \Gamma$ and $x?^{\mathbf{t}}[].\, \Gamma$ respectively. We assume that $\mu\alpha.\Gamma$ binds α in Γ. We write $[\Gamma/\alpha]$ for the capture-avoiding substitution of Γ for α. We write $Null(\Gamma)$ if Γ does not contain a process type of the form $x?^{\mathbf{t}}[\tau].\, \Gamma_1$ or $x!^{\mathbf{t}}[\tau].\, \Gamma_1$. When $\tau = (\tilde{x})\Gamma$, we call $\|\tilde{x}\|$ the arity of τ and write $\|\tau\|$.

The tuple type $(x_1, \ldots, x_n)\Gamma$ is the type of an n-tuple, whose elements $x_1, \ldots, x_n$ should be used according to Γ. $\mathbf{0}$ is the type of the inaction. $x!^{\mathbf{t}}[\tau].\, \Gamma$ is the type of a process that uses x for sending a tuple of type τ, and then behaves according to Γ. The output on x must be tagged with $\mathbf{t}$. Similarly, $x?^{\mathbf{t}}[\tau].\, \Gamma$ is the type of a process that uses x for receiving a tuple of type τ, and then behaves according to Γ. For example, if a process should have type $x?^{\mathbf{t}_1}[].\, y!^{\mathbf{t}_2}[].\, \mathbf{0}$, then $x?^{\mathbf{t}_1}[].\, y!^{\mathbf{t}_2}[]$ is allowed but neither $y!^{\mathbf{t}_2}[].\, x?^{\mathbf{t}_1}[].\, \mathbf{0}$ nor $x?^{\mathbf{t}_1}[].\, \mathbf{0} \mid y!^{\mathbf{t}_2}[]$ is. In this way, we can express more precise information on the usage of channels than previous type systems [17, 18, 27]. $\mathbf{t}.\Gamma$ is the type of a process that behaves according to Γ after some action annotated with $\mathbf{t}$ (which is an input or an output action on some channel) occurs.[3] $\Gamma_1 \mid \Gamma_2$ is the type of a process that behaves according to Γ_1 and Γ_2 in parallel. The type $\gamma_1 + \cdots + \gamma_n$ represents an external choice: A process of that type must behave according to one of $\gamma_1, \ldots, \gamma_n$, depending on the communications provided by the environment. On the other hand, the type $\Gamma_1 \& \Gamma_2$ represents an internal choice: A process of that type can behave according to either Γ_1 or Γ_2, irrespectively of what communications are provided by the environment. For example, the process type $x!^{\mathbf{t}}[].\, y!^{\mathbf{t}'}[] \& y!^{\mathbf{t}'}[].\, x!^{\mathbf{t}}[]$ means that x and y can be used sequentially for output in any order; So, both $x!^{\mathbf{t}}[].\, y!^{\mathbf{t}'}[]$ and $y!^{\mathbf{t}'}[].\, x!^{\mathbf{t}}[]$ can have this type.[4]

[3] Instead of $\mathbf{t}.\Gamma$, we could introduce process types $\mathbf{t}?.\Gamma$ and $\mathbf{t}!.\Gamma$ to distinguish between input and output actions. We do not do so in this paper for simplicity.

[4] So, $\Gamma_1 \& \Gamma_2$ is similar to an intersection type $\Gamma_1 \wedge \Gamma_2$. The difference is that a value of type $\Gamma_1 \& \Gamma_2$ can be used *only once* according to either Γ_1 or Γ_2.

We use the standard notation $\mu\alpha.\Gamma$ for recursive types. For example, $\mu\alpha.(x?^{\mathbf{t}}[\tau].\,\alpha)$ is the type of a process that uses x for receiving a tuple of type τ repeatedly.

Notice that, unlike the π-calculus processes in Section 2, the process types contain no operators for creating fresh channels or passing channels through other channels. Thanks to this, we can check properties of types (as abstract processes) more easily than those of the π-calculus processes. Instead, we have some operators that do not have their counterparts in processes. A process type of the form $\mathbf{t}.\Gamma$ plays an important role in guaranteeing complex properties like deadlock-freedom. For example, we can express the type of $(\nu x)\,(x?^{\mathbf{t}_1}[].\,y!^{\mathbf{t}_2}[]\,|\,y?^{\mathbf{t}_3}[].\,\mathbf{0})$ as $\mathbf{t}_1.y!^{\mathbf{t}_2}[]\,|\,y?^{\mathbf{t}_3}[]$, which implies that the output on y is not performed until an action labelled with $\mathbf{t}_1$ succeeds. Since actually it never succeeds, we know that the input from y is kept waiting forever.

3.2 Subtyping

We introduce a subtyping relation $\Gamma_1 \leq \Gamma_2$, meaning that a process of type Γ_1 may behave like that of type Γ_2. For example, $\Gamma_1 \& \Gamma_2 \leq \Gamma_1$ should hold. The subtyping relation depends on the property we want to guarantee: For example, if we are only concerned with arity-mismatch errors [7, 37], we may identify $\mathbf{t}.\Gamma$ with Γ, and $x!^{\mathbf{t}}[\tau].\,\Gamma$ with $x!^{\mathbf{t}}[\tau]\,|\,\Gamma$, but we cannot do so if we are concerned with more complex properties like deadlock-freedom. Therefore, we state here only necessary conditions that should be satisfied by the subtyping relations of all instances of our type system.

We need some auxiliary operations.

Definition 3.2.1: Let S be a subset of $\mathbf{Var}$. Unary operations $\Gamma{\downarrow_S}$ and $\Gamma{\uparrow_S}$ on process types are defined by:

$$
\begin{array}{rcl}
\mathbf{0}{\downarrow_S} & = & \mathbf{0} \\
\alpha{\downarrow_S} & = & \alpha \\
(x?^{\mathbf{t}}[\tau].\,\Gamma){\downarrow_S} & = & \begin{cases} x?^{\mathbf{t}}[\tau].\,(\Gamma{\downarrow_S}) & \text{if } x \in S \\ \mathbf{t}.(\Gamma{\downarrow_S}) & \text{otherwise} \end{cases} \\
(x!^{\mathbf{t}}[\tau].\,\Gamma){\downarrow_S} & = & \begin{cases} x!^{\mathbf{t}}[\tau].\,(\Gamma{\downarrow_S}) & \text{if } x \in S \\ \mathbf{t}.(\Gamma{\downarrow_S}) & \text{otherwise} \end{cases} \\
(\mathbf{t}.\Gamma){\downarrow_S} & = & \mathbf{t}.(\Gamma{\downarrow_S}) \\
(\gamma_1 + \cdots + \gamma_n){\downarrow_S} & = & (\gamma_1{\downarrow_S}) + \cdots + (\gamma_n{\downarrow_S}) \\
(\Gamma_1\,|\,\Gamma_2){\downarrow_S} & = & (\Gamma_1{\downarrow_S})\,|\,(\Gamma_2{\downarrow_S}) \\
(\Gamma_1 \& \Gamma_2){\downarrow_S} & = & (\Gamma_1{\downarrow_S}) \& (\Gamma_2{\downarrow_S}) \\
(\mu\alpha.\Gamma){\downarrow_S} & = & \mu\alpha.(\Gamma{\downarrow_S}) \\
\Gamma{\uparrow_S} & = & \Gamma{\downarrow_{\mathbf{Var}\setminus S}}
\end{array}
$$

$\Gamma{\downarrow_S}$ extracts from Γ information on the usage of only the channels in S, while $\Gamma{\uparrow_S}$ extracts information on the usage of the channels not in S.

Example 3.2.2: Let $\Gamma = y?^{\mathbf{t}}[\tau'].\,x!^{\mathbf{t}'}[\tau'].\,\mathbf{0}$. Then $\Gamma{\downarrow_{\{x\}}} = \mathbf{t}.x!^{\mathbf{t}'}[\tau].\,\mathbf{0}$ and $\Gamma{\uparrow_{\{x\}}} = y?^{\mathbf{t}}[\tau].\,\mathbf{t}'.\mathbf{0}$.

Definition 3.2.3 [subtyping]: A preorder $\leq$ on process types is a *proper subtyping relation* if it satisfies the rules given in Figure 3 ($\Gamma_1 \cong \Gamma_2$ denotes $\Gamma_1 \leq \Gamma_2 \wedge \Gamma_2 \leq \Gamma_1$).

In the rest of this paper, we assume that $\leq$ always denotes a proper subtyping relation. We extend $\leq$ to a subtyping relation on tuple types by: $\tau_1 \leq \tau_2$ if and only if there exist $\tilde{x}, \Gamma_1$, and Γ_2 such that $\tau_1 = (\tilde{x})\Gamma_1$, $\tau_2 = (\tilde{x})\Gamma_2$, and $\Gamma_1 \leq \Gamma_2$.

The axiom $\Gamma{\downarrow_S}\,|\,\Gamma{\uparrow_S} \leq \Gamma$ allows us to forget information on dependencies between some channels. For example, if $\Gamma = y?^{\mathbf{t}}[\tau'].\,x!^{\mathbf{t}'}[\tau'].\,\mathbf{0}$, then $\Gamma{\downarrow_{\{x\}}}\,|\,\Gamma{\uparrow_{\{x\}}} = \mathbf{t}.x!^{\mathbf{t}'}[\tau].\,\mathbf{0}\,|\,y?^{\mathbf{t}}[\tau].\,\mathbf{t}'.\mathbf{0}$ is a subtype of Γ. Notice that $\Gamma{\downarrow_{\{x\}}}\,|\,\Gamma{\uparrow_{\{x\}}}$ expresses a more liberal usage of x, y than Γ: While Γ means that x is used for output only after y is used for input, $\Gamma{\downarrow_{\{x\}}}\,|\,\Gamma{\uparrow_{\{x\}}}$ only says that x is used for output after *some* event $\mathbf{t}$, not necessarily an input from y.

3.3 Reduction of Process Types

We want to reason about the behavior of a process by inspecting the behavior of its abstraction, i.e., process type. We therefore define reduction of process types, so that each reduction step of a process is matched by a reduction step of its process type. For example, the reduction of a process

$$x?^{\mathbf{t}_1}[z].\,z!^{\mathbf{t}_2}[]\,|\,x!^{\mathbf{t}_2}[y] \longrightarrow y!^{\mathbf{t}_2}[]$$

is matched by:

$$x?^{\mathbf{t}_1}[\tau]\,|\,x!^{\mathbf{t}_3}[\tau].\,y!^{\mathbf{t}_2}[] \longrightarrow y!^{\mathbf{t}_2}[]$$

for $\tau = (z)z!^{\mathbf{t}_2}[]$. As is the case for reductions of processes, we annotate each reduction with information on which channel and events are involved in the reduction.

Definition 3.3.1: A reduction relation $\Gamma_1 \xrightarrow{L} \Gamma_2$ on process types (where $L \subseteq \mathbf{T} \cup \{x^{\mathbf{t}_1,\mathbf{t}_2} \mid (x \in \mathbf{Var}) \wedge (\mathbf{t}_1, \mathbf{t}_2 \in \mathbf{T})\}$) is the least relation closed under the rules in Figure 4.

We write $\Gamma \longrightarrow \Gamma'$ when $\Gamma \xrightarrow{L} \Gamma'$ for some L.

3.4 Consistency of Process Types

If a process type is a correct abstraction of a process, we can verify a property of the process by verifying the corresponding property of the process type. When a process type satisfies such a property, we say that the process type is *consistent*. The consistency condition depends on the property we require for processes. So, we state here only necessary conditions that every consistency condition should satisfy. Consistency conditions for specific instances are given in Section 4.

Definition 3.4.1 [well-formedness]: A process type Γ is well-formed, written $WF(\Gamma)$, if there exist no x, τ_1, τ_2, $\mathbf{t}_1$, $\mathbf{t}_2$, Γ_1, Γ_2, and Γ_3 that satisfy the following conditions:

1. $\Gamma \longrightarrow^* \cdots + x!^{\mathbf{t}_1}[\tau_1].\,\Gamma_1 + \cdots \,|\, \cdots + x?^{\mathbf{t}_2}[\tau_2].\,\Gamma_2 + \cdots \,|\,\Gamma_3$.
2. $\tau_1 \not\leq \tau_2$.

Remark 3.4.2: We could replace the first condition with "Γ contains $x!^{\mathbf{t}_1}[\tau_1]$ and $x?^{\mathbf{t}_2}[\tau_2]$," which can be checked more easily. We do not do so, however, to allow maximal flexibility of type systems. Under the above condition, we can allow process types like $x?^{\mathbf{t}_1}[\tau_1].\,x!^{\mathbf{t}_2}[\tau_2]\,|\,x!^{\mathbf{t}_3}[\tau_1].\,x?^{\mathbf{t}_4}[\tau_2]$, which allows x to be used for first communicating a value of type τ_1, and then for communicating a value of type τ_2.

Definition 3.4.3 [consistency]: A predicate *ok* on process types is a *proper consistency predicate* if it satisfies the following conditions:

$$\Gamma \mid \mathbf{0} \cong \Gamma \quad \text{(Sub-Empty)}$$

$$\Gamma_1 \mid \Gamma_2 \cong \Gamma_2 \mid \Gamma_1 \quad \text{(Sub-Commut)}$$

$$\Gamma_1 \mid (\Gamma_2 \mid \Gamma_3) \cong (\Gamma_1 \mid \Gamma_2) \mid \Gamma_3 \quad \text{(Sub-Assoc)}$$

$$\mu\alpha.\Gamma \cong [\mu\alpha.\Gamma/\alpha]\Gamma \quad \text{(Sub-Rec)}$$

$$\Gamma_1 \& \Gamma_2 \leq \Gamma_i \ (i \in \{1,2\}) \quad \text{(Sub-IChoice)}$$

$$\frac{\Gamma \leq \Gamma'}{*\Gamma \leq *\Gamma'} \quad \text{(Sub-Rep)}$$

$$\frac{\Gamma_1 \leq \Gamma_1' \qquad \Gamma_2 \leq \Gamma_2'}{\Gamma_1 \mid \Gamma_2 \leq \Gamma_1' \mid \Gamma_2'} \quad \text{(Sub-Par)}$$

$$\frac{\gamma_i \leq \gamma_i' \text{ for each } i \in \{1,\ldots,n\}}{\gamma_1 + \cdots + \gamma_n \leq \gamma_1' + \cdots + \gamma_n'} \quad \text{(Sub-Choice)}$$

$$\frac{\Gamma \leq \Gamma'}{\Gamma{\downarrow_S} \leq \Gamma'{\downarrow_S}} \quad \text{(Sub-Restrict)}$$

$$\frac{\Gamma \leq \Gamma'}{[y/x]\Gamma \leq [y/x]\Gamma'} \quad \text{(Sub-Subst)}$$

$$(\Gamma{\downarrow_S} \mid \Gamma{\uparrow_S}) \leq \Gamma \quad \text{(Sub-Divide)}$$

Figure 3: Necessary Conditions on Subtyping Relation

$$\frac{\tau_1 \leq \tau_2}{\cdots + x!^{\mathbf{t}_1}[\tau_1].\,\Gamma_1 + \cdots \mid \cdots + x?^{\mathbf{t}_2}[\tau_2].\,\Gamma_2 + \cdots \xrightarrow{\{x^{\mathbf{t}_1,\mathbf{t}_2}\}} \Gamma_1 \mid \Gamma_2} \quad \text{(TER-Com)}$$

$$\Gamma \xrightarrow{\emptyset} \Gamma \quad \text{(TER-Skip)}$$

$$\frac{\Gamma_1 \xrightarrow{L_1} \Gamma_1' \qquad \Gamma_2 \xrightarrow{L_2} \Gamma_2'}{\Gamma_1 \mid \Gamma_2 \xrightarrow{L_1 \cup L_2} \Gamma_1' \mid \Gamma_2'} \quad \text{(TER-Par)}$$

$$\cdots + \mathbf{t}.\Gamma + \cdots \xrightarrow{\{\mathbf{t}\}} \Gamma \quad \text{(TER-Ev)}$$

$$\frac{\Gamma_1 \leq \Gamma_1' \qquad \Gamma_1' \xrightarrow{L} \Gamma_2' \qquad \Gamma_2' \leq \Gamma_2}{\Gamma_1 \xrightarrow{L} \Gamma_2} \quad \text{(TER-Sub)}$$

Figure 4: Reduction of Process Types

1. If $ok(\Gamma)$, then $WF(\Gamma)$.
2. If $ok(\Gamma)$ and $\Gamma \longrightarrow \Gamma'$, then $ok(\Gamma')$.
3. If $ok(\Gamma_1)$ and $Null(\Gamma_2)$, then $ok(\Gamma_1 \mid \Gamma_2)$

In the rest of this paper, we assume that ok always refers to a proper consistency predicate.

Because process types form a much simpler process calculus than the π-calculus, we expect that the predicate ok is normally much easier to verify than the corresponding property of a process. The actual procedure to verify ok, however, depends on the definition of the subtyping relation $\leq$: If we are not interested in linearity information [17], we can introduce the rule $\Gamma \mid \Gamma \cong \Gamma$ so that the reductions of a process type can be reduced to a finite-state machine. But if we take $\leq$ to be the least proper subtyping relation, we need to use a more complex system like Petri nets, as is the case for our previous type system for deadlock-freedom [18].

3.5 Typing

A type judgment is of the form $\Gamma \triangleright P$, where Γ is a closed (i.e., containing no free type variables) process type. It means that P behaves as specified by Γ.

Typing rules are given in Figure 5. The rules (T-Par), (T-Choice), and (T-Rep) say that an abstraction of a process constructed by using a process constructor $\mid, +$, or $*$ can be obtained by composing abstractions of its subprocesses with the corresponding constructor of process types.

The key rules are (T-Out), (T-In), and (T-New). Note that channels can be dynamically created and passed through other channels in the process calculus, while in the calculus of process types, there are no corresponding mechanisms. So, we must somehow approximate the behavior of a process in those rules.

In the rule (T-Out), we cannot express information that $[\tilde{z}]$ is passed through x at the type level. Instead, we put $[\tilde{z}/\tilde{y}]\Gamma_2$, which expresses how the channels $\tilde{z}$ are used by a receiver, into the continuation of the output action.

In the rule (T-In), information on how received channels $\tilde{y}$ are used is put into the tuple type $(\tilde{y})(\Gamma{\downarrow_{\{\tilde{y}\}}})$. Because we want to keep only information on the usage of $\tilde{y}$, we apply $\cdot{\downarrow_{\{\tilde{y}\}}}$ to remove information on the usage of the other variables. Information on the usage of the other variables is kept in the continuation $\Gamma{\uparrow_{\{y_1,\ldots,y_n\}}}$ of the input action. For example, consider a process $x?^{\mathbf{t}_1}[y].\,y?^{\mathbf{t}_2}[].\,z!^{\mathbf{t}_3}[]$. Its subprocess $y?^{\mathbf{t}_2}[].\,z!^{\mathbf{t}_3}[]$ is typed under the process type $y?^{\mathbf{t}_2}[].\,z!^{\mathbf{t}_3}[].\,\mathbf{0}$. By applying (T-In), we obtain the following type judgment:

$$x?^{\mathbf{t}_1}[(y)y?^{\mathbf{t}_2}[].\,\mathbf{t}_3.\mathbf{0}].\,\mathbf{t}_2.z!^{\mathbf{t}_3}[].\,\mathbf{0} \triangleright x?^{\mathbf{t}_1}[y].\,y?^{\mathbf{t}_2}[].\,z!^{\mathbf{t}_3}[]$$

Notice that the parameter type $(y)y?^{\mathbf{t}_2}[].\,\mathbf{t}_3.\mathbf{0}$ of the channel x carries only information that *some* event $\mathbf{t}_3$ occurs after y is used for input, not that z is used for output. On the other hand, the continuation part $\mathbf{t}_2.z!^{\mathbf{t}_3}[].\,\mathbf{0}$ says just that z is used for output only after some event $\mathbf{t}_2$ occurs.

In the rule (T-New), we check by the condition $ok(\Gamma{\downarrow_{\{\tilde{x}\}}})$ that $\tilde{x}$ are used in a consistent manner, and forget information on the use of $\tilde{x}$ by $\cdot{\uparrow_{\{\tilde{x}\}}}$.

3.6 Properties of the Type System

The general type system given above is parameterized by the subtyping relation $\leq$ and the consistency predicate ok,

$$\mathbf{0} \triangleright \mathbf{0} \quad \text{(T-Zero)}$$

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

$$\frac{\Gamma \triangleright P}{*\Gamma \triangleright *P} \quad \text{(T-Rep)}$$

$$\frac{\Gamma' \triangleright P \qquad \Gamma \leq \Gamma'}{\Gamma \triangleright P} \quad \text{(T-Sub)}$$

$$\frac{\gamma_i \triangleright G_i \text{ for each } i \in \{1,\ldots,n\}}{\gamma_1 + \cdots + \gamma_n \triangleright G_1 + \cdots + G_n} \quad \text{(T-Choice)}$$

$$\frac{\Gamma_1 \triangleright P}{x!^{\mathbf{t}}[(\tilde{y})\Gamma_2].\,(\Gamma_1 \mid [\tilde{z}/\tilde{y}]\Gamma_2) \triangleright x!^{\mathbf{t}}[\tilde{z}].\,P} \quad \text{(T-Out)}$$

$$\frac{\Gamma \triangleright P}{x?^{\mathbf{t}}[(\tilde{y})(\Gamma{\downarrow_{\{\tilde{y}\}}})].\,(\Gamma{\uparrow_{\{\tilde{y}\}}}) \triangleright x?^{\mathbf{t}}[\tilde{y}].\,P} \quad \text{(T-In)}$$

$$\frac{\Gamma \triangleright P \qquad ok(\Gamma{\downarrow_{\{\tilde{x}\}}})}{\Gamma{\uparrow_{\{\tilde{x}\}}} \triangleright (\nu\tilde{x})\,P} \quad \text{(T-New)}$$

Figure 5: Typing Rules

which determine the exact properties of each instance of the type system. As we show below, however, several important properties can be proved independently of a choice of the subtyping relation and the consistency predicate. In particular, we can prove that if $\Gamma \triangleright P$ holds, Γ is a correct abstraction of P in the sense that P satisfies a certain invariant property if Γ satisfies the corresponding property (Theorem 3.6.2).

Type Preservation

We define a mapping $l^{\#}$ from labels of process reductions to labels of type reductions by: $(x^{\mathbf{t}_1,\mathbf{t}_2})^{\#} = \{x^{\mathbf{t}_1,\mathbf{t}_2}\}$ and $(\epsilon^{\mathbf{t}_1,\mathbf{t}_2})^{\#} = \{\mathbf{t}_1, \mathbf{t}_2\}$. The following theorem guarantees that if $\Gamma \triangleright P$ holds, for every reduction of P, there is a corresponding reduction of Γ.

Theorem 3.6.1 [subject reduction]: If $\Gamma \triangleright P$ and $P \xrightarrow{l} Q$ with $WF(\Gamma)$, then there exists Γ' such that $\Gamma \xrightarrow{l^{\#}} \Gamma'$ and $\Gamma' \triangleright Q$.

As a corollary, it follows that a process satisfies a certain invariant condition p if the process type of P satisfies the corresponding consistency condition.

Theorem 3.6.2: Suppose $p(P)$ holds for any Γ such that $\Gamma \triangleright P$ and $ok(\Gamma)$. If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p(Q)$ holds for every Q such that $P \longrightarrow^* Q$.

Proof: By mathematical induction on the length of the reduction sequence $P \longrightarrow \cdots \longrightarrow Q$, using Theorem 3.6.1 and the fact that ok is preserved by reduction. □

Normalization of Type Derivation

The normal derivation theorem given below is useful for studying a relationship between a process and its process type, and also for developing type-check/reconstruction algorithms. We write $\Gamma \triangleright_N P$ if $\Gamma \triangleright P$ is derivable by using (T-Sub) only immediately before (T-In) or (T-Out).

Theorem 3.6.3 [normal derivation]: If $\Gamma \triangleright P$, then $\Gamma' \triangleright_N P$ for some Γ' such that $\Gamma \leq \Gamma'$.

Proof: This follows from the fact that each application of the rule (T-Sub) above a rule except for (T-In) and (T-Out) can be permuted downwards. □

As a corollary, it follows that if a process is trying to perform an input action, its process type is also trying to perform the corresponding action. (A similar property holds also for output.)

Corollary 3.6.4: If $\Gamma \triangleright (\widetilde{\nu x}_{1..k})\,(\cdots + y?^{\mathbf{t}}[\tilde{z}].\,P + \cdots \mid Q)$ and $ok(\Gamma)$, then the following conditions hold.

1. If $y \notin \{\tilde{x_1}, \ldots, \tilde{x_k}\}$, then $\Gamma \leq \cdots + y?^{\mathbf{t}}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2$ for some τ, Γ_1, and Γ_2.
2. If $y \in \{\tilde{x_1}, \ldots, \tilde{x_k}\}$, then $\Gamma \leq \cdots + \mathbf{t}.\Gamma_1 + \cdots \mid \Gamma_2$ for some Γ_1 and Γ_2.

Conversely, if a process type obtained by normal derivation is trying to perform some action, the process is also trying to perform the corresponding action.

Theorem 3.6.5:

1. If $\cdots + \mathbf{t}.\Gamma_1 + \cdots \mid \Gamma_2 \triangleright_N P$, then $P \preceq (\widetilde{\nu x}_{1..k})\,(y?^{\mathbf{t}}[\tilde{z}].\,Q \mid R)$ or $P \preceq (\widetilde{\nu x}_{1..k})\,(y!^{\mathbf{t}}[\tilde{z}].\,Q \mid R)$, with $y \in \{\tilde{x_1}, \ldots, \tilde{x_k}\}$.
2. If $\cdots + y?^{\mathbf{t}}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2 \triangleright_N P$, then $P \preceq (\widetilde{\nu x}_{1..k})\,(y?^{\mathbf{t}}[\tilde{z}].\,Q \mid R)$ with $y \notin \{\tilde{x_1}, \ldots, \tilde{x_k}\}$.

Proof: Trivial by the definition of $\Gamma \triangleright_N P$. □

Type Check/Reconstruction

By using Theorem 3.6.3, we can also formalize a common part of type-check/reconstruction algorithms: By reading the typing rules in a bottom-up manner, we can develop an algorithm that inputs a process expression and outputs a set of subtype constraints and consistency conditions on process types (see Appendix A). A process is typable if and only if the set of constraints output by the algorithm is satisfiable. Thus, to develop a type-check/reconstruction algorithm for each instance of our type system, it suffices to develop an algorithm to solve constraints on process types. Such algorithms have already been developed for specific type systems [12, 18].

4 Applications

We show that a variety of type systems — those for arity-mismatch check, race detection, deadlock detection, and static garbage-channel collection — can indeed be obtained

as instances of the generic type system. Thanks to the common properties in Section 3.6, only a small amount of extra work is necessary to define each instance and prove its correctness.

The invariant properties of well-typed processes that the type systems should guarantee are shown in Table 1. The condition $p_1(P)$ means that no arity-mismatch error occurs immediately. (So, if p_1 is an invariant condition, no arity-mismatch error occurs during reduction of P.) $p_2(P)$ means that P is not in a race condition on any output actions annotated with $\mathbf{t}$. $p_3(P)$ means that P is not deadlocked on any actions annotated with $\mathbf{t}$ in the sense that whenever P is trying to perform an action annotated with $\mathbf{t}$, P can be further reduced. $p_4(P)$ means that after a channel has been used for an input action annotated with $\mathbf{t}$, it is no longer used. So, it is safe to deallocate the channel after the action annotated with $\mathbf{t}$. For example, the process $(\nu x)(x!^{\mathbf{t}_1}[] \mid x!^{\mathbf{t}_2}[] \mid x?^{\mathbf{t}_3}[].x?^{\mathbf{t}}[].\mathbf{0})$ satisfies this property.

Table 2 shows the consistency condition for each type system. $ok_2(\Gamma)$ means that no race occurs on output actions annotated with $\mathbf{t}$ during reductions of the abstract process Γ. $ok_3(\Gamma)$ means that whenever Γ is reduced to a process type trying to perform an action annotated with an event $\mathbf{t}'$ less than or equal to $\mathbf{t}$, Γ can be further reduced on some channel or on an event less than $\mathbf{t}'$. (Here, we assume that $\prec$ is some well-founded relation on events, and $\mathbf{t} \preccurlyeq \mathbf{t}'$ means $\mathbf{t} \prec \mathbf{t}'$ or $\mathbf{t} = \mathbf{t}'$.) $ok_4(\Gamma)$ means that, after Γ has been reduced on an action involving a channel x and the event $\mathbf{t}$, the reduced process type no longer performs an input or output action on the same channel.

Let $\leq$ be the least proper subtyping relation. We can prove the following type-soundness theorem for all of the above type systems. By Theorem 3.6.2, it suffices to show that $\Gamma \triangleright P$ and $ok_i(\Gamma)$ imply $p_i(P)$ for each i, by using Theorems 3.6.3–3.6.5 (see Appendix B.2).

Theorem 4.1: Let ok be ok_i ($i \in \{1,2,3,4\}$). If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_i(Q)$ holds for every Q such that $P \longrightarrow^* Q$.

Actually, $\leq$ can be any proper subtyping relation except for the case $i = 3$. Choosing an appropriate subtyping relation for each type system would simplify type-checking or type-reconstruction. For example, in the above type systems, we can identify $\mathbf{t}.\Gamma$ with Γ by the rule $\mathbf{t}.\Gamma \cong \Gamma$, except for the case $i = 3$. For a naive arity-mismatch check [7, 37], we can ignore the order of communications by introducing rules like $x?^{\mathbf{t}}[\tau].\Gamma \cong x?^{\mathbf{t}}[\tau] \mid \Gamma$.

The following examples indicate that our framework not only subsumes many of the existing type systems but also provides more powerful type systems than them (see also Section 5).

Example 4.2: The process $x?[y].x?[].\mathbf{0} \mid x![w].x![]$ is well typed in the first ($i = 1$) type system. So, unlike in earlier type systems [7, 37] for arity-mismatch check, the same channel can be used for communicating values of different types.[5]

Example 4.3: The second type system guarantees that the process $(\nu l, x)(l!^{\mathbf{t}_0}[] \mid *l?^{\mathbf{t}_1}[].x!^{\mathbf{t}}[].l!^{\mathbf{t}_3}[] \mid x?^{\mathbf{t}_4}[].\mathbf{0})$ is race-free on the channel x. So, unlike the linear π-calculus [17], our type system can guarantee lack of race conditions even on channels that are used more than once.[6]

[5] Yoshida's type system [38] also allows such use of channels.

[6] Flanagan and Abadi's type system [4] also gives such a guarantee. Because their target calculus has locks as primitives, the problem is a little simpler.

Example 4.4: The third type system rejects the process $P = (\nu x)(\nu y)(x?^{\mathbf{t}}[].y!^{\mathbf{t}_1}[] \mid y?^{\mathbf{t}_2}[].x!^{\mathbf{t}_3}[])$. The type of the sub-process $x?^{\mathbf{t}}[].y!^{\mathbf{t}_1}[] \mid y?^{\mathbf{t}_2}[].x!^{\mathbf{t}_3}[]$ is $x?^{\mathbf{t}}[].y!^{\mathbf{t}_1}[] \mid y?^{\mathbf{t}_2}[].x!^{\mathbf{t}_3}[]$. So, in order for P to be well-typed, the following constraints must be satisfied:

$$ok_3(x?^{\mathbf{t}}[].\mathbf{t}_1.\mathbf{0} \mid \mathbf{t}_2.x!^{\mathbf{t}_3}[])$$
$$ok_3(\mathbf{t}.y!^{\mathbf{t}_1}[] \mid y?^{\mathbf{t}_2}[].\mathbf{t}_3.\mathbf{0})$$

The former constraint requires that $\mathbf{t}_2 \prec \mathbf{t}$ (because the input from x succeeds only after the event $\mathbf{t}_2$ succeeds), while the latter requires that $\mathbf{t} \prec \mathbf{t}_2$, hence a contradiction.

Remark 4.5: A type environment in a usual type system corresponds to the equivalence class of a process type with respect to the relation $\cong$ derived from an appropriate subtyping relation. (Recall that $\Gamma_1 \cong \Gamma_2$ is defined as $\Gamma_1 \leq \Gamma_2 \wedge \Gamma_2 \leq \Gamma_1$.) For example, the type environment

$$x : [[]/O]/(O|I), z : []/(O|I.I)$$

given in Section 1 corresponds to the equivalence class of a process type

$$x!^{\mathbf{t}}[(y)y!^{\mathbf{t}}[]] \mid x?^{\mathbf{t}}[(y)y!^{\mathbf{t}}[]] \mid z!^{\mathbf{t}}[] \mid z?^{\mathbf{t}}[].z?^{\mathbf{t}}[],$$

with respect to $\cong$ that satisfy the rules $x!^{\mathbf{t}}[\tau].\Gamma \cong x!^{\mathbf{t}'}[\tau].\Gamma$, $x?^{\mathbf{t}}[\tau].\Gamma \cong x?^{\mathbf{t}'}[\tau].\Gamma$, $\mathbf{t}.\Gamma \cong \Gamma$, and $(\Gamma\downarrow_S \mid \Gamma\uparrow_S) \cong \Gamma$. The last rule removes information on the order of communications between different channels. A type environment of the linear π-calculus [17] is obtained by further removing information on channel usage, by adding the rules $x!^{\mathbf{t}}[\tau].\Gamma \cong x!^{\mathbf{t}}[\tau] \mid \Gamma$, $x?^{\mathbf{t}}[\tau].\Gamma \cong x?^{\mathbf{t}}[\tau] \mid \Gamma$, and $\Gamma \mid \Gamma \cong *\Gamma$. A type environment of the input-only/output-only channel type system [27] is obtained by further adding the rule $*\Gamma \cong \Gamma$.

5 Further Applications: Analysis of Race and Deadlock of Concurrent Objects

The type systems for race- and deadlock-freedom, presented in the last section, are indeed powerful enough to guarantee some useful properties about concurrent objects. In essence, a concurrent object is regarded as a set of processes that provides a collection of services (e.g., methods) [13, 19, 29], just as a sequential object is a set of functions. Clients refer to an object through a record of channels that represent locations of those services. Hence, by giving an appropriate type to the record, we can enforce a certain protocol that clients should respect. Since our type system can capture, in particular, temporal dependency on the utilized services, it is possible to guarantee race-freedom of accesses to methods, studied by Abadi, Flanagan and Freund [4, 6], and deadlock-freedom for objects with non-uniform service availability, studied by Puntigam [30]. Note that, so far, these properties have been discussed only for languages with primitive notion of objects. This section demonstrates how our type system can guarantee these properties.

We first describe race-free accesses to methods. For example, the following process waits for a request on *newob*, and upon receiving a request, creates an object with a lock l, a method m to print out the string "Hello, " appended to a given string, and a channel r to receive a reply from the method, and exports its interface $[l, m, r]$ through the reply channel r'.

$$*newob?[r'].(\nu l, m, r)($$
$$\quad l!^{\mathbf{t}'}[] \mid *m?^{\mathbf{t}'}[s].print!^{\mathbf{t}'}[\text{"Hello, "}].print!^{\mathbf{t}'}[s].r!^{\mathbf{t}'}[]$$
$$\quad \mid r'![l, m, r])$$

$p_1(P)$	There exist no $\tilde{x_1}, \ldots, \tilde{x_k}$, x, $\tilde{z}$, $\tilde{w}$, $\mathbf{t}$, $\mathbf{t}'$, Q_1, Q_2, and Q_3 such that $P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x!^{\mathbf{t}}[\tilde{z}].\,Q_1 + \cdots \mid \cdots + x?^{\mathbf{t}'}[\tilde{w}].\,Q_2 + \cdots \mid Q_3)$ with $\|\tilde{z}\| \neq \|\tilde{w}\|$.
$p_2(P)$	There exist no x, $\tilde{x_1}, \ldots, \tilde{x_n}$, $\tilde{z}$, $\tilde{w}$, $\mathbf{t}'$, Q_1, Q_2, Q_3 such that $P \preceq (\widetilde{\nu x}_{1..n})\,(\cdots + x!^{\mathbf{t}}[\tilde{z}].\,Q_1 + \cdots \mid \cdots + x!^{\mathbf{t}'}[\tilde{w}].\,Q_2 + \cdots \mid Q_3)$.
$p_3(P)$	If there exist $\tilde{x_1}, \ldots, \tilde{x_n}$, y, $\tilde{z}$, Q, R such that $P \preceq (\widetilde{\nu x}_{1..n})\,(\cdots + y?^{\mathbf{t}}[\tilde{z}].\,Q + \cdots \mid R)$ or $P \preceq (\widetilde{\nu x}_{1..n})\,(\cdots + y!^{\mathbf{t}}[\tilde{z}].\,Q + \cdots \mid R)$, then $P \longrightarrow P'$ for some P'.
$p_4(P)$	For any $\mathbf{t}'$, x, $\tilde{w_1}, \ldots, \tilde{w_n}$, P', and Q, if $P \longrightarrow^* \preceq (\widetilde{\nu w}_{1..n})\,P'$ and $P' \xrightarrow{x^{\mathbf{t}',\mathbf{t}}} \longrightarrow^* Q$, then there exist no Q_1, Q_2 such that $Q \preceq (\widetilde{\nu u})\,(\cdots + x?[\tilde{y}].\,Q_1 + \cdots \mid Q_2)$ or $Q \preceq (\widetilde{\nu u})\,(\cdots + x![\tilde{y}].\,Q_1 + \cdots \mid Q_2)$.

Table 1: Properties of Processes

$ok_1(\Gamma)$	$WF(\Gamma)$
$ok_2(\Gamma)$	$WF(\Gamma)$, and there exist no x, $\mathbf{t}'$, τ_1, τ_2, Γ_1, Γ_2, Γ_3 such that $\Gamma \longrightarrow^* \cdots + x!^{\mathbf{t}}[\tau_1].\,\Gamma_1 + \cdots \mid \cdots + x!^{\mathbf{t}'}[\tau_2].\,\Gamma_2 + \cdots \mid \Gamma_3$.
$ok_3(\Gamma)$	$WF(\Gamma)$, and if $\mathbf{t}' \preccurlyeq \mathbf{t}$, $\Gamma \longrightarrow^* \Gamma'$ and Γ' is $\cdots + y?^{\mathbf{t}'}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2$ or $\cdots + y!^{\mathbf{t}'}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2$ for some y, τ, $\mathbf{t}'$, Γ_1, Γ_2, then $\Gamma' \xrightarrow{L} \Gamma''$ for some L and Γ'' such that (i) $L = \{x^{\mathbf{t}_1,\mathbf{t}_2}\}$ or (ii) $L = \{\mathbf{t}''\}$ and $\mathbf{t}'' \prec \mathbf{t}'$.
$ok_4(\Gamma)$	$WF(\Gamma)$, and for any $x, \mathbf{t}'$, and Γ', if $\Gamma \longrightarrow^* \xrightarrow{x^{\mathbf{t}',\mathbf{t}}} \Gamma'$, then there exist no τ, $\mathbf{t}''$, Γ_1, Γ_2 such that $\Gamma' \longrightarrow^* \cdots + x?^{\mathbf{t}''}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2$ or $\Gamma' \longrightarrow^* \cdots + x!^{\mathbf{t}''}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2$.

Table 2: Consistency Conditions

Since the method m should not be invoked simultaneously,[7] clients should acquire and release the lock l before and after the invocation of m, respectively. Indeed by using ok_2, it is guaranteed that there are no simultaneous outputs to m: The exported interface $[l, m, r]$ (through which clients access to the object) can be given a type:

$$(l, m, r) \star l?^{\mathbf{t}'}[\,].\,m!^{\mathbf{t}}[\mathit{String}].\,r?^{\mathbf{t}'}[\,].\,l!^{\mathbf{t}'}[\,],$$

where $\star\Gamma$ abbreviates $\mu\alpha.(\mathbf{0}\&(\Gamma \mid \alpha))$, meaning that the tuple can be used according to Γ by arbitrarily many processes. (Here, we assume that the subtyping relation $\leq$ satisfies $\mathbf{t}.\Gamma \cong \Gamma$.) Then, a client

$$\begin{array}{l} l?^{\mathbf{t}'}[\,].\,m!^{\mathbf{t}}[\text{"Atsushi"}].\,r?^{\mathbf{t}'}[\,].\,l!^{\mathbf{t}'}[\,] \\ \mid l?^{\mathbf{t}'}[\,].\,m!^{\mathbf{t}}[\text{"Naoki"}].\,r?^{\mathbf{t}'}[\,].\,l!^{\mathbf{t}'}[\,] \end{array}$$

is well typed, but $m!^{\mathbf{t}}[\text{"Atsushi"}].\,r?^{\mathbf{t}'}[\,].\,\cdots$ is not. The above type of the interface roughly corresponds to the object type $[m : \varsigma(l)\mathit{String} \to \mathit{Unit} \cdot \{l\} \cdot +]$ of Abadi and Flanagan's type system [4], which means that the method m can be invoked only after the lock on the object is acquired.

Similarly, we can express non-uniform service availability in our type system. For example, this is a process that creates a one-place buffer:

$$\begin{array}{l} *\mathit{newbuf}?\,[r].\,(\nu \mathit{put}, \mathit{get}, b)\,(\\ \quad b![\,] \mid *b?[\,].\,\mathit{put}?[x].\,\mathit{get}?[r'].\,(r'![x] \mid b![\,]) \\ \quad \mid r![\mathit{put}, \mathit{get}]) \end{array}$$

Now, two methods put and get are provided but they are available only alternately. By using ok_3, we can guarantee that invocations of the methods put and get never get deadlocked. The interface $[\mathit{put}, \mathit{get}]$ can be given a type:

$$(\mathit{put}, \mathit{get})\mu\alpha.(\mathbf{0}\&(\mathit{put}!^{\mathbf{t}}[\tau] \mid \infty.\mathit{get}!^{\mathbf{t}}[(r)r!^{\mathbf{t}}[\tau]].\,\infty.\alpha)),$$

which says that an output to put must come in parallel to or before an output to get. (Here, $\infty.\Gamma$ abbreviates $\mu\alpha.(\Gamma\&\mathbf{t}_1.\alpha\&\cdots\&\mathbf{t}_n.\alpha)$ where $\{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$ is the set of events occurring in the program. It means that it is allowed to wait for arbitrary events before using the value according to Γ.) Then, both

$$\mathit{put}!^{\mathbf{t}}[v] \mid (\nu r)\,\mathit{get}!^{\mathbf{t}}[r].\,r?[x].\,\cdots$$

and

$$\mathit{put}!^{\mathbf{t}}[v].\,(\nu r)\,\mathit{get}!^{\mathbf{t}}[r].\,r?[x].\,\cdots$$

are well-typed (and hence never get deadlocked on $\mathit{put}!$ and $\mathit{get}!$) while $(\nu r')\,\mathit{get}!^{\mathbf{t}}[r'].\,r'?[x].\,\mathit{put}!^{\mathbf{t}}[v].\,\cdots$ is not.

[7] We do not want an output like "Hello Hello ".

6 Towards a General Type Soundness Theorem

In Section 3.6, we have shown that a process satisfies a certain property p if its process type satisfies the *corresponding* consistency condition ok. However, it was left to the designer of a specific type system to find a consistency condition that *corresponds to* a process property of interest and prove that the correspondence is indeed correct. In fact, in Section 4, we had to find a suitable consistency condition on process types and prove its correctness (Theorem 4.1) for each type system. Also, there remains a general question about the power of our generic type system: What kind of type system can be obtained as an instance? Clearly, not all properties can be verified in our type system: For example, the property "a process can create at most n channels" cannot be verified, because process types contain no information on channel creation. This section gives a partial answer to those questions: For a certain class of properties of processes, there is indeed a systematic way for obtaining the corresponding consistency condition ok on process types, so that the instantiated type system is sound. For lack of space, details are omitted: They are found in the full paper [11].

We first introduce logical formulas [1, 32] to formally state properties of processes and types.

Definition 6.1: The set **Prop** of formulas is given by the following syntax (i, j denote variables ranging over the set

Nat of non-negative integers and n denotes a non-negative integer or a variable ranging over **Nat**):

$$\begin{array}{rcl} A & ::= & \mathbf{tt} \mid x!^{\mathbf{t}}n \mid x?^{\mathbf{t}}n \mid (A \mid B) \mid \langle l \rangle A \mid ev(A) \\ & & \mid \neg A \mid A \vee B \mid \exists x.A \mid \exists \mathbf{t}.A \mid \exists \zeta : C.A \\ C & ::= & \{m_1 \neq n_1, \ldots, m_k \neq n_k\} \end{array}$$

A formula A describes a property of both processes and types. Intuitively, $x!^{\mathbf{t}}n$ means that some sub-process is ready to output an n-tuple on the channel x and that the output is tagged with $\mathbf{t}$. Similarly, $x?^{\mathbf{t}}n$ means that some sub-process is ready to input an n-tuple. The formula $A \mid B$ means that the process is parallel composition of a process satisfying A and another process satisfying B. $\langle l \rangle A$ means that the process can be reduced in one step to a process satisfying A and that the reduction is labelled with l. $ev(A)$ means that the process can be reduced to a process satisfying A in a finite number of steps.[8]

As a property of processes, the formal semantics $[\![A]\!]_{\mathbf{pr}}$ of each formula (i.e., the set of processes satisfying the formula) is defined by (**Proc** is the set of processes and $FV(A)$ is the set of free variables in A):[9]

$$\begin{array}{rl} [\![\mathbf{tt}]\!]_{\mathbf{pr}} & = \mathbf{Proc} \\ [\![x!^{\mathbf{t}}n]\!]_{\mathbf{pr}} & = \{P \mid P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x!^{\mathbf{t}}[\tilde{y}].\,Q + \cdots \mid R), \\ & \quad x \notin \{\tilde{x_1}, \ldots, \tilde{x_k}\}, \|\tilde{y}\| = n\} \\ [\![x?^{\mathbf{t}}n]\!]_{\mathbf{pr}} & = \{P \mid P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x?^{\mathbf{t}}[\tilde{y}].\,Q + \cdots \mid R), \\ & \quad x \notin \{\tilde{x_1}, \ldots, \tilde{x_k}\}, \|\tilde{y}\| = n\} \\ [\![A \mid B]\!]_{\mathbf{pr}} & = \{P \mid P \preceq (\widetilde{\nu x})\,(Q \mid R), \\ & \quad (\widetilde{\nu x})\,Q \in [\![A]\!]_{\mathbf{pr}}, (\widetilde{\nu x})\,R \in [\![B]\!]_{\mathbf{pr}}\} \\ [\![\langle l \rangle A]\!]_{\mathbf{pr}} & = \{P \mid P \xrightarrow{l} Q, Q \in [\![A]\!]_{\mathbf{pr}}\} \\ [\![ev(A)]\!]_{\mathbf{pr}} & = \{P \mid P \longrightarrow^{*} Q, Q \in [\![A]\!]_{\mathbf{pr}}\} \\ [\![\neg A]\!]_{\mathbf{pr}} & = \mathbf{Proc} \backslash [\![A]\!]_{\mathbf{pr}} \\ & \cdots \end{array}$$

Similarly, a formula can be regarded also as a property of process types (**Type** is the set of process types):

$$\begin{array}{rl} [\![\mathbf{tt}]\!]_{\mathbf{ty}} & = \mathbf{Type} \\ [\![x!^{\mathbf{t}}n]\!]_{\mathbf{ty}} & = \{\Gamma \mid \Gamma \leq \cdots + x!^{\mathbf{t}}[\tau].\,\Delta_1 + \cdots \mid \Delta_2, \|\tau\| = n\} \\ [\![x?^{\mathbf{t}}n]\!]_{\mathbf{ty}} & = \{\Gamma \mid \Gamma \leq \cdots + x?^{\mathbf{t}}[\tau].\,\Delta_1 + \cdots \mid \Delta_2, \|\tau\| = n\} \\ [\![A \mid B]\!]_{\mathbf{ty}} & = \{\Gamma \mid \Gamma \leq (\Delta_1 \mid \Delta_2), \Delta_1 \in [\![A]\!]_{\mathbf{ty}}, \Delta_2 \in [\![B]\!]_{\mathbf{ty}}\} \\ [\![\langle l \rangle A]\!]_{\mathbf{ty}} & = \{\Gamma \mid \Gamma \xrightarrow{l^{\#}} \Delta, \Delta \in [\![A]\!]_{\mathbf{ty}}\} \\ [\![ev(A)]\!]_{\mathbf{ty}} & = \{\Gamma \mid \Gamma \longrightarrow^{*} \Delta, \Delta \in [\![A]\!]_{\mathbf{ty}}\} \\ [\![\neg A]\!]_{\mathbf{ty}} & = \mathbf{Type} \backslash [\![A]\!]_{\mathbf{ty}} \\ & \cdots \end{array}$$

We can show that for any negative formula A defined below, a process P satisfies A (i.e., $P \in [\![A]\!]_{\mathbf{pr}}$) if its process type Γ satisfies the same formula A. Our type system is therefore sound at least for properties described using negative formulas.

Definition 6.2 [positive/negative formulas]: The set $\mathcal{F}^+$ ($\mathcal{F}^-$, resp.) of positive (negative, resp.) formulas are the least set satisfying the following rules:

$$\begin{array}{l} \mathbf{tt} \in \mathcal{F}^+ \cap \mathcal{F}^- \\ x!^{\mathbf{t}}n, x?^{\mathbf{t}}n \in \mathcal{F}^+ \\ A, B \in \mathcal{F}^+ \Rightarrow \\ \qquad A \mid B, A \vee B, \langle l \rangle A, ev(A), \exists x.A, \exists \mathbf{t}.A, \exists i : C.A \in \mathcal{F}^+ \\ A, B \in \mathcal{F}^- \Rightarrow A \vee B, \exists x.A, \exists \mathbf{t}.A, \exists i : C.A \in \mathcal{F}^- \\ A \in \mathcal{F}^+ \Rightarrow \neg A \in \mathcal{F}^- \\ A \in \mathcal{F}^- \Rightarrow \neg A \in \mathcal{F}^+ \end{array}$$

Theorem 6.3: Suppose $\Gamma \triangleright P$ and $WF(\Gamma)$. If $A \in \mathcal{F}^-$ and $\Gamma \in [\![A]\!]_{\mathbf{ty}}$, then $P \in [\![A]\!]_{\mathbf{pr}}$ holds. Conversely, if $A \in \mathcal{F}^+$ and $P \in [\![A]\!]_{\mathbf{pr}}$, then $\Gamma \in [\![A]\!]_{\mathbf{ty}}$ holds.

Proof sketch: This follows by induction on the structure of A. The cases for $\langle l \rangle A$ and $ev(A)$ follow from Theorem 3.6.1 and the cases for $x!^{\mathbf{t}}n$, $x?^{\mathbf{t}}n$, and $A \mid B$ follow from Theorem 3.6.3 and Corollary 3.6.4. The other cases follow immediately from the definitions of $[\![A]\!]_{\mathbf{pr}}$ and $[\![A]\!]_{\mathbf{ty}}$. □

Corollary 6.4 [type soundness]: Let $A \in \mathcal{F}^-$ and P be a closed process. Suppose that $ok(\Gamma)$ implies $\Gamma \in [\![A]\!]_{\mathbf{ty}}$. Suppose also that $\Gamma \triangleright P$ and $ok(\Gamma)$. Then, if $P \longrightarrow^{*} (\widetilde{\nu x})\,(\nu \tilde{y})\,Q$, then $(\widetilde{\nu x})\,Q \in [\![A]\!]_{\mathbf{pr}}$.

Intuitively, the last sentence of the above corollary means that all the channels created during reductions of P are used according to A. The corollary implies that, in order to guarantee that property, it suffices to define the consistency condition ok by $ok(\Gamma) \iff WF(\Gamma) \wedge inv(A)$ (where $inv(A) = \neg ev(\neg A)$).

The type systems for lack of arity mismatch, race detection, garbage-channel collection discussed in Section 4 can be automatically obtained by using the above corollary: For example, in the case of arity-mismatch check, we can let A be $\neg ev(\exists x.\exists \mathbf{t}, \mathbf{t}'.\exists i, j : \{i \neq j\}.(x!^{\mathbf{t}}i \mid x?^{\mathbf{t}'}j))$. In the case of race detection, we can let A be $\neg ev(\exists x.\exists \mathbf{t}'.\exists i, j : \emptyset.(x!^{\mathbf{t}}i \mid x!^{\mathbf{t}'}j))$. We can also obtain a variant of the linear channel type system [17]. Let A be $\neg\exists x.\exists \mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3.\exists n.ev(\langle x^{\mathbf{t}_1,\mathbf{t}_2} \rangle ev(x!^{\mathbf{t}_3}n \vee x?^{\mathbf{t}_3}n))$; Then it is guaranteed that every channel is used at most once.

Note that our type system is sound also for some non-negative formulas: Indeed, the deadlock-free property is not described as a non-negative formula, but our type system is still sound as proved in Appendix B.2. It is left for future work to identify a larger class of properties for which our type system is sound, and obtain a general type soundness theorem (like Corollary 6.4 above) for that class.

7 Limitations and Extensions

Although a variety of type systems can be obtained as its instances, our generic type system is of course not general enough to obtain all kinds of type systems. There are two major sources of limitations of our type system: One is the way in which processes are abstracted, and the other is the way the consistency condition ok on types is formalized.

Limitations caused by abstraction Because information on channel creation is lost in process types (recall the rule (T-New)), we cannot obtain type systems to guarantee properties like "at most n channels are created." We can overcome that limitation to some extent, by introducing a process type

[8] Instead, we could introduce a general fixed-point operator [32].

[9] Formally, $[\![A]\!]_{\mathbf{pr}}$ is parameterized by an assignment of variables to non-negative integers: See the full paper [11].

$\mathbf{new}_k.\Gamma$, which means that the process behaves like Γ after creating k channels.

Some information is also lost in the rule (T-In): Because information on the usage of bound channels (expressed by $\Gamma{\downarrow_{\{\tilde{y}\}}}$) is put into the continuation of an output process and that on the usage of free channels (expressed by $\Gamma{\uparrow_{\{\tilde{y}\}}}$) is put into the continuation of the input process, the causality information between communications on bound channels and those on free channels is lost. We can improve the type system by removing the restriction that tuple types cannot contain free variables (Notation 3.1.2) and changing the rule (T-In) into the following rule, to allow an arbitrary decomposition of information on the usage of bound and free channels:

$$\frac{\Gamma \triangleright P \qquad \Gamma_1 \,|\, \Gamma_2 \,\leq\, \Gamma \qquad FV(\Gamma_2) \cap \{\tilde{y}\} = \emptyset}{x?^{\mathbf{t}}[(\tilde{y})\Gamma_1].\,\Gamma_2 \triangleright x?^{\mathbf{t}}[\tilde{y}].\,P}$$

This kind of extension is necessary to account for some existing type systems like Abadi and Flanagan's type system for race detection [4]. In fact, the analysis of race discussed in Section 5 works only when the lock l and the other interfaces m and r are created simultaneously and passed together; Otherwise, dependencies between the usage of l, m and r are lost. To get rid of such restriction, the above extension of the rule (T-In) and an extension of the rule (T-New) discussed in the next paragraph is necessary.

Limitations caused by the formalization of ok To obtain common properties useful for proving type soundness (in Sections 3.6 and 6), we required that the consistency condition ok must be an invariant condition (recall Definition 3.4.3). This requirement is, however, sometimes too strong. For example, suppose that we want to guarantee a property "Before a channel x is used for output, y must be used for input." (This kind of requirement arises, for example, in ensuring safe locking [5].) Note that this property is not an invariant condition: Once y is used for input, x can be used immediately. One way to overcome this limitation would be to annotate each channel creation $(\nu\tilde{x})$ with the history of reductions, and parameterize ok with the history.

Another limitation comes from the side condition $ok(\Gamma{\downarrow_{\{\tilde{x}\}}})$ of the rule (T-New): Because of the operation $\cdot{\downarrow_{\{\tilde{x}\}}}$, only the causality between simultaneously created channels can be directly controlled. We can overcome this limitation by parameterizing the condition ok with a set of channels of interest, and replacing $ok(\Gamma{\downarrow_{\{\tilde{x}\}}})$ with $ok(\Gamma, \{\tilde{x}\})$.

Other extensions There are many other useful extensions. Combining our type system with polymorphism, existential types, etc. would be useful. We expect that polymorphism can be introduced in a similar manner to Pierce and Sangiorgi's polymorphic π-calculus [28].

The type system for deadlock-freedom in Section 4 is actually naive. More special treatment of event tags $\mathbf{t}$ is necessary to obtain a sophisticated type system for deadlock-freedom [14] (see the full paper [11]).

Besides type-soundness proofs and type inference issues studied in this paper, it would be interesting to formalize other aspects of type systems through our generic type system. Typed process equivalence would be especially important, because it is hard even for specific type systems [17, 27, 28].

Another interesting extension is generalization of the target language. If we can replace the π-calculus with a more abstract process calculus like Milner's action calculi [24], type systems for other process calculi can also be discussed uniformly.

8 Related Work

General framework of type systems Previous proposals of a general framework [9, 20, 21] are (i) so abstract (e.g., [9, 20]) that only a limited amount of work can be shared for developing concrete type systems, and/or (ii) not general (e.g., [20, 21]) enough to account for recent advanced type systems. Honda's framework [9] is more abstract than ours. Moreover, his framework only deal with what he call *additive* systems, where the composability of processes are determined solely by channel-wise compatibility: So, it cannot deal with properties like deadlock-freedom, for which inter-channel dependency is important. On the other hand, his framework can deal with a wide range of process calculi as target languages, not only the π-calculus. In König's type system based on hypergraphs [20], type environments do not change during reduction of a process. So, it cannot deal with dynamically changing properties like linearity [17]. Moreover, the target calculus is less expressive than the π-calculus, our target calculus: It cannot express dynamic creation of channels.

Other type systems viewing types as processes Some previous type systems also use process-like structures to express types. Yoshida's type system [38] (which guarantees a certain deadlock-freedom property) uses graphs to express the order of communications. Her type system is, however, specialized for a particular property, and the condition corresponding to our consistency condition seems too strong, even for guaranteeing deadlock-freedom.

Nielson and Nielson [25] also use CCS-like process terms to express the behavior of CML programs. Because their analysis approximates a set of channels by using an abstract channel called a region, it is not suitable for analyses of deadlock-freedom (see [14] for the reason), race detection, linearity analysis, etc, where the identify of a channel is important.

Process-like terms have been used as types also in type systems for deadlock-freedom [30] and related properties [31] of concurrent objects. As briefly outlined in Section 5, our type system can guarantee such properties without having concurrent objects as primitives.

Abstract interpretation As mentioned in Section 1, our generic type system can be viewed as a kind of abstract interpretation framework [2], in the sense that properties of programs are verified by reasoning about abstract versions of those programs. From this viewpoint, our contribution is a novel formalization of a specific subclass of abstract interpretation for the π-calculus (for which no satisfactory general abstract interpretation framework has been developed to the authors' knowledge) as a type system. Another novelty seems to be that while conventional abstract interpretation often uses a denotational semantics to claim the soundness of an analysis, our type system uses an operational semantics, which seems to be more convenient for analyses of concurrent processes.

Non-standard type systems for functional languages Unlike standard type systems for functional languages, our type system keeps track of not only the shape of each value but also information on how each value is accessed. In this respect, there seems to be some connection between our type system and non-standard type systems for functional languages, especially those for memory management [15, 36]. It would be interesting to study whether they can be encoded into some extension of our generic type system.

9 Conclusion

We have proposed a general type system for concurrent processes, where types are expressed as abstract processes. We have shown that a variety of non-trivial type systems can be obtained as its instances, and that their correctness can be proved in a uniform manner. Future work includes a study of a more general version of the type soundness theorem in Section 6, and extensions of our generic type system discussed in Section 7, to give a complete account of the existing type systems for the π-calculus.

Acknowledgment

We would like to thank Andrew Gordon, Eijiro Sumii, and anonymous referees for useful discussions and comments.

References

[1] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Proceedings of SAS'97*, LNCS 1302, pages 114–126. Springer-Verlag, 1997.

[4] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99*, LNCS 1664, pages 288–303. Springer-Verlag, 1999.

[5] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP 1999*, LNCS 1576, pages 91–108, 1999.

[6] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

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

[8] M. Hennessy and J. Riely. Information flow vs. resource access in the information asynchronous pi-calculus. In *Proceedings of ICALP 2000*, LNCS 1853. Springer-Verlag, 2000.

[9] K. Honda. Composing processes. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 344–357, 1996.

[10] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*, LNCS 1782. Springer-Verlag, 2000.

[11] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. Tech. rep., Department of Information Science, University of Tokyo, 2000. to appear.

[12] A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation*, 161:1–44, 2000.

[13] C. B. Jones. A pi-calculus semantics for an object-based design notation. In *Proceedings of CONCUR'93*, LNCS, pages 158–172. Springer-Verlag, 1993.

[14] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

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

[16] N. Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *Proceedings of IFIP International Conference on Theoretical Computer Science (TCS2000)*, LNCS 1872, pages 365–389, 2000.

[17] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[18] N. Kobayashi, E. Sumii, and S. Saito. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*, LNCS 1877, pages 489–503. Springer-Verlag, 2000.

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

[20] B. König. Generating type systems for process graphs. In *Proceedings of CONCUR'99*, LNCS 1664, pages 352–367. Springer-Verlag, 1999.

[21] B. König. Analysing input/output-capabilities of mobile processes with a generic type system. In *Proceedings of ICALP2000*, LNCS 1853. Springer-Verlag, 2000.

[22] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[23] 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.

[24] R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

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

[26] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 276–290, 1999.

[27] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[28] B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the Association for Computing Machinery (JACM)*, 47(5):531–584, 2000.

[29] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, LNCS 907, pages 187–215. Springer-Verlag, 1995.

[30] F. Puntigam and C. Peter. Changeable interfaces and promised messages for concurrent components. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 141–145, 1999.

[31] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *Proceedings of CONCUR2000*, LNCS 1877, pages 474–488, 2000.

[32] C. Stirling. Modal and temporal logics for processes. In *Logics for Concurrency*, LNCS 1043, pages 149–237, 1996.

[33] E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, ENTCS 16(3), pages 55–77, 1998.

[34] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[35] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 162–173, 1992.

[36] M. Tofte and J.-P. Talpin. Implementation of the call-by-value lambda-calculus using a stack of regions. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[37] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In *CONCUR'93*, LNCS 715, pages 524–538. Springer-Verlag, 1993.

[38] N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, LNCS 1180, pages 371–387. Springer-Verlag, 1996.

Appendix

A Type Reconstruction

The type reconstruction algorithm PT given below takes a process expression as input and outputs a pair of a type environment (extended with expressions like $\alpha\uparrow_S$ and $\alpha\downarrow_S$) and a set of constraints on variables in the type environment. Each constraint is of the form either $ok(\Gamma)$ or $\alpha \leq \Gamma$. The obtained pair gives a principal typing of the input in the sense that all the possible type environments, under which the process expression is well typed, are obtained from the type environment in the pair, by replacing free variables in the constraint so that the constraint is satisfied.

$PT(P) = (\Gamma, C)$:

$PT(\mathbf{0}) = (\mathbf{0}, \emptyset)$
$PT(x!^{\mathbf{t}}[\tilde{z}].\, P_0) =$
  let $(\Gamma_0, C_0) = PT(P_0)$
  in $(x!^{\mathbf{t}}[(\tilde{z})\alpha_x].\, (\alpha \,|\, \alpha_x), C_0 \cup \{\alpha \leq \Gamma_0\})$
  (where α and α_x are fresh)
$PT(x?^{\mathbf{t}}[\tilde{y}].\, P_0) =$
  let $(\Gamma_0, C_0) = PT(P_0)$
  in $(x?^{\mathbf{t}}[(\tilde{y})\alpha\downarrow_{\{\tilde{y}\}}].\, (\alpha\uparrow_{\{\tilde{y}\}}), C_0 \cup \{\alpha \leq \Gamma_0\})$
  (where α is fresh)
$PT(P_1 \,|\, P_2) =$
  let $(\Gamma_1, C_1) = PT(P_1)$
    $(\Gamma_2, C_2) = PT(P_2)$
  in $(\Gamma_1 \,|\, \Gamma_2, C_1 \cup C_2)$
$PT(P_1 + \cdots + P_n) =$
  let $(\Gamma_1, C_1) = PT(P_1)$
    $\vdots$
    $(\Gamma_n, C_n) = PT(P_n)$
  in $(\Gamma_1 + \cdots + \Gamma_n, C_1 \cup \cdots \cup C_n)$
$PT(*P_0) =$
  let $(\Gamma_0, C_0) = PT(P_0)$
  in $(*\Gamma_0, C_0)$
$PT((\nu\tilde{x})\, P_0) =$
  let $(\Gamma_0, C_0) = PT(P_0)$
  in $(\Gamma_0\uparrow_{\{\tilde{x}\}}, C_0 \cup \{ok(\Gamma_0\downarrow_{\{\tilde{x}\}})\})$

B Proofs of Theorems

B.1 Proof of Subject Reduction Theorem (Theorem 3.6.1)

Lemma B.1.1: $(\Gamma\uparrow_{S_1})\uparrow_{S_2} = (\Gamma\uparrow_{S_2})\uparrow_{S_1} = \Gamma\uparrow_{S_1 \cup S_2}$.

Proof: By induction on the structure of Γ. □

Lemma B.1.2 [inversion]: Suppose $\Gamma \triangleright P$.

1. If $P = P_1 \,|\, P_2$, then there exist Γ_1 and Γ_2 such that $\Gamma_i \triangleright P_i$ for $i = 1, 2$ and $\Gamma \leq \Gamma_1 \,|\, \Gamma_2$.
2. If $P = P_1 + \cdots + P_n$, then there exist $\Gamma_1, \ldots, \Gamma_n$ such that $\Gamma_i \triangleright P_i$ for $i = 1, \ldots, n$ and $\Gamma \leq \Gamma_1 + \cdots + \Gamma_n$.
3. If $P = x!^{\mathbf{t}}[\tilde{y}].\, P_0$, then there exist Γ_0 and Γ_x such that $\Gamma_0 \triangleright P_0$ and $\Gamma \leq x!^{\mathbf{t}}[(\tilde{y})\Gamma_x].\, (\Gamma_0 \,|\, \Gamma_x)$.
4. If $P = x?^{\mathbf{t}}[\tilde{y}].\, P_0$, then there exists Γ_0 such that $\Gamma_0 \triangleright P_0$ and $\Gamma \leq x?^{\mathbf{t}}[(\tilde{y})\Gamma_0\downarrow_{\{\tilde{y}\}}].\, (\Gamma_0\uparrow_{\{\tilde{y}\}})$.

5. If $P = (\nu\tilde{x})\,P_0$, then there exists Γ_0 such that $\Gamma_0 \triangleright P_0$ and $\Gamma \leq \Gamma_0\uparrow_{\{\tilde{x}\}}$ with $ok(\Gamma\downarrow_{\{\tilde{x}\}})$.

6. If $P = *P_0$, then there exists Γ_0 such that $\Gamma_0 \triangleright P_0$ and $\Gamma \leq *\Gamma_0$.

Proof: Immediate from the fact that a type derivation of $\Gamma \triangleright P$ must end with an application of the rule corresponding to the form of P, followed by zero or more applications of the rule T-Sub. □

Lemma B.1.3: If $\Gamma \xrightarrow{S} \Gamma'$, then $\Gamma\uparrow_{\{\tilde{x}\}} \xrightarrow{S'} \Gamma'\uparrow_{\{\tilde{x}\}}$ where S' is the least set that satisfies the following conditions:

1. $x_i^{\mathbf{t}_1,\mathbf{t}_2} \in S$ implies $x_i^{\mathbf{t}_1,\mathbf{t}_2} \notin S'$ and $\{\mathbf{t}_1, \mathbf{t}_2\} \subseteq S'$;
2. $\mathbf{t} \in S$ implies $\mathbf{t} \in S'$; and
3. $y^{\mathbf{t}_1,\mathbf{t}_2} \in S$ and $y \notin \{\tilde{x}\}$ imply $y^{\mathbf{t}_1,\mathbf{t}_2} \in S'$.

Proof: By induction on the derivation of $\Gamma \xrightarrow{S} \Gamma'$ with a case analysis on the last rule used. We show only the main base case below; the other cases are easy.

Case TER-Com: $\Gamma = \cdots + y!^{\mathbf{t}_1}[\tau_1].\,\Gamma_1 + \cdots \mid \cdots + y?^{\mathbf{t}_2}[\tau_2].\,\Gamma_2 + \cdots$, $\Gamma' = \Gamma_1 \mid \Gamma_2$, $S = \{y^{\mathbf{t}_1,\mathbf{t}_2}\}$, $\tau_1 \leq \tau_2$

We have two subcases according to whether $y \in \{\tilde{x}\}$ or not. We will show the subcase where $y = x_i$; the other subcase is easy. Since $\Gamma\uparrow_{\{\tilde{x}\}} = \cdots + \mathbf{t}_1.\Gamma_1\uparrow_{\{\tilde{x}\}} + \cdots \mid \cdots + \mathbf{t}_2.\Gamma_2\uparrow_{\{\tilde{x}\}} + \cdots$, it is easy to show that

$$\Gamma\uparrow_{\{\tilde{x}\}} \xrightarrow{\{\mathbf{t}_1,\mathbf{t}_2\}} \Gamma_1\uparrow_{\{\tilde{x}\}} \mid \Gamma_2\uparrow_{\{\tilde{x}\}}$$

finishing the subcase. □

Lemma B.1.4: If $WF(\Gamma\downarrow_S)$ and $WF(\Gamma\uparrow_S)$, then $WF(\Gamma)$.

Proof: Suppose $WF(\Gamma)$ does not hold. Then, there exist x, τ_1, τ_2, $\mathbf{t}_1$, $\mathbf{t}_2$, Γ_1, Γ_2, and Γ_3 such that

$$\Gamma \longrightarrow^* \cdots + x!^{\mathbf{t}_1}[\tau_1].\,\Gamma_1 + \cdots \mid \cdots + x?^{\mathbf{t}_2}[\tau_2].\,\Gamma_2 + \cdots \mid \Gamma_3$$

and $\tau_1 \not\leq \tau_2$. By using Lemma B.1.3 repeatedly, either

$$\Gamma\uparrow_S \longrightarrow^* \cdots + x!^{\mathbf{t}_1}[\tau_1].\,(\Gamma_1\uparrow_S) + \cdots \mid \cdots + x?^{\mathbf{t}_2}[\tau_2].\,(\Gamma_2\uparrow_S) + \cdots \mid \Gamma_3\uparrow_S$$

or

$$\Gamma\downarrow_S \longrightarrow^* \cdots + x!^{\mathbf{t}_1}[\tau_1].\,(\Gamma_1\downarrow_S) + \cdots \mid \cdots + x?^{\mathbf{t}_2}[\tau_2].\,(\Gamma_2\downarrow_S) + \cdots \mid \Gamma_3\downarrow_S,$$

contradicting the assumption $WF(\Gamma\downarrow_S)$ and $WF(\Gamma\uparrow_S)$. □

Lemma B.1.5 [substitution]: If $\Gamma \triangleright P$, then $[y/x]\Gamma \triangleright [y/x]P$.

Proof: By straightforward induction on the derivation of $\Gamma \triangleright P$. □

Lemma B.1.6: If $\Gamma \triangleright P$ and $P \preceq Q$, then $\Gamma \triangleright Q$.

Proof: By structural induction on the derivation of $P \preceq Q$ with a case analysis of the last rule used. We show a few interesting cases below; the other cases are easy.

Case: $P = (\nu\tilde{x})\,P_1 \mid P_2$ $\quad Q = (\nu\tilde{x})\,(P_1 \mid P_2)$, $\tilde{x}$ are not free in P_2

By Lemma B.1.2, there exist Γ_1, Γ_1', and Γ_2 such that $\Gamma_i \triangleright P_i$ for $i \in \{1,2\}$ and $\Gamma \leq \Gamma_1' \mid \Gamma_2$ and $\Gamma_1' \leq \Gamma_1\uparrow_{\{\tilde{x}\}}$ with $ok(\Gamma_1\downarrow_{\{\tilde{x}\}})$. By the rule T-Par, $\Gamma_1 \mid \Gamma_2 \triangleright P_1 \mid P_2$. Without loss of generality, we can assume $\tilde{x}$ are not free in P_2 so that $\Gamma_2\uparrow_{\{\tilde{x}\}} = \Gamma_2$ and $Null(\Gamma_2\downarrow_{\{\tilde{x}\}})$. Then, $(\Gamma_1 \mid \Gamma_2)\uparrow_{\{\tilde{x}\}} = \Gamma_1\uparrow_{\{\tilde{x}\}} \mid \Gamma_2$ and, by the third condition in Definition 3.4.3, $ok((\Gamma_1 \mid \Gamma_2)\downarrow_{\{\tilde{x}\}})$. Thus,

$$\Gamma_1\uparrow_{\{\tilde{x}\}} \mid \Gamma_2 \triangleright (\nu\tilde{x})\,(P_1 \mid P_2)$$

by the rule T-New. Finally, it is easy to show $\Gamma \leq \Gamma_1\uparrow_{\{\tilde{x}\}} \mid \Gamma_2$, and so, by the rule T-Sub, $\Gamma \triangleright Q$.

Case: $P = *P_0$ $\quad Q = *P_0 \mid P_0$

By Lemma B.1.2, we have $\Gamma_0 \triangleright P_0$ and $\Gamma \leq *\Gamma_0$. Then, by the rule T-Par, $\Gamma_0 \mid *\Gamma_0 \triangleright P_0 \mid *P_0$. Since $*\Gamma_0 \leq \Gamma_0 \mid *\Gamma_0$, by using the rule T-Sub, $\Gamma \triangleright Q$. □

Proof of Theorem 3.6.1: By induction on the derivation of $P \xrightarrow{l} Q$ with a case analysis on the last rule used. We show only main cases below.

Case R-Com: $P = \cdots + x!^{\mathbf{t}}[\tilde{z}].\,P_0 + \cdots \mid \cdots + x?^{\mathbf{t}'}[\tilde{y}].\,Q_0 + \cdots$, $Q = P_0 \mid [\tilde{z}/\tilde{y}]Q_0$ $\quad l = x^{\mathbf{t},\mathbf{t}'}$

By Lemma B.1.2 and the subtyping rules, there exist Γ_1, Γ_2, and Γ_3 such that

$$\Gamma \leq \cdots + x!^{\mathbf{t}}[(\tilde{z})\Gamma_3].\,(\Gamma_1 \mid \Gamma_3) + \cdots \mid \cdots + x?^{\mathbf{t}'}[(\tilde{y})\Gamma_2\downarrow_{\{\tilde{y}\}}].\,(\Gamma_2\uparrow_{\{\tilde{y}\}})$$
$$\Gamma_1 \triangleright P_0 \qquad \Gamma_2 \triangleright Q_0.$$

Then, since $WF(\Gamma)$, it must be the case that $(\tilde{z})\Gamma_3 \leq (\tilde{y})\Gamma_2\downarrow_{\{\tilde{y}\}}$, that is, $\Gamma_3 \leq [\tilde{z}/\tilde{y}]\Gamma_2\downarrow_{\{\tilde{y}\}}$. We can show

$$\Gamma \xrightarrow{\{x^{\mathbf{t},\mathbf{t}'}\}} \Gamma_1 \mid [\tilde{z}/\tilde{y}]\Gamma_2$$

by the following calculation:

$$\begin{array}{lll} \Gamma & \xrightarrow{\{x^{\mathbf{t},\mathbf{t}'}\}} & \Gamma_1 \mid \Gamma_3 \mid \Gamma_2\uparrow_{\{\tilde{y}\}} \\ & \leq & \Gamma_1 \mid [\tilde{z}/\tilde{y}]\Gamma_2\downarrow_{\{\tilde{y}\}} \mid \Gamma_2\uparrow_{\{\tilde{y}\}} \\ & = & \Gamma_1 \mid [\tilde{z}/\tilde{y}](\Gamma_2\downarrow_{\{\tilde{y}\}} \mid \Gamma_2\uparrow_{\{\tilde{y}\}}) \\ & \leq & \Gamma_1 \mid [\tilde{z}/\tilde{y}]\Gamma_2 \quad \text{(Sub-Divide and Sub-Subst)} \end{array}$$

By Lemma B.1.5, $[\tilde{z}/\tilde{y}]\Gamma_2 \triangleright [\tilde{z}/\tilde{y}]Q_0$. Finally, by the rule T-Par, $\Gamma_1 \mid [\tilde{z}/\tilde{y}]\Gamma_2 \triangleright P_0 \mid [\tilde{z}/\tilde{y}]Q_0$, finishing the case.

Case R-Par: $P = P_0 \mid R$ $\quad P_0 \xrightarrow{l} Q_0$ $\quad Q = Q_0 \mid R$

By Lemma B.1.2, there exist Γ_1 and Γ_2 such that

$$\Gamma \leq \Gamma_1 \mid \Gamma_2 \quad \Gamma_1 \triangleright P_0 \quad \Gamma_2 \triangleright R$$

To use the induction hypothesis, we show $WF(\Gamma_1)$ by contradiction. Suppose $WF(\Gamma_1)$ does not hold. Then, $WF(\Gamma_1 \mid \Gamma_2)$ does not hold, either; it means $WF(\Gamma)$ does not hold. Thus, $WF(\Gamma_1)$.

By the induction hypothesis, there exists Γ_1' such that $\Gamma_1 \xrightarrow{l^\#} \Gamma_1'$ and $\Gamma_1' \triangleright Q_0$. By the rules TER-Skip, TER-Par, and TER-Sub, $\Gamma \xrightarrow{l^\#} \Gamma_1' \mid \Gamma_2$. By the rule T-Par, $\Gamma_1' \mid \Gamma_2 \triangleright Q_0 \mid R$, finishing the case.

Case R-NEW1: $P = (\nu\tilde{x})\,P_0 \quad P_0 \xrightarrow{y^{\mathbf{t},\mathbf{t}'}} Q_0 \quad y \in \{\tilde{x}\}$
$Q = (\nu\tilde{x})\,Q_0 \quad l = \epsilon^{\mathbf{t},\mathbf{t}'}$

By Lemma B.1.2, there exists Γ_0 such that

$$\Gamma_0 \triangleright P_0 \quad \Gamma \leq \Gamma_0\!\uparrow_{\{\tilde{x}\}} \quad ok(\Gamma_0\!\downarrow_{\{\tilde{x}\}}).$$

To use the induction hypothesis, we will show $WF(\Gamma_0)$. Since $WF(\Gamma_0\!\downarrow_{\{\tilde{x}\}})$ and $WF(\Gamma_0\!\uparrow_{\{\tilde{x}\}})$ from the assumptions, we have $WF(\Gamma_0)$ by Lemma B.1.4.

By the induction hypothesis, there exists Γ_0' such that $\Gamma_0 \xrightarrow{\{y^{\mathbf{t},\mathbf{t}'}\}} \Gamma_0'$ and $\Gamma_0' \triangleright Q_0$. By Lemma B.1.3, $\Gamma_0\!\uparrow_{\{\tilde{x}\}} \xrightarrow{\{\mathbf{t},\mathbf{t}'\}} \Gamma_0'\!\uparrow_{\{\tilde{x}\}}$ and $\Gamma_0\!\downarrow_{\{\tilde{x}\}} \xrightarrow{\{y^{\mathbf{t},\mathbf{t}'}\}} \Gamma_0'\!\downarrow_{\{\tilde{x}\}}$. Then, $ok(\Gamma_0'\!\downarrow_{\{\tilde{x}\}})$ and, by the rule T-NEW, $\Gamma_0'\!\uparrow_{\{\tilde{x}\}} \triangleright (\nu\tilde{x})\,Q_0$, finishing the case. □

B.2 Proof of Theorem 4.1

Lemma B.2.7: Let $\leq$ be any proper subtyping relation and ok be ok_1. If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_1(P)$ holds.

Proof: Suppose that $\Gamma \triangleright P$ and $ok(\Gamma)$ hold but $p_1(P)$ does not hold. By the definition of p_1, there exist Q_1 and Q_2 such that

$$P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x!^{\mathbf{t}}[\tilde{z}].\,Q_1 + \cdots \mid \cdots + x?^{\mathbf{t}'}[\tilde{w}].\,Q_2 + \cdots \mid Q_3)$$

with $\|\tilde{z}\| \neq \|\tilde{w}\|$. By Theorem 3.6.3, it must be the case that

$$\begin{array}{l}\Gamma_k \triangleright_N \cdots + x!^{\mathbf{t}}[\tilde{z}].\,Q_1 + \cdots \mid \cdots + x?^{\mathbf{t}'}[\tilde{w}].\,Q_2 + \cdots \mid Q_3\\ ok_1(\Gamma_i\!\downarrow_{\{\tilde{x_i}\}}) \text{ for each } i \in \{1,\ldots,k\}\\ \Gamma_{i-1} = \Gamma_i\!\uparrow_{\{\tilde{x_i}\}} \text{ for each } i \in \{1,\ldots,k\}\\ \Gamma \leq \Gamma_0\end{array}$$

$\Gamma \leq \Gamma_0$ and $ok_1(\Gamma)$ imply $ok_1(\Gamma_0)$. By the typing rules, Γ_k must be of the form $\cdots + x!^{\mathbf{t}}[\tau].\,\Delta_1 + \cdots \mid \cdots + x!^{\mathbf{t}}[\tau'].\,\Delta_1 + \cdots \mid \Delta_3$ with $\|\tau\| = \|\tilde{z}\| \neq \|\tilde{w}\| = \|\tau'\|$. This contradicts with the facts $ok_1(\Gamma_0)$ and $ok_1(\Gamma_i\!\downarrow_{\{\tilde{x_i}\}})$ for each $i \in \{1,\ldots,k\}$. □

Lemma B.2.8: Let $\leq$ be any proper subtyping relation and ok be ok_2. If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_2(P)$ holds.

Proof: Similar to the proof of Lemma B.2.7. □

Lemma B.2.9: Let $\leq$ be the least proper subtyping relation, and let ok be ok_3. If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_3(P)$ holds.

Proof: The proof proceeds by induction on $\mathbf{t}$. Suppose $\Gamma \triangleright P$, $ok(\Gamma)$, and $P \longrightarrow^* P' = (\widetilde{\nu x}_{1..n})\,(\cdots + y?^{\mathbf{t}}[\tilde{z}].\,Q + \cdots \mid R)$. (The case where $P \longrightarrow^* P' = (\widetilde{\nu x}_{1..n})\,(\cdots + y!^{\mathbf{t}}[\tilde{z}].\,Q + \cdots \mid R)$ is similar.)

Without loss of generality, we can assume that y is free in P'. Otherwise, by Theorem 3.6.3, we have

$$\begin{array}{l}\Gamma_n \triangleright \cdots + y?^{\mathbf{t}}[\tilde{z}].\,Q + \cdots \mid R\\ \Gamma_{i-1} = \Gamma_i\!\uparrow_{\{\tilde{x_i}\}} \text{ for } i = 1,\ldots,n\\ ok(\Gamma_i\!\downarrow_{\{\tilde{x_i}\}}) \text{ for } i = 1,\ldots,n\\ \Gamma \leq \Gamma_0\\ y \in \{\tilde{x}_j\} \text{ for some } j\end{array}$$

By Lemma B.1.1, we have

$$\Gamma_j\!\downarrow_{\{\tilde{x_j}\}} \triangleright (\nu\tilde{w})\,(\widetilde{\nu x}_{1..j-1})\,(\widetilde{\nu x}_{j+1..n})\,(\cdots + y?^{\mathbf{t}}[\tilde{z}].\,Q + \cdots \mid R)$$

and $ok(\Gamma_j\!\downarrow_{\{\tilde{x_j}\}})$.

By Theorem 3.6.1 and the definition of ok, there exists Δ such that $\Delta \triangleright P'$ and $\Gamma \longrightarrow^* \Delta$. By Theorem 3.6.3, there exists τ, Γ_1, and Γ_2 such that

$$\begin{array}{l}\Delta \leq \cdots + y?^{\mathbf{t}'}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2\\ \cdots + y?^{\mathbf{t}'}[\tau].\,\Gamma_1 + \cdots \mid \Gamma_2 \triangleright_N P'\end{array}$$

Because $ok(\Gamma)$ holds, it must be the case that $\Gamma' \xrightarrow{L} \Gamma''$ for some L such that (i) $L = \{x^{\mathbf{t}_1,\mathbf{t}_2}\}$ or (ii) $L = \{\mathbf{t}''\}$ and $\mathbf{t}'' \prec \mathbf{t}'$. So, by the typing rules, one of the following conditions must hold (c.f. Lemma 3.6.5):

1. $P' \preceq (\widetilde{\nu u}_{1..m})\,(\cdots + w?^{\mathbf{t}''}[\tilde{z}].\,Q_1 + \cdots \mid Q_2)$ with $w \in \{\tilde{u_1},\ldots,\tilde{u_m}\}$.
2. $P' \preceq (\widetilde{\nu u}_{1..m})\,(\cdots + w!^{\mathbf{t}''}[\tilde{z}].\,Q_1 + \cdots \mid Q_2)$ with $w \in \{\tilde{u_1},\ldots,\tilde{u_m}\}$.
3. $P' \preceq (\widetilde{\nu u}_{1..m})\,(\cdots + x!^{\mathbf{t}_1}[\tilde{z}].\,Q_1 + \cdots \mid \cdots + x?^{\mathbf{t}_2}[\tilde{z}].\,Q_2 + \cdots \mid Q_3)$.

In the first and second cases, $P' \longrightarrow$ follows by induction hypothesis. In the third case, $P' \longrightarrow$ follows immediately. □

Lemma B.2.10: Let $\leq$ be any proper subtyping relation and ok be ok_4. If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_4(P)$ holds.

Proof: Suppose that $\Gamma \triangleright P$ and $ok_4(\Gamma)$ hold. Suppose also that $P \longrightarrow^* \preceq (\widetilde{\nu w})\,P'$ and $P' \xrightarrow{x^{\mathbf{t}',\mathbf{t}}} \longrightarrow^* (\widetilde{\nu u})\,(\cdots + x?[\tilde{y}].\,Q_1 + \cdots \mid Q_2)$. Without loss of generality (c.f. the proof of Lemma B.2.9), we can assume that x is free in P. So, we have

$$P \longrightarrow^* \xrightarrow{x^{\mathbf{t}',\mathbf{t}}} \longrightarrow^* (\widetilde{\nu w})\,(\widetilde{\nu u})\,(\cdots + x?^{\mathbf{t}''}[\tilde{y}].\,Q_1 + \cdots \mid Q_2).$$

By Theorem 3.6.1, there exists Γ' such that $\Gamma \longrightarrow^* \xrightarrow{x^{\mathbf{t}',\mathbf{t}}} \longrightarrow^* \Gamma'$ and $\Gamma' \triangleright (\widetilde{\nu w})\,(\widetilde{\nu u})\,(\cdots + x?^{\mathbf{t}''}[\tilde{y}].\,Q_1 + \cdots \mid Q_2)$. By Corollary 3.6.4, it must be the case that $\Gamma' \leq \cdots + x?^{\mathbf{t}''}[\tau].\,\Gamma_1' + \cdots \mid \Gamma_2'$. This contradicts with the assumption $ok_4(\Gamma)$. The case for output is similar. □

Proof of Theorem 4.1: This follows immediately from Lemmas B.2.7–B.2.10 and Theorem 3.6.2. □