A Generic Type System for the Pi-Calculus*

Atsushi Igarashi
Kyoto University
email:igarashi@kuis.kyoto-u.ac.jp

Naoki Kobayashi
Tokyo Institute of Technology
email:kobayasi@cs.titech.ac.jp

June 10, 2003

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: Since 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 [40], multiplicities (how often each channel is used) [30], race conditions [10, 12], deadlock [27, 31, 43, 52], livelock [28], and information flow [19, 22, 23].

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. Moreover, 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 [49], and exception analysis [8, 39]) can be obtained as instances of the effect analysis [47, 48].

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.

*A preliminary summary of this paper appeared in Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

1.2 Main Ideas

The main idea of the present work is to express types and type environments as *abstract processes*. 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 [5].) 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 [28, 31, 46]. 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 [31, 46], 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 $z![\,] \,|\, z?[\,].\, z?[\,].\, \mathbf{0}$, 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 [28, 31, 46].

We push the above "type environments as abstract processes" view further, and express type environments as CCS-like processes (unlike in CCS [35], however, we have no operator for hiding or creating channels). The type environment of the above process P is expressed as $x![\tau].\, z![\tau'] \,|\, x?[\tau].\, \mathbf{0} \,|\, z?[\tau'].\, z?[\tau'].\, \mathbf{0}$. 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 [43, 44].

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 parameterized by types, so that it 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 (see Section 3).

- 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. Using those properties, we also prove a general type soundness property, from which the soundness of a certain class of instances of the generic type system can be immediately obtained (see Section 4).

- 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 soundness. Thanks to the general properties mentioned above, the proof for each instance of the generic type system is quite short; indeed, the soundness of most of the instances follows as an immediate corollary of the general type soundness theorem (see Sections 5 and 6).

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 Section 4 discusses the soundness of the generic type system. Section 5 derives a variety of type systems as instances of the generic type system. To further demonstrate the strength of our framework, Section 6 shows that deadlock and race conditions of concurrent objects can also be analyzed within our generic type system. Section 7 formalizes a part of type-checking/reconstruction algorithms that is common to all instances of the generic type system. Section 8 discusses limitations and extensions of our generic type system. Section 9 discusses related work and Section 10 concludes this paper.

2 Target Language

This section introduces the syntax and the operational semantics of our target language.

2.1 Syntax

Our calculus is basically a subset of the polyadic π-calculus [36]. 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.

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

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 $(\widetilde{\nu \tilde{x}_{1..n}})\, P$ or $(\widetilde{\nu \tilde{x}})\, P$. 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

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

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

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

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

$$(\nu \tilde{x})\, P \,|\, Q \preceq (\nu \tilde{x})\, (P \,|\, Q) (\text{if } \tilde{x} \text{ are not free in } Q) \qquad \text{(SP-New)}$$

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

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

Figure 1: Structural Preorder

inaction $\mathbf{0}$ and write $x!^{\mathbf{t}}[\tilde{y}]$ and $x?^{\mathbf{t}}[\tilde{y}]$ for $x!^{\mathbf{t}}[\tilde{y}].\,\mathbf{0}$ and $x?^{\mathbf{t}}[\tilde{y}].\,\mathbf{0}$ respectively. 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 also abbreviate $x!^{\mathbf{t}}[\,].\,P$ and $x?^{\mathbf{t}}[\,].\,P$ to $x!^{\mathbf{t}}.\,P$ and $x?^{\mathbf{t}}.\,P$ respectively. We give precedence to prefixes ($(\nu \tilde{x})$, $x!^{\mathbf{t}}[\tilde{y}].$, and $x?^{\mathbf{t}}[\tilde{y}].$), $+$, and $|$ in this order.

The meanings of $\mathbf{0}$, $x!^{\mathbf{t}}[\tilde{y}].\,P$, $x?^{\mathbf{t}}[\tilde{y}].\,P$, and $P \,|\, Q$ 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.[1] $*P$ denotes infinitely many copies of P running in parallel.

2.2 Operational Semantics

As usual [36], 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 label (which we call a *reduction label*) of the form $x^{\mathbf{t},\mathbf{t}'}$ or $\epsilon^{\mathbf{t},\mathbf{t}'}$. A reduction label records which channels and events are involved in the reduction: The label $x^{\mathbf{t},\mathbf{t}'}$ means that a communication occurss on a channel x and the output and input processes are labeled with $\mathbf{t}$ and $\mathbf{t}'$ respectively. The reduction label $\epsilon^{\mathbf{t},\mathbf{t}'}$ means that a communication occurs on a bound channel and the output and input processes are labeled with with $\mathbf{t}$ and $\mathbf{t}'$ respectively. Reduction labels are used to state properties of a process in Section 5.

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: Let l be a reduction label, and S be a set of variables. We define $l{\uparrow_S}$ by:

$$\begin{array}{rcl} (x^{\mathbf{t},\mathbf{t}'}){\uparrow_S} & = & \begin{cases} \epsilon^{\mathbf{t},\mathbf{t}'} & \text{if } x \in S \\ x^{\mathbf{t},\mathbf{t}'} & \text{otherwise} \end{cases} \\ (\epsilon^{\mathbf{t},\mathbf{t}'}){\uparrow_S} & = & \epsilon^{\mathbf{t},\mathbf{t}'} \end{array}$$

[1]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.

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

$$\frac{P \xrightarrow{l} Q}{P \mid R \xrightarrow{l} Q \mid R} \qquad \text{(R-Par)}$$

$$\frac{P \xrightarrow{l} Q}{(\nu \tilde{x})\,P \xrightarrow{l \uparrow_{\{\tilde{x}\}}} (\nu \tilde{x})\,Q} \qquad \text{(R-New)}$$

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

Figure 2: Reduction Relation

Definition 2.2.3: The reduction relation $\xrightarrow{l}$ is the least relation closed under the rules in Figure 2.

Notation 2.2.4: We write $P \longrightarrow Q$ if $P \xrightarrow{l} Q$ for some l.

Notation 2.2.5: 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

This section introduces our generic type system and shows its properties.

3.1 Types

We first define the syntax of types. We have two kinds of types: types of tuples (called *tuple types*) and those of processes (called *process types*). Process types correspond to the type environments mentioned in Section 1; they express abstract behavior of processes. We assume that we have countably infinite sets of type variables for each size of tuples. We write $\alpha^{(n)}$ for a type variable ranging over a type of n-tuples.

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

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

A meta-variable $\tau^{(n)}$ ranges over the type of n-tuples. We often omit n when it is not important or it is clear form the context.

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. We use the standard notation $\mu\alpha.\tau$ for recursive types. Recursive types are used to express a channel that is used infinitely often (e.g., a channel that is repeatedly used for input), as well as a channel that carries itself (e.g., $x!^{\mathbf{t}}[x]$). Examples are given later in Example 3.1.6.

$\mathbf{0}$ is the type of the inaction. $x!^{\mathbf{t}}[\tau^{(n)}].\,\Gamma$ is the type of a process that uses x for sending an n-tuple of type τ, and then behaves according to Γ. The output on x must be tagged with $\mathbf{t}$. Similarly, $x?^{\mathbf{t}}[\tau^{(n)}].\,\Gamma$ is the type of a process that uses x for receiving an n-tuple of type τ, and then behaves according to Γ. In this way, we can express more precise information on the usage of channels than previous type systems [30, 31, 40]. $\mathbf{t}.\Gamma$ 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.[2] $\Gamma_1 \,|\, \Gamma_2$ 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. The type $\tau^{(n)}\langle x_1, \ldots, x_n\rangle$ describes a process that uses channels $x_1, \ldots, x_n$ according to type τ. Here, the arity information is used to exclude out ill-formed types like $((x)\Gamma)\langle y, z\rangle$.

Notation 3.1.2: We write $\|\tau^{(n)}\|$ for the arity n. We have two kinds of binders: one for (channel) variables and the other for type variables. The tuple type $(\tilde{x})\Gamma$ binds the variables $\tilde{x}$ in Γ. We assume that $\mu\alpha.\Gamma$ binds the type variable α in Γ. We assume that α-conversions are implicitly applied so that bound variables are always different from each other and free variables. We write $[\tilde{y}/\tilde{x}]$ and $[\tau/\alpha]$ for the capture-avoiding substitution of $\tilde{y}$ for $\tilde{x}$ and that of τ for α respectively. We write $*\Gamma$ for $(\mu\alpha.(\,)(\Gamma \,|\, \alpha\langle\,\rangle))\langle\,\rangle$ (where α does not appear in Γ); It is the type of infinitely many copies of a process of type Γ. We often omit $\mathbf{0}$ and write $x!^{\mathbf{t}}[\tau]$, $x?^{\mathbf{t}}[\tau]$, and $\mathbf{t}$ for $x!^{\mathbf{t}}[\tau].\,\mathbf{0}$, $x?^{\mathbf{t}}[\tau].\,\mathbf{0}$, and $\mathbf{t}.\mathbf{0}$ 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 write $\mathit{Null}(\Gamma)$ if Γ does not contain a process type of the form $x?^{\mathbf{t}}[\tau].\,\Gamma_1$ or $x!^{\mathbf{t}}[\tau].\,\Gamma_1$. We give precedence to prefixes ($\mathbf{t}.$, $x!^{\mathbf{t}}[\tau].$, and $x?^{\mathbf{t}}[\tau].$), $+$, $\&$, and $|$ in this order.

Definition 3.1.3: A tuple type or process type is *closed* if it contains no free type variables. A tuple or process type is *semi-closed* if it contains no free type variables inside $[\,]$. A tuple type τ is *contractive* in α if τ contains free occurences of α only inside $[\,]$.

Example 3.1.4: A process type $x?[\mu\alpha.(x)(x?^{\mathbf{t}}[\tau].\,\alpha\langle x\rangle)].\alpha'\langle x\rangle$ is semi-closed, but not closed. $x![\alpha]$ is contractive in α but $x?[\alpha].\,\alpha\langle\,\rangle$ is not.

Notice that for most of the process constructors introduced in the previous section, there are the corresponding constructors for process types. The same symbols are used for them to clarify the correspondence. Unlike the π-calculus processes in Section 2, however, 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. An internal choice $\Gamma_1 \& \Gamma_2$ is necessary to express non-deterministic behavior of a process. For example, suppose that a process P_1 behaves like Γ_1 and a process P_2 behaves like Γ_2. Then, the process $\nu x(x! \,|\, x?.\,P_1 \,|\, x?.\,P_2)$ behaves like $\Gamma_1 \& \Gamma_2$. 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 $(\nu x)\,(x?^{\mathbf{t}_1}.\,y!^{\mathbf{t}_2} \,|\, y?^{\mathbf{t}_3})$ 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 labeled with $\mathbf{t}_1$ succeeds. Since actually it never succeeds, we know that the input from y is kept waiting forever.

[2] 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.

Example 3.1.5: The process type $x?^{\mathbf{t}_1}.\,y!^{\mathbf{t}_2}$ describes a process that uses x for input *and then* uses y for output. So, the process $x?^{\mathbf{t}_1}.\,y!^{\mathbf{t}_2}$ has this type, but neither $y!^{\mathbf{t}_2}.\,x?^{\mathbf{t}_1}$ nor $x?^{\mathbf{t}_1}\,|\,y!^{\mathbf{t}_2}$ has this type. The process type $x!^{\mathbf{t}}.\,y!^{\mathbf{t}'}\&y!^{\mathbf{t}'}.\,x!^{\mathbf{t}}$ describes a process that uses x and y once for output sequentially in any order; So, both $x!^{\mathbf{t}}.\,y!^{\mathbf{t}'}$ and $y!^{\mathbf{t}'}.\,x!^{\mathbf{t}}$ can have this type.[3] The process $x!^{\mathbf{t}}\,|\,y!^{\mathbf{t}'}$, however, does not have this type.

Example 3.1.6: The type $(x,y)(x?[\tau].\,y![\tau])$ describes a pair whose first element should be first used for sending a value of type τ, and then whose second element should be used for receiving a value of type τ. The type $\mu\alpha.(x)(x?^{\mathbf{t}}[\tau].\,\alpha\langle x\rangle)$ describes a channel that is used for receiving a tuple of type τ repeatedly. The type $\mu\alpha.(x)(x?^{\mathbf{t}}[\alpha])$ describes a channel that is used for receiving a channel of the same type (so, the received channel is again used for receiving a channel of the same type).

We define the set of free (channel) variables in a process type as follows. Note that $\mathbf{FV}(\Gamma)$ is sometimes different from the set of variables appearing free syntactically: For example, x is not an element of $\mathbf{FV}(((y)\mathbf{0})\langle x\rangle)$. Intuitively, $\mathbf{FV}(\Gamma)$ denotes the set of free variables whose renaming changes the meaning of Γ. For example, x is in the set $\mathbf{FV}(x?^{\mathbf{t}})$ since renaming of x with a different variable (y, for example) changes the meaning of the process type. On the other hand, x is not an element of $\mathbf{FV}(((z)\mathbf{0})\langle x\rangle)$ since $((z)\mathbf{0})\langle x\rangle$ is essentially equivalent to $\mathbf{0}$ (with respect to the subtyping relation introduced later) and renaming of x with y does not change its meaning.

Definition 3.1.7: Let Γ be a process type. The set $\mathbf{FV}(\Gamma)$ of variables is defined by:

$$
\begin{aligned}
\mathbf{FV}(\mathbf{0}) &= \emptyset\\
\mathbf{FV}(x?^{\mathbf{t}}[\tau].\,\Gamma) &= \{x\}\cup\mathbf{FV}(\tau)\cup\mathbf{FV}(\Gamma)\\
\mathbf{FV}(x!^{\mathbf{t}}[\tau].\,\Gamma) &= \{x\}\cup\mathbf{FV}(\tau)\cup\mathbf{FV}(\Gamma)\\
\mathbf{FV}(\mathbf{t}.\Gamma) &= \mathbf{FV}(\Gamma)\\
\mathbf{FV}(\gamma_1+\cdots+\gamma_n) &= \mathbf{FV}(\gamma_1)\cup\cdots\cup\mathbf{FV}(\gamma_n)\\
\mathbf{FV}(\Gamma_1\,|\,\Gamma_2) &= \mathbf{FV}(\Gamma_1)\cup\mathbf{FV}(\Gamma_2)\\
\mathbf{FV}(\Gamma_1\&\Gamma_2) &= \mathbf{FV}(\Gamma_1)\cup\mathbf{FV}(\Gamma_2)\\
\mathbf{FV}(\tau\langle\tilde{x}\rangle) &= \mathbf{FV}(\tau)\cup\{x_i\mid i\in\mathbf{FVI}(\tau)\}\\
\mathbf{FV}(\alpha) &= \emptyset\\
\mathbf{FV}((\tilde{x})\Gamma) &= \mathbf{FV}(\Gamma)\backslash\{\tilde{x}\}\\
\mathbf{FV}(\mu\alpha.\tau) &= \mathbf{FV}(\tau)\\
\mathbf{FVI}(\alpha) &= \emptyset\\
\mathbf{FVI}((\tilde{x})\Gamma) &= \{i\mid x_i\in\mathbf{FV}(\Gamma)\}\\
\mathbf{FVI}(\mu\alpha.\tau) &= \mathbf{FVI}(\tau)
\end{aligned}
$$

We define operations on types. The operation $\Gamma{\downarrow_S}$ defined below 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.

Definition 3.1.8: Let S be a subset of $\mathbf{Var}\cup\mathbf{Nat}$. Then, unary operations $\cdot{\downarrow_S}$ and $\cdot{\uparrow_S}$ on semi-closed tuple types and process types are defined by:

$$\mathbf{0}{\downarrow_S} = \mathbf{0}$$

[3] 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.

$$
\begin{array}{rcl}
(x?^{\mathbf{t}}[\tau].\,\Gamma){\downarrow_S} & = & \left\{\begin{array}{ll} x?^{\mathbf{t}}[\tau].\,(\Gamma{\downarrow_S}) & \text{if } x \in S \\ \mathbf{t}.(\Gamma{\downarrow_S}) & \text{otherwise} \end{array}\right. \\
(x!^{\mathbf{t}}[\tau].\,\Gamma){\downarrow_S} & = & \left\{\begin{array}{ll} x!^{\mathbf{t}}[\tau].\,(\Gamma{\downarrow_S}) & \text{if } x \in S \\ \mathbf{t}.(\Gamma{\downarrow_S}) & \text{otherwise} \end{array}\right. \\
(\mathbf{t}.\Gamma){\downarrow_S} & = & \mathbf{t}.(\Gamma{\downarrow_S}) \\
(\gamma_1 + \cdots + \gamma_n){\downarrow_S} & = & (\gamma_1{\downarrow_S}) + \cdots + (\gamma_n{\downarrow_S}) \\
(\Gamma_1 \,|\, \Gamma_2){\downarrow_S} & = & (\Gamma_1{\downarrow_S}) \,|\, (\Gamma_2{\downarrow_S}) \\
(\Gamma_1 \& \Gamma_2){\downarrow_S} & = & (\Gamma_1{\downarrow_S}) \& (\Gamma_2{\downarrow_S}) \\
(\tau\langle x_1, \ldots, x_n\rangle){\downarrow_S} & = & (\tau{\downarrow_{S \cup S'}})\langle x_1, \ldots, x_n\rangle \\
 & & \quad \text{where } S' = \{i \mid x_i \in S\} \\
\alpha{\downarrow_S} & = & \alpha \\
((x_1, \ldots, x_n)\Gamma){\downarrow_S} & = & (x_1, \ldots, x_n)(\Gamma{\downarrow_{S'}}) \\
 & & \text{where } S' = (S \backslash \mathbf{Nat}) \cup \{x_i \mid i \in S\} \\
(\mu\alpha.\tau){\downarrow_S} & = & \mu\alpha.(\tau{\downarrow_S}) \\
\Gamma{\uparrow_S} & = & \Gamma{\downarrow_{\mathbf{Var}\backslash S}}
\end{array}
$$

Example 3.1.9: Let $\Gamma = y?^{\mathbf{t}}[\tau].\,x!^{\mathbf{t}'}[\tau']$. Then $\Gamma{\downarrow_{\{x\}}} = \mathbf{t}.x!^{\mathbf{t}'}[\tau']$ and $\Gamma{\uparrow_{\{x\}}} = y?^{\mathbf{t}}[\tau].\,\mathbf{t}'$.

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 [16, 50], 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.

Definition 3.2.1 [subtyping]: A preorder $\leq$ on process types and tuple 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.

Some explanation of the rules for recursive types would be required. The rule (Sub-Unfold) allows some occurences of a recursive type variable α to be replaced by its definition. The rule (Sub-Rec) is the same as the contraction rule of Amadio and Cardelli [1], except for the definition of contractiveness. The standard rule for unfolding recursive types:

$$\mu\alpha.\tau \cong [\mu\alpha.\tau/\alpha]\tau$$

can be obtained from (Sub-Unfold) and (Sub-Rec). The rule (Sub-Rep) is a degenerated case of the following standard rule:

$$\frac{\Sigma, \alpha \leq \alpha' \triangleright \tau \leq \tau'}{\Sigma \triangleright \mu\alpha.\tau \leq \mu\alpha'.\tau'}$$

Here, Σ is a sequence of assumptions of the form $\alpha_1 \leq \alpha_2$. It is possible to replace (Sub-Rep) with the above rule. Then, (Sub-Rep) is derivable using this rule as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{\triangleright \Gamma \leq \Gamma' \quad \alpha \leq \alpha' \triangleright \alpha \leq \alpha'}{\cdots}}{\alpha \leq \alpha' \triangleright (\,)(\Gamma \,|\, \alpha\langle\,\rangle) \leq (\,)(\Gamma' \,|\, \alpha'\langle\,\rangle)}}{\triangleright \mu\alpha.(\,)(\Gamma \,|\, \alpha\langle\,\rangle) \leq \mu\alpha'.(\,)(\Gamma' \,|\, \alpha'\langle\,\rangle)}}{\triangleright (\mu\alpha.(\,)(\Gamma \,|\, \alpha\langle\,\rangle))\langle\,\rangle \leq (\mu\alpha'.(\,)(\Gamma' \,|\, \alpha'\langle\,\rangle))\langle\,\rangle}$$

In addition to the rules in Figure 3, we sometimes use the following axiom in examples.

$$\frac{\Gamma \text{ is closed (i.e., } \Gamma \text{ contains no free type variables)}}{(\Gamma{\downarrow_S} \,|\, \Gamma{\uparrow_S}) \leq \Gamma} \qquad (\text{SUB-DIVIDE})$$

This axiom is not required for type soundness to hold, but it makes more processes to be typed. The rule allows us to forget information about dependencies between some channels. For example, if $\Gamma = y?^{\mathbf{t}}[\tau].\, x!^{\mathbf{t}'}[\tau']$, then $\Gamma{\downarrow_{\{x\}}} \,|\, \Gamma{\uparrow_{\{x\}}} = \mathbf{t}.x!^{\mathbf{t}'}[\tau'] \,|\, y?^{\mathbf{t}}[\tau].\, \mathbf{t}'$ is a subtype of Γ. Note that the closedness of Γ is required since, without that condition, we would get $\alpha\langle\,\rangle \,|\, \alpha\langle\,\rangle \leq \alpha\langle\,\rangle$. 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

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

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 in the case for reductions of processes, we annotate each reduction with information about 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 $L \in \mathbf{T} \cup \{x^{\mathbf{t}_1,\mathbf{t}_2} \mid (x \in \mathbf{Var}) \wedge (\mathbf{t}_1, \mathbf{t}_2 \in \mathbf{T})\} \cup \{\epsilon^{\mathbf{t}_1,\mathbf{t}_2} \mid \mathbf{t}_1, \mathbf{t}_2 \in \mathbf{T}\}$) is the least relation closed under the rules in Figure 4.

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

The rule (TER-COM2) is used to simulate communication on inner channels. For example, the process reduction:

$$(\nu x)\,(x!^{\mathbf{t}_1} \,|\, x?^{\mathbf{t}_2}) \xrightarrow{\epsilon^{\mathbf{t}_1,\mathbf{t}_2}} (\nu x)\,\mathbf{0}$$

is simulated by:

$$\mathbf{t}_1 \,|\, \mathbf{t}_2 \xrightarrow{\epsilon^{\mathbf{t}_1,\mathbf{t}_2}} \mathbf{0}.$$

The rule (TER-EV) allows a process type of the form $\mathbf{t}.\Gamma$ to be reduced by itself. This is used to model communication of a process with an unknown environment. This rule is necessary to reason about behavior of a process in a compositional manner. For example, consider a process:

$$(\nu x)\,(y?^{\mathbf{t}_1}.\, x!^{\mathbf{t}_2}[1] \,|\, x?^{\mathbf{t}_3}[y].\, z!^{\mathbf{t}_4}[y+1]),$$

$$\Gamma \,|\, \mathbf{0} \cong \Gamma \qquad \text{(Sub-Empty)}$$

$$\Gamma_1 \,|\, \Gamma_2 \cong \Gamma_2 \,|\, \Gamma_1 \qquad \text{(Sub-Commut)}$$

$$\Gamma_1 \,|\, (\Gamma_2 \,|\, \Gamma_3) \cong (\Gamma_1 \,|\, \Gamma_2) \,|\, \Gamma_3 \qquad \text{(Sub-Assoc)}$$

$$\frac{[\alpha/\beta]\tau' = \tau}{\mu\alpha.\tau \cong \mu\alpha.[\mu\alpha.\tau/\beta]\tau'} \qquad \text{(Sub-Unfold)}$$

$$\frac{[\tau'/\alpha]\tau \cong \tau' \qquad \tau \text{ is contractive in } \alpha}{\tau' \cong \mu\alpha.\tau} \qquad \text{(Sub-Rec)}$$

$$((\tilde{x})\Gamma)\langle\tilde{y}\rangle \cong [\tilde{y}/\tilde{x}]\Gamma \qquad \text{(Sub-Beta)}$$

$$\frac{\Gamma \,\leq\, \Gamma'}{(\tilde{x})\Gamma \,\leq\, (\tilde{x})\Gamma'} \qquad \text{(Sub-Abs)}$$

$$\frac{\tau \,\leq\, \tau'}{\tau\langle\tilde{x}\rangle \,\leq\, \tau'\langle\tilde{x}\rangle} \qquad \text{(Sub-App)}$$

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

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

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

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

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

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

Figure 3: Necessary Conditions on Subtyping Relation

$$\frac{\tau_1 \leq \tau_2}{\cdots + x!^{\mathbf{t}_1}[\tau_1].\,\Gamma_1 + \cdots \mid \cdots + x?^{\mathbf{t}_2}[\tau_2].\,\Gamma_2 + \cdots \xrightarrow{x^{\mathbf{t}_1,\mathbf{t}_2}} \Gamma_1 \mid \Gamma_2} \quad \text{(TER-Com1)}$$

$$\cdots + \mathbf{t}_1.\Gamma_1 + \cdots \mid \cdots + \mathbf{t}_2.\Gamma_2 + \cdots \xrightarrow{\epsilon^{\mathbf{t}_1,\mathbf{t}_2}} \Gamma_1 \mid \Gamma_2 \quad \text{(TER-Com2)}$$

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

$$\frac{\Gamma_1 \xrightarrow{L} \Gamma_1'}{\Gamma_1 \mid \Gamma_2 \xrightarrow{L} \Gamma_1' \mid \Gamma_2} \quad \text{(TER-Par)}$$

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

Figure 4: Reduction of Process Types

which communicates with an environment through channels y and z. To check that the channel x is used in a consistent manner (without looking at the environment), we drop information about y and z and check the behavior of the following type:

$$\mathbf{t}_1.x!^{\mathbf{t}_2}[int].\mid x?^{\mathbf{t}_3}[int].\,\mathbf{t}_4.$$

By reducing the type:

$$\mathbf{t}_1.x!^{\mathbf{t}_2}[int].\mid x?^{\mathbf{t}_3}[int].\,\mathbf{t}_4 \xrightarrow{\mathbf{t}_1} x!^{\mathbf{t}_2}[int].\mid x?^{\mathbf{t}_3}[int].\,\mathbf{t}_4 \xrightarrow{x^{\mathbf{t}_2,\mathbf{t}_3}} \mathbf{t}_4 \xrightarrow{\mathbf{t}_4} \mathbf{0},$$

we find that there is no wrong communication on x.

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 instead. We write *ok* for the corresponding property of process types and call it a *consistency condition*. The consistency condition depends on the property that 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 5.

The following well-formedness condition requires that there is no disagreement about communication between input and output processes. For example, the process type $x?[string] \mid x![int]$ is ill-formed, since it specifies that a sub-process is waiting to receive a string along x and another sub-process is trying to send an integer along the same channel.

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

Remark 3.4.2: It is possible to 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 make type systems flexible. 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:

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 $ok(\Gamma_1 \,|\, \Gamma_2)$

In the rest of this paper, we assume that *ok* always refers to a proper consistency predicate. We say that a process type Γ is *consistent* if $WF(\Gamma)$ holds.

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 [30], we can introduce the rule $\Gamma \,|\, \Gamma \cong \Gamma$ so that the reductions of a process type can be reduced to a finite-state machine. If we take $\leq$ to be the least proper subtyping relation, however, we need to use a more complex system like Petri nets, as in the case for our previous type system for deadlock-freedom [31].

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 $|, +,$ or $*$ can be obtained by composing abstractions of its subprocesses with the corresponding constructor for 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), the lefthand assumption means that P uses free channels according to $\Gamma_1 \,|\, \Gamma_2$, and the righthand assumption means that information about the usage of received channels $\tilde{y}$ is not in Γ_2 but in Γ_1. In the conclusion, the information about the usage of $\tilde{y}$ is put into the tuple type $(\tilde{y})\Gamma_2$. Information about the usage of other channels is put into either the tuple type $(\tilde{y})\Gamma_2$, so that the usage of those channels is taken into account by the output process, or the continuation Γ_1, so that the usage of those channels is taken into account by this input process. For example, consider a process $x?^{\mathbf{t}_1}[y].\, y?^{\mathbf{t}_2}.\, z!^{\mathbf{t}_3}$. Its subprocess $y?^{\mathbf{t}_2}.\, z!^{\mathbf{t}_3}$ has the process type $y?^{\mathbf{t}_2}.\, z!^{\mathbf{t}_3}$. By applying (T-Sub) and (Sub-Divide), we obtain the following type judgment:

$$\mathbf{t}_2.z!^{\mathbf{t}_3} \,|\, y?^{\mathbf{t}_2}.\, \mathbf{t}_3 \triangleright y?^{\mathbf{t}_2}.\, z!^{\mathbf{t}_3}$$

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

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

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

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

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

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

$$\frac{\Gamma_1 \mid \Gamma_2 \triangleright P \qquad \mathbf{FV}(\Gamma_1) \cap \{\tilde{y}\} = \emptyset}{x?^{\mathbf{t}}[(\tilde{y})\Gamma_2].\,\Gamma_1 \triangleright x?^{\mathbf{t}}[\tilde{y}].\,P} \qquad \text{(T-In)}$$

$$\frac{\Gamma \triangleright P \qquad ok(\Gamma{\downarrow_{\{\tilde{x}\}}}) \qquad \mathbf{FV}(\Gamma{\uparrow_{\{\tilde{x}\}}}) \cap \{\tilde{x}\} = \emptyset}{\Gamma{\uparrow_{\{\tilde{x}\}}} \triangleright (\nu\tilde{x})\,P} \qquad \text{(T-New)}$$

Figure 5: Typing Rules

By applying (T-In), we obtain:

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

The parameter type $(y)y?^{\mathbf{t}_2}.\,\mathbf{t}_3$ of the channel x carries information that y is used for input and then some event $\mathbf{t}_3$ occurs. On the other hand, the continuation part $\mathbf{t}_2.z!^{\mathbf{t}_3}$ says that some event $\mathbf{t}_2$ occurs after the input on x, and then z is used for output. Alternatively, we can obtain the following type judgment:

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

This judgment means that y is used for input and then z is used for output.

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 about the use of $\tilde{x}$ by $\cdot{\uparrow_{\{\tilde{x}\}}}$. The condition $\mathbf{FV}(\Gamma{\uparrow_{\{\tilde{x}\}}}) \cap \{\tilde{x}\} = \emptyset$ ensures that x is no longer visible from the outside.

The rules (T-Out) and (T-In) are asymmetric in the sense that information about the continuation of a receiver process is transfered to a sender process but not vice versa. This design choice is motivated by the observation that the names of the channels transmitted via a communication are statically known only to the sender, so that we must put information about how the transmitted channels are used by the receiver into the type of the sender. For example, consider a process containing $x?[y].\,y!$ and $x![z]$ as sub-processes. Since the name z may not be in the scope of the sub-process $x?[y].\,y!$, we have to put information that z is used for output into the type of $x![z]$, as $x![\tau].\,z!$. An extension to treat input and output processes in a more symmetric manner is discussed in Section 8.

Example 3.5.1: Consider a process $x?^{\mathbf{t}_1}[y].\, l?^{\mathbf{t}_2}.\, y!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4}$. After receiving a channel y on x, it receives a null tuple on l, sends a null tuple on y, and sends a null tuple on l. (If l is used as a lock, "receiving a value on l" and "sending a value on l" are interpreted as "acquiring lock l" and "releasing lock l" respectively: See Section 6.) The parallel composition of the above process with $x!^{\mathbf{t}_5}[z]$ is typed as follows:

$$\dfrac{\dfrac{\dfrac{\cdots}{l?^{\mathbf{t}_2}.\, y!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4} \triangleright l?^{\mathbf{t}_2}.\, y!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4}}}{x?^{\mathbf{t}_1}[\tau] \triangleright x?^{\mathbf{t}_1}[y].\, l?^{\mathbf{t}_2}.\, y!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4}} \qquad \dfrac{\cdots}{x!^{\mathbf{t}_5}[\tau].\, l?^{\mathbf{t}_2}.\, z!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4} \triangleright x!^{\mathbf{t}_5}[z]}}{x?^{\mathbf{t}_1}[\tau] \mid x!^{\mathbf{t}_5}[\tau].\, l?^{\mathbf{t}_2}.\, z!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4} \triangleright x?^{\mathbf{t}_1}[y].\, l?^{\mathbf{t}_2}.\, y!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4} \mid x!^{\mathbf{t}_5}[z]}$$

In the derivation above, τ denotes $(y)l?^{\mathbf{t}_2}.\, y!^{\mathbf{t}_3}.\, l!^{\mathbf{t}_4}$. The conclusion implies that after a communication on x, lock l is acquired, a null tuple is sent on z, and then l is released.

Example 3.5.2: Consider a process $x!^{\mathbf{t}_1}[x] \mid *x?^{\mathbf{t}_2}[y].\, y!^{\mathbf{t}_3}[y]$. Let τ be $\mu\alpha.(x)(x!^{\mathbf{t}_3}[\alpha].\, \alpha\langle x\rangle)$ and τ' be its expansion: $(x)(x!^{\mathbf{t}_3}[\tau].\, \tau\langle x\rangle)$. Then, the process is typed as follows:

$$\dfrac{x!^{\mathbf{t}_1}[\tau'].\, \tau'\langle x\rangle \triangleright x!^{\mathbf{t}_1}[x] \qquad \dfrac{\dfrac{\dfrac{y!^{\mathbf{t}_3}[\tau].\, \tau\langle y\rangle \triangleright y!^{\mathbf{t}_3}[y]}{\mathbf{0} \mid y!^{\mathbf{t}_3}[\tau].\, \tau\langle y\rangle \triangleright y!^{\mathbf{t}_3}[y]}}{x?^{\mathbf{t}_2}[\tau'] \triangleright x?^{\mathbf{t}_2}[y].\, y!^{\mathbf{t}_3}[y]}}{*x?^{\mathbf{t}_2}[\tau'] \triangleright *x?^{\mathbf{t}_2}[y].\, y!^{\mathbf{t}_3}[y]}}{x!^{\mathbf{t}_1}[\tau'].\, \tau'\langle x\rangle \mid *x?^{\mathbf{t}_2}[\tau'] \triangleright x!^{\mathbf{t}_1}[x] \mid *x?^{\mathbf{t}_2}[y].\, y!^{\mathbf{t}_3}[y]}$$

4 Type Soundness

The general type system given above is parameterized by the subtyping relation $\leq$ and the consistency predicate *ok*, which determine the exact properties of each instance of the type system. Therefore, proofs of type soundness also depend on each instance. As we show below, however, several important properties can be proved independently of a choice of the subtyping relation and the consistency predicate. In Section 4.1, we show a collection of basic properties of the generic type system. These properties imply that the behavior of a process is simulated by its type in a certain sense, so that we can check properties of a process by checking the corresponding properties of the process's type.

Since the properties in Section 4.1 are rather low-level, some work is still necessary to prove the type soundness of each instance of the generic type system (though the work would be much easier than proving soundness of each type system from scratch). In Section 4.2, we show a more "high-level" theorem about type soundness. From the theorem, we can automatically obtain type soundness of a certain class of instances of the generic type system.

4.1 Basic Properties of the Generic Type System

Type Preservation

First, Theorem 4.1.1 below guarantees that if $\Gamma \triangleright P$ holds, for every reduction of P, there is a corresponding reduction of Γ and the reduced process has the reduced process type. In this sense the behavior of a well-typed process is modeled by its process type.

Theorem 4.1.1 [subject reduction]: If $\Gamma \triangleright P$ and $P \stackrel{l}{\longrightarrow} P'$ with $\mathit{WF}(\Gamma)$, then there exists Γ' such that $\Gamma \stackrel{l}{\longrightarrow} \Gamma'$ and $\Gamma' \triangleright P'$.

Proof: See Appendix A.1. $\square$

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

Theorem 4.1.2: 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 4.1.1 and the fact that ok is preserved by reduction (of process types). $\square$

Normalization of Type Derivation

The normal derivation theorem given below states that, from any type derivation, it is possible to obtain a "syntax-directed" type derivation of the same conclusion. It 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 4.1.3 [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. $\square$

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

1. If $y \notin \{\tilde{x_1}, \ldots, \tilde{x_k}\}$, then $\Gamma \leq \cdots + y?^{\mathbf{t}}[\tau].\,\Gamma_1 + \cdots \,|\, \Gamma_2$ for some τ, Γ_1, and Γ_2.
2. If $y \in \{\tilde{x_1}, \ldots, \tilde{x_k}\}$, then $\Gamma \leq \cdots + \mathbf{t}.\Gamma_1 + \cdots \,|\, \Gamma_2$ 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 4.1.5:

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

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

4.2 General Type Soundness Theorem

In Section 4.1, we have shown that a process satisfies a certain property p if its process type satisfies the *corresponding* consistency condition ok. However, it is 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. There also 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 does not carry information about 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.

We first introduce logical formulas [2, 45] to formally state properties of processes and types.

Definition 4.2.6: The set **Prop** of formulas is given by the following syntax:

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

Here, the meta-variable ζ ranges over the set **IVar** of variables (called *integer variables*), disjoint from **Var**. The meta-variables n, m_i, n_i range over the union of the set **IVar** and the set **Nat** of non-negative integers. We often abbreviate $\neg(\neg A \vee \neg B)$ to $A \wedge B$.

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. A formula $A \,|\, B$ means that the process is parallel composition of a process satisfying A and another process satisfying B. A formula $\langle l \rangle A$ means that the process can be reduced in one step to a process satisfying A and that the reduction is labeled with l. A formula $ev(A)$ means that the process can be reduced to a process satisfying A in a finite number of steps.[4] For example, the formula

$$\neg \exists \mathbf{t}, \mathbf{t}'.\exists \zeta_1, \zeta_2 \!:\! \emptyset.ev(x!^{\mathbf{t}}\zeta_1 \,|\, x!^{\mathbf{t}'}\zeta_2)$$

means that no two processes try to output a value on x simultaneously.

We define the formal semantics of formulas below. As usual, $\exists x.A$, $\exists \mathbf{t}.A$, and $\exists \zeta\!:\!C.A$ bind x, $\mathbf{t}$, and ζ respectively. We write $\mathbf{FIV}(A)$ for the set of free integer variables in A. We define substitutions for variables, events, and integer variables in a customary manner. For example, $[n/\zeta]A$ denotes the formula obtained from A by substituting n for all free occurrences of ζ. We write $\mathbf{FV}(A)$ for the set of free variables (in **Var**), and $\mathbf{FIV}(A)$ for the set of free integer variables (in **IVar**).

Definition 4.2.7: The size of a formula A, written $size(A)$, is defined by:

$size(\mathtt{true}) = size(x!^{\mathbf{t}}n) = size(x?^{\mathbf{t}}n) = 1$
$size(A \,|\, B) = size(A \vee B) = size(A) + size(B) + 1$
$size(\langle l \rangle A) = size(ev(A)) = size(\neg A) = size(\exists x.A) = size(\exists \mathbf{t}.A) = size(\exists \zeta\!:\!C.A) = size(A) + 1$

We define the semantics $[\![A]\!]_{\mathbf{pr}}$ of a formula A as the set of processes satisfying A. To define it, we introduce an auxiliary function $[\![A]\!]^S_{\mathbf{pr}}$, which denotes the set of processes satisfying A provided that all channels in S are invisible.

[4] It is possible to introduce a general fixed-point operator [45] instead, but we did not do so in this paper for simplicity.

Definition 4.2.8: Let A be a formula such that $\mathbf{FIV}(A) = \emptyset$, and let S be a finite set of variables. The sets $[\![A]\!]_{\mathbf{pr}}$ and $[\![A]\!]^S_{\mathbf{pr}}$ of processes are defined by (**Proc** is the set of processes):

$$
\begin{array}{rcl}
[\![A]\!]_{\mathbf{pr}} & = & [\![A]\!]^{\emptyset}_{\mathbf{pr}} \\
[\![\mathtt{true}]\!]^S_{\mathbf{pr}} & = & \mathbf{Proc} \\
[\![x!^{\mathbf{t}}n]\!]^S_{\mathbf{pr}} & = & \{P \mid P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x!^{\mathbf{t}}[\tilde{y}].\,Q + \cdots \mid R), x \notin S \cup \{\tilde{x_1}, \ldots, \tilde{x_k}\}, \|\tilde{y}\| = n\} \\
[\![x?^{\mathbf{t}}n]\!]^S_{\mathbf{pr}} & = & \{P \mid P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x?^{\mathbf{t}}[\tilde{y}].\,Q + \cdots \mid R), x \notin S \cup \{\tilde{x_1}, \ldots, \tilde{x_k}\}, \|\tilde{y}\| = n\} \\
[\![A \mid B]\!]^S_{\mathbf{pr}} & = & \{P \mid P \preceq (\widetilde{\nu x}_{1..k})\,(Q \mid R), Q \in [\![A]\!]^{S\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}}_{\mathbf{pr}}, R \in [\![B]\!]^{S\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}}_{\mathbf{pr}}\} \\
[\![\langle l\rangle A]\!]^S_{\mathbf{pr}} & = & \{P \mid P \xrightarrow{l'} Q, l = l'\!\uparrow_S, Q \in [\![A]\!]^S_{\mathbf{pr}}\} \\
[\![ev(A)]\!]^S_{\mathbf{pr}} & = & \{P \mid P \longrightarrow^* Q, Q \in [\![A]\!]^S_{\mathbf{pr}}\} \\
[\![\neg A]\!]^S_{\mathbf{pr}} & = & \mathbf{Proc}\backslash[\![A]\!]^S_{\mathbf{pr}} \\
[\![A \vee B]\!]^S_{\mathbf{pr}} & = & [\![A]\!]^S_{\mathbf{pr}} \cup [\![B]\!]^S_{\mathbf{pr}} \\
[\![\exists x.A]\!]^S_{\mathbf{pr}} & = & \displaystyle\bigcup_{y\in\mathbf{Var}} [\![[y/x]A]\!]^S_{\mathbf{pr}} \\
[\![\exists \mathbf{t}.A]\!]^S_{\mathbf{pr}} & = & \displaystyle\bigcup_{\mathbf{t}'\in\mathbf{T}} [\![[\mathbf{t}'/\mathbf{t}]A]\!]^S_{\mathbf{pr}} \\
[\![\exists\zeta : C.A]\!]^S_{\mathbf{pr}} & = & \{P \mid P \in [\![[n/\zeta]A]\!]^S_{\mathbf{pr}}, n \in \mathbf{Nat}, \models [n/\zeta]C\}
\end{array}
$$

Here, $\models C$ means that all the inequalities $i \neq j$ hold. We write $P \models A$ when $P \in [\![A]\!]_{\mathbf{pr}}$ holds.

It is easy to prove that $[\![A]\!]_{\mathbf{pr}}$ and $[\![A]\!]^S_{\mathbf{pr}}$ are well defined, by induction on the size of A.

We show some properties of $[\![A]\!]^S_{\mathbf{pr}}$.

Lemma 4.2.9: If $(\nu\tilde{x})\,P \xrightarrow{l} Q$, then there exists Q' such that $P \xrightarrow{l'} Q'$ with $(\nu\tilde{x})\,Q' \preceq Q$ and $l'\!\uparrow_{\{\tilde{x}\}} = l$.

Proof: This follows by straightforward induction on derivation of $(\nu\tilde{x})\,P \xrightarrow{l} Q$. □

Lemma 4.2.10: If $P \preceq Q$, then $P \in [\![A]\!]^S_{\mathbf{pr}}$ if and only if $Q \in [\![A]\!]^S_{\mathbf{pr}}$.

Proof: This lemma follows by induction on the size of A. □

Lemma 4.2.11: Suppose that $P \in \mathbf{Proc}$, $A \in \mathbf{Prop}$, and $S \cup \{\tilde{x}\} \subseteq \mathbf{Var}$. Suppose also that $S \cap \{\tilde{x}\} = \emptyset$. Then, $(\nu\tilde{x})\,P \in [\![A]\!]^S_{\mathbf{pr}}$ holds if and only if $P \in [\![A]\!]^{S\cup\{\tilde{x}\}}_{\mathbf{pr}}$ holds.

Proof: The proof proceeds by induction on the size of A. We show only main cases; The other cases are similar or trivial.

- Case $A = x!^{\mathbf{t}}n$: the required property follows by:

$$
\begin{array}{rcl}
(\nu\tilde{x})\,P \in [\![A]\!]^S_{\mathbf{pr}} & \iff & ((\nu\tilde{x})\,P \preceq (\nu\tilde{x})\,(\widetilde{\nu x}_{1..k})\,(\cdots + x!^{\mathbf{t}}[\tilde{y}].\,Q + \cdots \mid R)) \\
& & \wedge(x \notin S \cup \{\tilde{x}\} \cup \{\tilde{x_1}, \ldots, \tilde{x_k}\}) \wedge \|\tilde{y}\| = n \\
& \iff & (P \preceq (\widetilde{\nu x}_{1..k})\,(\cdots + x!^{\mathbf{t}}[\tilde{y}].\,Q + \cdots \mid R)) \\
& & \wedge(x \notin S \cup \{\tilde{x}\} \cup \{\tilde{x_1}, \ldots, \tilde{x_k}\}) \wedge \|\tilde{y}\| = n \\
& \iff & P \in [\![A]\!]^{S\cup\{\tilde{x}\}}_{\mathbf{pr}}
\end{array}
$$

- Case $A = A_1 \,|\, A_2$: the required property follows by:

$$\begin{array}{rcl}
(\nu\tilde{x})\,P \in [\![A]\!]_{\mathbf{pr}}^{S} & \iff & (\nu\tilde{x})\,P \preceq (\nu\tilde{x})\,(\widetilde{\nu x}_{1..k})\,(P_1 \,|\, P_2) \\
& & \wedge P_1 \in [\![A_1]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}} \wedge P_2 \in [\![A_2]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}} \\
& \iff & P \preceq (\widetilde{\nu x}_{1..k})\,(P_1 \,|\, P_2) \\
& & \wedge P_1 \in [\![A_1]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}} \wedge P_2 \in [\![A_2]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}} \\
& \iff & P \in [\![A]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}}
\end{array}$$

- Case $A = \langle l\rangle B$: Suppose $(\nu\tilde{x})\,P \in [\![A]\!]_{\mathbf{pr}}^{S}$. Then, there exists Q such that $(\nu\tilde{x})\,P \xrightarrow{l'} Q$ with $l'{\uparrow_S} = l$ and $Q \in [\![B]\!]_{\mathbf{pr}}^{S}$. By Lemma 4.2.9, there exists Q' such that $P \xrightarrow{l''} Q'$ with $(\nu\tilde{x})\,Q' \preceq Q$ and $l''{\uparrow_{\{\tilde{x}\}}} = l'$. By Lemma 4.2.10 and $Q \in [\![B]\!]_{\mathbf{pr}}^{S}$, we have $(\nu\tilde{x})\,Q' \in [\![B]\!]_{\mathbf{pr}}^{S}$. By induction hypothesis, we have $Q' \in [\![B]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}}$. Since $l''{\uparrow_{S\cup\{\tilde{x}\}}} = l'{\uparrow_S} = l$, we have $P \in [\![A]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}}$ as required.

  On the other hand, suppose $P \in [\![A]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}}$, with $S\cap\{\tilde{x}\} = \emptyset$. Then, there exists Q and l' such that $P \xrightarrow{l'} Q$ with $l'{\uparrow_{S\cup\{\tilde{x}\}}} = l$ and $Q \in [\![B]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}}$. By rule (R-New), we have $(\nu\tilde{x})\,P \xrightarrow{l'\uparrow_{\{\tilde{x}\}}} (\nu\tilde{x})\,Q$. By applying the induction hypothesis to $Q \in [\![B]\!]_{\mathbf{pr}}^{S\cup\{\tilde{x}\}}$, we obtain $(\nu\tilde{x})\,Q \in [\![B]\!]_{\mathbf{pr}}^{S}$. Since $(l'{\uparrow_{\{\tilde{x}\}}}){\uparrow_S} = l'{\uparrow_{S\cup\{\tilde{x}\}}} = l$, we have $(\nu\tilde{x})\,P \in [\![A]\!]_{\mathbf{pr}}^{S}$ as required. □

A formula is interpreted also as a property of process types (**Type** is the set of process types):

Definition 4.2.12: Let A be a formula such that $\mathbf{FIV}(A) = \emptyset$, and The set $[\![A]\!]_{\mathbf{ty}}$ and $[\![A]\!]_{\mathbf{ty}}^{S}$ of process types are defined by:

$$\begin{array}{rcl}
[\![\mathtt{true}]\!]_{\mathbf{ty}} & = & \mathbf{Type} \\
[\![x!^{\mathbf{t}}n]\!]_{\mathbf{ty}} & = & \{\Gamma \mid \Gamma \leq \cdots + x!^{\mathbf{t}}[\tau].\,\Delta_1 + \cdots \mid \Delta_2, \|\tau\| = n\} \\
[\![x?^{\mathbf{t}}n]\!]_{\mathbf{ty}} & = & \{\Gamma \mid \Gamma \leq \cdots + x?^{\mathbf{t}}[\tau].\,\Delta_1 + \cdots \mid \Delta_2, \|\tau\| = n\} \\
[\![A \,|\, B]\!]_{\mathbf{ty}} & = & \{\Gamma \mid \Gamma \leq (\Delta_1 \,|\, \Delta_2), \Delta_1 \in [\![A]\!]_{\mathbf{ty}}, \Delta_2 \in [\![B]\!]_{\mathbf{ty}}\} \\
[\![\langle l\rangle A]\!]_{\mathbf{ty}} & = & \{\Gamma \mid \Gamma \xrightarrow{l} \Delta, \Delta \in [\![A]\!]_{\mathbf{ty}}\} \\
[\![ev(A)]\!]_{\mathbf{ty}} & = & \{\Gamma \mid \Gamma \longrightarrow^{*} \Delta, \Delta \in [\![A]\!]_{\mathbf{ty}}\} \\
[\![\neg A]\!]_{\mathbf{ty}} & = & \mathbf{Type}\backslash[\![A]\!]_{\mathbf{ty}} \\
[\![A \vee B]\!]_{\mathbf{ty}} & = & [\![A]\!]_{\mathbf{ty}} \cup [\![B]\!]_{\mathbf{ty}} \\
[\![\exists x.A]\!]_{\mathbf{ty}} & = & \bigcup_{y\in\mathbf{Var}} [\![[y/x]A]\!]_{\mathbf{ty}} \\
[\![\exists \mathbf{t}.A]\!]_{\mathbf{ty}} & = & \bigcup_{\mathbf{t}'\in\mathbf{T}} [\![[\mathbf{t}'/\mathbf{t}]A]\!]_{\mathbf{ty}} \\
[\![\exists\zeta : C.A]\!]_{\mathbf{ty}} & = & \{\Gamma \mid \Gamma \in [\![[n/\zeta]A]\!]_{\mathbf{ty}}, n \in \mathbf{Nat}, \models [n/\zeta]C\} \\
[\![A]\!]_{\mathbf{ty}}^{S} & = & \{\Gamma \mid \Gamma{\uparrow_S} \in [\![A]\!]_{\mathbf{ty}}\}
\end{array}$$

We write $\Gamma \models A$ when $\Gamma \in [\![A]\!]_{\mathbf{ty}}$ holds.

To formally define what property of process types *corresponds to* a property of processes, we introduce two binary relations on formulas.

$$x!^{\mathbf{t}}n \hookrightarrow x!^{\mathbf{t}}n \qquad \text{(F-Out)}$$

$$x?^{\mathbf{t}}n \hookrightarrow x?^{\mathbf{t}}n \qquad \text{(F-In)}$$

$$\frac{A \hookrightarrow B}{\langle l \rangle A \hookrightarrow \langle l \rangle B} \qquad \text{(F-Red)}$$

$$\frac{A \hookrightarrow B}{ev(A) \hookrightarrow ev(B)} \qquad \text{(F-Ev)}$$

$$\frac{A_1 \hookrightarrow B_1 \qquad A_2 \hookrightarrow B_2}{A_1 \,|\, A_2 \hookrightarrow B_1 \,|\, B_2} \qquad \text{(F-Par)}$$

$$\frac{A \rightarrowtail B}{\neg B \hookrightarrow \neg A} \qquad \text{(F-Neg1)}$$

$$\frac{A \hookrightarrow B}{\neg B \rightarrowtail \neg A} \qquad \text{(F-Neg2)}$$

Figure 6: Inference Rules for Proving $A \rightarrowtail B$ and $A \hookrightarrow B$

Notation 4.2.13: We write $\mathcal{GS}(A)$ for the set of substitutions that map all the free integer variables in A to non-negative integers.

Definition 4.2.14: Binary relations $\rightarrowtail$ and $\hookrightarrow$ on formulas are defined by:

$$\begin{aligned}
A \rightarrowtail B \iff & \forall \Gamma \in \mathbf{Type}, P \in \mathbf{Proc}, S \subseteq \mathbf{Var}, \theta \in \mathcal{GS}(A \,|\, B) \\
& ((\Gamma \triangleright P \wedge WF(\Gamma) \wedge \Gamma \in [\![\theta A]\!]^S_{\mathbf{ty}}) \Rightarrow P \in [\![\theta B]\!]^S_{\mathbf{pr}}) \\
A \hookrightarrow B \iff & \forall \Gamma \in \mathbf{Type}, P \in \mathbf{Proc}, S \subseteq \mathbf{Var}, \theta \in \mathcal{GS}(A \,|\, B). \\
& ((\Gamma \triangleright P \wedge WF(\Gamma) \wedge P \in [\![\theta A]\!]^S_{\mathbf{pr}}) \Rightarrow \Gamma \in [\![\theta B]\!]^S_{\mathbf{ty}})
\end{aligned}$$

Intuitively, $A \rightarrowtail B$ means that if a process type satisfies the property A, every process of that type satisfies the property B. Conversely, $A \hookrightarrow B$ means that if a process satisfies A, its type satisfies B.

We obtain sound inference rules to derive the relations $A \rightarrowtail B$ and $A \hookrightarrow B$ as shown in Figure 6. We have also standard logical rules like:

$$\frac{A \rightarrowtail B}{A \rightarrowtail B \vee C} \qquad\qquad \frac{A \rightarrowtail C \qquad B \rightarrowtail C}{A \vee B \rightarrowtail C}$$

The soundness of inference rules is proved as follows.

Lemma 4.2.15: The inference rules in Figure 6 are sound.

Proof: Without loss of generality, we can assume that formulas do not contain free integer variables. The soundness of the rules (F-Out) and (F-In) follow immediately from Corollary 4.1.4.

To show the soundness of (F-Red), suppose that $A \hookrightarrow B$, $P \in [\![\langle l \rangle A]\!]^S_{\mathbf{pr}}$, and $\Gamma \triangleright P$. By $P \in [\![\langle l \rangle A]\!]^S_{\mathbf{pr}}$, there must exist Q and l' such that $P \xrightarrow{l'} Q$ with $l' {\uparrow_S} = l$ and $Q \in [\![A]\!]^S_{\mathbf{pr}}$. By Theorem 4.1.1, there exists Δ such that $\Gamma \xrightarrow{l'} \Delta$ and $\Delta \triangleright Q$. By Lemma A.1.3 in Appendix, we have $\Gamma{\uparrow_S} \xrightarrow{l} \Delta{\uparrow_S}$. By the assumption $A \hookrightarrow B$, we have $\Delta \in [\![B]\!]^S_{\mathbf{ty}}$, that is, $\Delta{\uparrow_S} \in [\![B]\!]_{\mathbf{ty}}$. So, we have $\Gamma{\uparrow_S} \in [\![\langle l\rangle B]\!]_{\mathbf{ty}}$, that is, $\Gamma \in [\![\langle l\rangle B]\!]^S_{\mathbf{ty}}$ as required. The soundness of (F-Ev) follows similarly.

To show the soundness of (F-Par), suppose $A_1 \hookrightarrow B_1$, $A_2 \hookrightarrow B_2$, $P \in [\![A_1 \,|\, A_2]\!]^S_{\mathbf{pr}}$, and $\Gamma \triangleright P$ with $WF(\Gamma)$. By the assumption $P \in [\![A_1 \,|\, A_2]\!]^S_{\mathbf{pr}}$, there exists $P_1, P_2, \tilde{x_1}, \ldots, \tilde{x_k}$ such that

$$\begin{array}{l} P \preceq (\widetilde{\nu x}_{1..k})\,(P_1 \,|\, P_2) \\ P_1 \in [\![A_1]\!]^{S\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}}_{\mathbf{pr}} \\ P_2 \in [\![A_2]\!]^{S\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}}_{\mathbf{pr}} \end{array}$$

By Lemma A.1.6, we have $\Gamma \triangleright (\widetilde{\nu x}_{1..k})\,(P_1 \,|\, P_2)$. By Theorem 4.1.3, we have

$$\begin{array}{l} \Delta_1 \triangleright P_1 \\ \Delta_2 \triangleright P_2 \\ \Gamma \,\leq\, \Delta_1{\uparrow_{\{\tilde{x_1},\ldots,\tilde{x_n}\}}} \,|\, \Delta_2{\uparrow_{\{\tilde{x_1},\ldots,\tilde{x_n}\}}} \\ WF((\Delta_1 \,|\, \Delta_2){\downarrow_{\{\tilde{x}_i\}}}) \text{ for } i = 1, \ldots, k \end{array}$$

Note that $WF(\Delta_1)$ and $WF(\Delta_2)$ follows from the last two conditions and $WF(\Gamma)$. So, by the assumptions $A_1 \hookrightarrow B_1$ and $A_2 \hookrightarrow B_2$, we have $\Delta_1 \in [\![B_1]\!]^{S\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}}_{\mathbf{ty}}$ and $\Delta_2 \in [\![B_2]\!]^{S\cup\{\tilde{x_1},\ldots,\tilde{x_k}\}}_{\mathbf{ty}}$, which implies $\Delta_1{\uparrow_{\{\tilde{x_1},\ldots,\tilde{x_n}\}}} \in [\![B_1]\!]^S_{\mathbf{ty}}$ and $\Delta_2{\uparrow_{\{\tilde{x_1},\ldots,\tilde{x_n}\}}} \in [\![B_2]\!]^S_{\mathbf{ty}}$. By $\Gamma \,\leq\, \Delta_1{\uparrow_{\{\tilde{x_1},\ldots,\tilde{x_n}\}}} \,|\, \Delta_1{\uparrow_{\{\tilde{x_1},\ldots,\tilde{x_n}\}}}$, we have $\Gamma \in [\![B_1 \,|\, B_2]\!]^S_{\mathbf{ty}}$ as required.

To show the soundness of (F-Neg1), suppose that $A \rightarrowtail B$, $P \in [\![\neg B]\!]^S_{\mathbf{pr}}$, and $\Gamma \triangleright P$. It suffices to show $\Gamma \not\in [\![A]\!]^S_{\mathbf{ty}}$. Suppose $\Gamma \in [\![A]\!]^S_{\mathbf{ty}}$. Then, by $A \rightarrowtail B$, it must be the case that $P \in [\![B]\!]^S_{\mathbf{pr}}$. But this contradicts with the assumption $P \in [\![\neg B]\!]^S_{\mathbf{pr}} = \mathbf{Proc}\backslash[\![B]\!]^S_{\mathbf{pr}}$. The proof of the soundness of (F-Neg2) is similar. $\square$

By using Lemma 4.2.15, 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. Therefore, our type system can be used at least for reasoning about properties described using negative formulas.

Definition 4.2.16 [positive/negative formulas]: The set $\mathcal{F}^+$ ($\mathcal{F}^-$, resp.) of positive (negative, resp.) formulas are the least set satisfying the following rules:

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

Intuitively, a formula is positive (negative, resp.) if sub-formulas of the form $A \,|\, B$, $\langle l\rangle A$, $ev(A)$, $x!^{\mathbf{t}}n$, $x?^{\mathbf{t}}n$ appear only in positions where the negation is applied an even (odd, resp.) number of times.

Theorem 4.2.17: Suppose $\Gamma \triangleright P$ and $WF(\Gamma)$. For any $S \subseteq \mathbf{Var}$, if $A \in \mathcal{F}^-$ with $\mathbf{FIV}(A) = \emptyset$ and $\Gamma \in [\![A]\!]^S_{\mathbf{ty}}$, then $P \in [\![A]\!]^S_{\mathbf{pr}}$ holds. Conversely, for any $S \subseteq \mathbf{Var}$, if $A \in \mathcal{F}^+$ with $\mathbf{FIV}(A) = \emptyset$ and $P \in [\![A]\!]^S_{\mathbf{pr}}$, then $\Gamma \in [\![A]\!]^S_{\mathbf{ty}}$ holds.

Proof: The proof proceeds by induction on the size of A. The cases for $x!^{\mathbf{t}}n$, $x?^{\mathbf{t}}n$, $A_1 \,|\, A_2$, $\langle l \rangle A'$, $ev(A')$, and $\neg A'$ follow immediately from Lemma 4.2.15.

Suppose $A = A_1 \vee A_2 \in \mathcal{F}^-$ and $\Gamma \in [\![A]\!]_{\mathbf{ty}}$. By the definition of $\mathcal{F}^-$, it must be the case that $A_1, A_2 \in \mathcal{F}^-$. By the assumption $\Gamma \in [\![A]\!]_{\mathbf{ty}}$, $\Gamma \in [\![A_i]\!]_{\mathbf{ty}}$ for $i = 1$ or 2. By using induction hypothesis, we obtain $P \in [\![A_i]\!]_{\mathbf{pr}} \subseteq [\![A]\!]_{\mathbf{pr}}$ as required. The other cases are similar. □

Lemma 4.2.18: If $\{\tilde{x}\} \cap \mathbf{FV}(P) = \emptyset$, then $P \in [\![A]\!]_{\mathbf{pr}}$ if and only if $(\nu \tilde{x})\, P \in [\![A]\!]_{\mathbf{pr}}$.

Proof: This follows by straightforward induction on the structure of A. □

Corollary 4.2.19 [type soundness]: Let $A \in \mathcal{F}^-$ and P be a closed process. Suppose that $ok(\Gamma)$ implies $\Gamma \in [\![A]\!]_{\mathbf{ty}}$ for any $\Gamma \in \mathbf{Type}$. Suppose also that $\Gamma \triangleright P$ and $ok(\Gamma)$. Then, if $P \longrightarrow^* \preceq (\widetilde{\nu x})\, (\nu \tilde{y})\, Q$, then $(\widetilde{\nu x})\, Q \in [\![A]\!]_{\mathbf{pr}}$.

Proof: By applying Theorem 4.1.1 to $\Gamma \triangleright P$ and $P \longrightarrow^* \preceq (\widetilde{\nu x}_{1..k})\, (\nu \tilde{y})\, Q$, we have $\Gamma' \triangleright (\widetilde{\nu x}_{1..k})\, (\nu \tilde{y})\, Q$ for some Γ' with $ok(\Gamma')$. By Theorem 4.1.3, we have Δ such that $\Delta \triangleright Q$ with $\Gamma' \leq \Delta \uparrow_{\{\tilde{x_1}, \ldots, \tilde{x_k}, \tilde{y}\}}$ and $ok(\Delta \downarrow_{\{\tilde{y}\}})$. By the assumption, we have $\Delta \downarrow_{\{\tilde{y}\}} \in [\![A]\!]_{\mathbf{ty}}$. Let $\{\tilde{w}\} = \mathbf{FV}(\Delta) \backslash \{\tilde{x_1}, \ldots, \tilde{x_k}, \tilde{y}\}$. Since $\Delta \downarrow_{\{\tilde{y}\}} = \Delta \uparrow_{\{\tilde{x_1}, \ldots, \tilde{x_k}, \tilde{w}\}}$, we have $\Delta \in [\![A]\!]_{\mathbf{ty}}^{\{\tilde{x_1}, \ldots, \tilde{x_k}, \tilde{w}\}}$. By using Theorem 4.2.17, we obtain $Q \in [\![A]\!]_{\mathbf{pr}}^{\{\tilde{x_1}, \ldots, \tilde{x_k}, \tilde{w}\}}$, which implies $(\nu \tilde{w})\, (\widetilde{\nu x}_{1..k})\, Q \in [\![A]\!]_{\mathbf{pr}}$ by Lemma 4.2.11. By Lemma 4.2.18, we have $(\widetilde{\nu x}_{1..k})\, Q \in [\![A]\!]_{\mathbf{pr}}$ as required. □

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 $ok(\Gamma) \iff WF(\Gamma) \wedge \Gamma \models inv(A)$ (where $inv(A) = \neg ev(\neg A)$).

Using the above corollary, we can obtain various type systems. For example, we can obtain a variant of the linear channel type system [30]. Let A be $\neg \exists x. \exists \mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3. \exists \zeta. ev(\langle x^{\mathbf{t}_1, \mathbf{t}_2} \rangle ev(x!^{\mathbf{t}_3} \zeta \vee x?^{\mathbf{t}_3} \zeta))$; Then it is guaranteed that every channel is used at most once. More examples are given in Section 5.

Note that our generic type system can be used also for reasoning about properties described by some non-negative formulas: Indeed, the deadlock-freedom property is not described as a non-negative formula, but as we show in Section 5, we can obtain a type system for deadlock-freedom as an instance of our generic type system and show its soundness (by using not the above corollary but more basic theorems presented in Section 4).

5 Applications

We show that a variety of type systems — those for arity-mismatch check, race detection, static garbage-channel collection, and deadlock detection, — can indeed be obtained as instances of the generic type system. Thanks to the common properties shown in Section 4, only a small amount of extra work is necessary to define each instance and prove its correctness.

Table 1 shows the invariant properties of processes that should be guaranteed by those type systems. 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 it is not the case that an input process labeled with $\mathbf{t}$ can be reduced and the same channel is used for input or output after that. The invariance of that property means that after a channel has been used for an input action annotated with $\mathbf{t}$, the channel is no longer used. So, it is safe to deallocate the channel

$p_1(P)$	$P \preceq (\widetilde{\nu w})\, P'$ implies $P' \models \neg\exists x.\exists \mathbf{t}_1,\mathbf{t}_2.\exists\zeta_1,\zeta_2:\{\zeta_1 \neq \zeta_2\}.(x!^{\mathbf{t}_1}\zeta_1 \mid x?^{\mathbf{t}_2}\zeta_2)$
$p_2(P)$	$P \preceq (\widetilde{\nu w})\, P'$ implies $P' \models \neg\exists x.\exists \mathbf{t}_1.\exists\zeta_1,\zeta_2.(x!^{\mathbf{t}}\zeta_1 \mid x!^{\mathbf{t}_1}\zeta_2)$
$p_3(P)$	$P \preceq (\widetilde{\nu w})\, P'$ implies $P' \models \neg\exists x.\exists \mathbf{t}_1,\mathbf{t}_2.\exists\zeta.\langle x^{\mathbf{t}_1,\mathbf{t}}\rangle ev(x!^{\mathbf{t}_2}\zeta \vee x?^{\mathbf{t}_2}\zeta)$
$p_4(P)$	$P \preceq (\widetilde{\nu w})\, P'$ implies $P' \models \neg\exists x,\exists\zeta.((x!^{\mathbf{t}}\zeta \vee x?^{\mathbf{t}}\zeta) \wedge \neg\exists y.\exists \mathbf{t}_1,\mathbf{t}_2.(\langle y^{\mathbf{t}_1,\mathbf{t}_2}\rangle \mathtt{true} \vee \langle \epsilon^{\mathbf{t}_1,\mathbf{t}_2}\rangle \mathtt{true}))$

Table 1: Properties of Processes

$ok_1(\Gamma)$	$WF(\Gamma)$
$ok_2(\Gamma)$	$WF(\Gamma)$ and $\Gamma \models \neg ev(\exists x.\exists \mathbf{t}_1.\exists\zeta_1,\zeta_2:\{\zeta_1 \neq \zeta_2\}.(x!^{\mathbf{t}}\zeta_1 \mid x!^{\mathbf{t}_1}\zeta_2))$
$ok_3(\Gamma)$	$WF(\Gamma)$ and $\Gamma \models \neg ev(\exists x.\exists \mathbf{t}_1,\mathbf{t}_2.\exists\zeta.\langle x^{\mathbf{t}_1,\mathbf{t}}\rangle ev(x!^{\mathbf{t}_2}\zeta \vee x?^{\mathbf{t}_2}\zeta))$
$ok_4(\Gamma)$	$WF(\Gamma)$ and $\Gamma \models \neg ev(\exists \mathbf{t}'.\exists x.\exists\zeta.(\mathbf{t}' \preccurlyeq \mathbf{t} \wedge (x!^{\mathbf{t}'}\zeta \vee x?^{\mathbf{t}'}\zeta)$ $\wedge \neg\exists y.\exists \mathbf{t}_1,\mathbf{t}_2.(\langle y^{\mathbf{t}_1,\mathbf{t}_2}\rangle \mathtt{true} \vee (\langle \mathbf{t}_1\rangle \mathtt{true} \wedge \mathbf{t}_1 \prec \mathbf{t}'))))$.

Table 2: Consistency Conditions

after the action annotated with $\mathbf{t}$. For example, the process $(\nu x)\,(x!^{\mathbf{t}_1} \mid x!^{\mathbf{t}_2} \mid x?^{\mathbf{t}_3}.\, x?^{\mathbf{t}})$ satisfies this property. $p_4(P)$ 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 reduced further.

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 reduction of the abstract process Γ. $ok_3(\Gamma)$ 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. $ok_4(\Gamma)$ means that whenever Γ is reduced to a process type trying to perform an action annotated with an event $\mathbf{t}_1$ less than or equal to $\mathbf{t}$, Γ can be further reduced on some channel or on an event $\mathbf{t}_2$ less than $\mathbf{t}_1$. Here, we assume that $\prec$ is some well-founded relation on events, and $\mathbf{t}_1 \preccurlyeq \mathbf{t}_2$ means $\mathbf{t}_1 \prec \mathbf{t}_2$ or $\mathbf{t}_1 = \mathbf{t}_2$. We added new formulas $\mathbf{t}_1 \preccurlyeq \mathbf{t}_2$, $\mathbf{t}_1 \prec \mathbf{t}_2$, and $\langle \mathbf{t}\rangle A$ to express properties of process types; their semantics is defined by:

$$
\begin{aligned}
[\![\mathbf{t}_1 \prec \mathbf{t}_2]\!]_{\mathbf{ty}} &= \begin{cases} \mathbf{Type} & \text{if } \mathbf{t}_1 \prec \mathbf{t}_2 \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\mathbf{t}_1 \preccurlyeq \mathbf{t}_2]\!]_{\mathbf{ty}} &= \begin{cases} \mathbf{Type} & \text{if } \mathbf{t}_1 \preccurlyeq \mathbf{t}_2 \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\langle \mathbf{t}\rangle A]\!]_{\mathbf{ty}} &= \{\Gamma \mid \Gamma \xrightarrow{\mathbf{t}} \Gamma' \wedge \Gamma' \in [\![A]\!]_{\mathbf{ty}}\}
\end{aligned}
$$

Soundness of the first three type systems follows immediately from Corollary 4.2.19.

Theorem 5.1: Let ok be ok_i ($i \in \{1,2,3\}$). If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_i(Q)$ holds for every Q such that $P \longrightarrow^* \preceq Q$.

Proof:

- Case for $i = 1$: Suppose that $\Gamma \triangleright P$, $ok(\Gamma)$, and $P \longrightarrow^* \preceq Q$. Let A be:

$$\neg ev(\exists x.\exists \mathbf{t}_1,\mathbf{t}_2.\exists\zeta_1,\zeta_2:\{\zeta_1 \neq \zeta_2\}.(x!^{\mathbf{t}_1}\zeta_1 \mid x?^{\mathbf{t}_2}\zeta_2)).$$

  To show $p_1(Q)$, it is sufficient to show that $Q \preceq (\widetilde{\nu w})\,(\nu \tilde{u})\, Q'$ implies $(\widetilde{\nu w})\, Q' \models A$. Since $ok_1(\Gamma)$ implies $\Gamma \models A$ and A is a negative formula, Corollary 4.2.19 implies the sufficient condition.

- Case for $i = 2$: Since ok_2 is an invariant condition, ok_2 is a proper consistency predicate. Suppose that $\Gamma \triangleright P$, $ok(\Gamma)$, and $P \longrightarrow^* \preceq Q$. Let A be:

$$\neg ev(\neg\exists x.\exists \mathbf{t}_1.\exists\zeta_1,\zeta_2:\{\zeta_1 \neq \zeta_2\}.(x!^{\mathbf{t}}\zeta_1 \mid x!^{\mathbf{t}_1}\zeta_2)).$$

To show $p_2(Q)$, it is sufficient to show that $Q \preceq (\widetilde{\nu w})\,(\nu \tilde{u})\,Q'$ implies $(\widetilde{\nu w})\,Q' \models A$. Since $ok_2(\Gamma)$ implies $\Gamma \models A$ and A is a negative formula, Corollary 4.2.19 implies the sufficient condition.

- Case for $i = 3$: Similar to the case for $i = 2$.

□

In the theorem above, $\leq$ can be any proper subtyping relation. Choosing an appropriate subtyping relation for each type system would simplify type-checking or type-reconstruction. For example, we can identify $\mathbf{t}.\Gamma$ with Γ by the rule $\mathbf{t}.\Gamma \cong \Gamma$, except for the case $i = 4$. For a naive arity-mismatch check [16, 50], we can ignore the order of communications by introducing rules like $x?^{\mathbf{t}}[\tau].\,\Gamma \cong x?^{\mathbf{t}}[\tau] \,|\, \Gamma$.

For the type system for deadlock-freedom ($ok = ok_4$), we need to choose a particular subtyping relation. Let $\leq_1$ be the least proper subtyping relation and $\leq_2$ be the least subtyping relation closed under the rules in Figure 3 with Sub-Divide. Then, deadlock-freedom holds for both relations:

Theorem 5.2: Let ok be ok_4 and $\leq$ be $\leq_i$ ($i \in \{1, 2\}$). If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_4(Q)$ holds for every Q such that $P \longrightarrow^* \preceq Q$.

We cannot use Corollary 4.2.19 to prove the theorem above, because $p_4(P)$ is described using a non-negative formula. Using more basic theorems shown in Section 4, however, we can easily prove the above theorem. By Theorem 4.1.2, it suffices to show that $\Gamma \triangleright P$ and $ok_4(\Gamma)$ imply $p_4(P)$, by using Theorems 4.1.3–4.1.5: See Appendix A.2 for a proof.

The following examples indicate that our framework not only has many of the existing type systems as instances but also can express more expressive type systems than them (see also Section 6).

Example 5.3: The process $x?[y].\,x? \,|\, x![w].\,x!$ is well typed in the first ($i = 1$) type system. So, unlike in earlier type systems for arity-mismatch check [16, 50], the same channel can be used for communicating values of different types.[5]

Example 5.4: The second type system guarantees that the process

$$(\nu l, x)\,(l!^{\mathbf{t}_0} \,|\, {*l?^{\mathbf{t}_1}}.\,x!^{\mathbf{t}}.\,l!^{\mathbf{t}_3} \,|\, x?^{\mathbf{t}_4})$$

is race-free on the channel x. So, unlike the linear type system [30] for the π-calculus, our type system can guarantee lack of race conditions even on channels that are used more than once.[6]

Example 5.5: The fourth type system (for deadlock-freedom) rejects the process

$$P = (\nu x)\,(\nu y)\,(x?^{\mathbf{t}}.\,y!^{\mathbf{t}_1} \,|\, y?^{\mathbf{t}_2}.\,x!^{\mathbf{t}_3}).$$

The type of the sub-process $x?^{\mathbf{t}}.\,y!^{\mathbf{t}_1} \,|\, y?^{\mathbf{t}_2}.\,x!^{\mathbf{t}_3}$ is $x?^{\mathbf{t}}.\,y!^{\mathbf{t}_1} \,|\, y?^{\mathbf{t}_2}.\,x!^{\mathbf{t}_3}$. So, in order for P to be well-typed, the following constraints must be satisfied:

$$ok_4(x?^{\mathbf{t}}.\,\mathbf{t}_1 \,|\, \mathbf{t}_2.x!^{\mathbf{t}_3})$$
$$ok_4(\mathbf{t}.y!^{\mathbf{t}_1} \,|\, y?^{\mathbf{t}_2}.\,\mathbf{t}_3)$$

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.

[5] Yoshida's type system [52] also allows such use of channels.

[6] Flanagan and Abadi's type system [10] also gives such a guarantee. Since their target calculus has locks as primitives, the problem is a little simpler.

Remark 5.6: 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 $\Gamma_1 \cong \Gamma_2$ means $\Gamma_1 \le \Gamma_2 \wedge \Gamma_2 \le \Gamma_1$.) 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

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

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 $(\Gamma\!\downarrow_S \,|\, \Gamma\!\uparrow_S) \cong \Gamma$. The last rule removes information about the order of communications between different channels. A type environment of the linear π-calculus [30] is obtained by further removing information about channel usage, by adding the rules $x!^{\mathbf{t}}[\tau].\,\Gamma \cong x!^{\mathbf{t}}[\tau] \,|\, \Gamma$, $x?^{\mathbf{t}}[\tau].\,\Gamma \cong x?^{\mathbf{t}}[\tau] \,|\, \Gamma$, and $\Gamma \,|\, \Gamma \cong *\Gamma$. A type environment of the input-only/output-only channel type system [40] is obtained by further adding the rule $*\Gamma \cong \Gamma$.

6 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) [26, 32, 42], 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 invoked services, it is possible to guarantee race-freedom of accesses to methods, studied by Abadi, Flanagan and Freund [10, 12], and deadlock-freedom for objects with non-uniform service availability, studied by Puntigam [43]. 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 how to enforce 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 and a method m to print out the string "Hello, " appended to a given string, and exports its interface $[l, m]$ through the reply channel r.

$$\begin{array}{l} *newob?\,[r].\,(\nu l, m)\,(\\ \quad l!^{\mathbf{t}'} \,|\, *m?^{\mathbf{t}'}[s, r'].\, print!^{\mathbf{t}'}[\text{“Hello, ”}].\, print!^{\mathbf{t}'}[s].\, r'!^{\mathbf{t}'} \\ \quad |\, r![l, m]) \end{array}$$

Since the method m should not be invoked concurrently,[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 $[l, m]$ (through which clients access to the object) can be given a type:

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

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

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 $\mathbf{t}.\Gamma \cong \Gamma$ and that *String* is some unary tuple type; the notation $(s : \mathit{String}, r)\Gamma$ stands for $(s, r)(\mathit{String}\langle s\rangle \mid \Gamma)$.) The process type $l?^{\mathbf{t}'}.\, m!^{\mathbf{t}}[(s : \mathit{String}, r)(r!^{\mathbf{t}'}[()l!^{\mathbf{t}'}])]$ means that (1) the lock must be acquired (by an input from l) before the method is invoked (by an output to m); (2) the method argument must be a pair of string s and a reply channel r; and (3) the lock must be released (by an output to l) but only after the reply from the object is sent to r. Note that the tuple type $(s : \mathit{String}, r)(r!^{\mathbf{t}'}[()l!^{\mathbf{t}'}])$ refers to the lock l as a free variable, making it possible to express temporal dependency between l, which is *not* passed to the method body, and the reply channel r.

Then, a client

$$\begin{array}{l}(\nu r_1)\,(l?^{\mathbf{t}'}.\, m!^{\mathbf{t}}[\text{“Atsushi”}, r_1].\, r_1?^{\mathbf{t}'}.\, l!^{\mathbf{t}'})\\ \mid (\nu r_2)\,(l?^{\mathbf{t}'}.\, m!^{\mathbf{t}}[\text{“Naoki”}, r_2].\, r_2?^{\mathbf{t}'}.\, l!^{\mathbf{t}'})\end{array}$$

is well typed, but neither

$$(\nu r_3)\,(m!^{\mathbf{t}}[\text{“Atsushi”}, r_3].\, r_3?^{\mathbf{t}'}.\, P)$$

nor

$$(\nu r_4)\,(l?^{\mathbf{t}'}.\, m!^{\mathbf{t}}[\text{“Naoki”}, r_4].\,(r_4?^{\mathbf{t}'} \mid l!^{\mathbf{t}'}))$$

is. 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 [10], which means that the method m can be invoked only after the lock on the object is acquired. Unfortunately, our type system is less expressive than theirs in some respects. As the example shows, channels representing a lock l and a method m guarded by the lock must be created at once; Otherwise, our type system will lose dependency among those channels. See Section 8 for a possible remedy against the problem.

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_4, 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:

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

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 *put*! and *get*!) while $(\nu r')\,\mathit{get}!^{\mathbf{t}}[r'].\, r'?[x].\, \mathit{put}!^{\mathbf{t}}[v].\,\cdots$ is not.

7 Type Checking/Reconstruction

By using Theorem 4.1.3, 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 process type with a set of subtype constraints and consistency conditions on the type. The pair of a process type and a set of constraints is principal in the sense that a process is typeable if and only if the set of constraints output by the algorithm is satisfiable. Indeed, the algorithm presented here is essentially the same as the first half of an existing type reconstruction algorithm [31].

To complete a type-check/reconstruction algorithm, it will be only required to develop an algorithm to solve constraints on process types. Since the definitions of a subtype relation and a consistency condition vary, such algorithms have to be developed for each instance. Some of them, which have prototype implementations, can be found in the literature [24, 31].

In this section, we mainly discuss the common part: the definition of principal typings and an algorithm to compute them.

7.1 Principal Typings

We first define the notion of principal typings in our type system. A principal typing is defined as a pair of a process type including *process type variables* and a set of constraints on the process type variables; all the possible process types, under which the process expression is well typed, are obtained from the process type in the pair, by replacing process type variables in the constraint so that the constraint is satisfied.

To define principal typings, we introduce a countably infinite set of process type variables, ranged over by ρ, and extend the definition of process types with the process type variables and the expressions $\Gamma{\downarrow_{\{x_1,\ldots,x_n\}}}$ and $\Gamma{\uparrow_{\{x_1,\ldots,x_n\}}}$.

Definition 7.1.1 [extended process type]: The set of process types is extended as follows.

$$\Gamma ::= \cdots \mid \rho \mid \Gamma{\downarrow_{\{x_1,\ldots,x_n\}}} \mid \Gamma{\uparrow_{\{x_1,\ldots,x_n\}}}$$

The metavariable C denotes a set of constraints of the form $\Gamma_1 \leq \Gamma_2$, $\mathbf{FV}(\Gamma) \cap \{\tilde{x}\} = \emptyset$ or $ok(\Gamma)$.

Notation 7.1.2: We write θ for a substitution of (non-extended) process types and tuple types for process type variables and tuple type variables, respectively. We write $\mathbf{FTV}(\Gamma)$ and $\mathbf{FTV}(C)$ for the sets of (process and tuple) type variables appearing free in Γ and C, respectively.

Then, principal typings are defined as follows.

Definition 7.1.3 [principal typing]: A pair (Γ, C) of an extended process type and a set of constraints is a *principal typing* of a process P if it satisfies the following conditions:

1. If θ is chosen so that $dom(\theta) \supseteq \mathbf{FTV}(\Gamma) \cup \mathbf{FTV}(C)$ and θC is satisfied, then $\theta\Gamma \triangleright P$.

2. If $\Gamma' \triangleright P$, then there exists a substitution θ such that θC and $\Gamma' \leq \theta\Gamma$ hold.

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

$PT(\mathbf{0}) = (\emptyset, \emptyset)$

$PT(P_1 \,|\, P_2) =$
let $(\Gamma_1, C_1) = PT(P_1)$
$(\Gamma_2, C_2) = PT(P_2)$
in $(\Gamma_1 \,|\, \Gamma_2, C_1 \cup C_2)$

$PT(*P_0) =$
let $(\Gamma_0, C_0) = PT(P_0)$
in $(*\Gamma_0, C_0)$

$PT(P_1 + \cdots + P_n) =$
let $(\Gamma_1, C_1) = PT(P_1)$
$\vdots$
$(\Gamma_n, C_n) = PT(P_n)$
in $(\Gamma_1 + \cdots + \Gamma_n, C_1 \cup \cdots \cup C_n)$

$PT(x!^{\mathbf{t}}[\tilde{z}].\, P_0) =$
let $(\Gamma_0, C_0) = PT(P_0)$
in $(x!^{\mathbf{t}}[\alpha^{(n)}].\, (\rho \,|\, \alpha^{(n)}\langle\tilde{z}\rangle), C_0 \cup \{\rho \,\leq\, \Gamma_0\})$
(where α and ρ are fresh)

$PT(x?^{\mathbf{t}}[\tilde{y}].\, P_0) =$
let $(\Gamma_0, C_0) = PT(P_0)$
in $(x?^{\mathbf{t}}[\alpha^{(n)}].\, \rho, C_0 \cup \{(\rho \,|\, \alpha^{(n)}\langle\tilde{y}\rangle) \,\leq\, \Gamma_0, \mathbf{FV}(\rho) \cap \{\tilde{y}\} = \emptyset\})$
(where α and ρ are fresh)

$PT((\nu\tilde{x})\, P_0) =$
let $(\Gamma_0, C_0) = PT(P_0)$
in $(\Gamma_0{\uparrow_{\{\tilde{x}\}}}, C_0 \cup \{ok(\Gamma_0{\downarrow_{\{\tilde{x}\}}}), \mathbf{FV}(\Gamma_0{\uparrow_{\{\tilde{x}\}}}) \cap \{\tilde{x}\} = \emptyset\})$

Figure 7: Algorithm PT

7.2 Algorithm to Compute Principal Typings

Thanks to Theorem 4.1.3, it is easy to derive syntax-directed typing rules, by combining T-SUB with T-OUT and T-IN from the original ones. Then, by reading the syntax-directed typing rules in a bottom-up manner, we can obtain a principal typing algorithm, which takes a process expression as an input and outputs its principal typing. The algorithm PT is shown in Figure 7.

Theorem 7.2.1: If $PT(P) = (\Gamma, C)$, then (Γ, C) is a principal typing of P.

Proof: Structural induction on P. □

8 Discussions

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

Limitations caused by the formalization of ok To obtain common properties useful for proving type soundness (in Section 4), 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 [11].) 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. Because of this restriction, unlike Abadi and Flanagan's type system for concurrent objects, we cannot deal with the case where an object and its lock are separately created.

One way to remove the restrictions mentioned above would be to add type annotation of the form $(\nu\tilde{x}\!:\!\Gamma)\,P$ to the channel creation primitive. Here, Γ specifies how newly created channels $\tilde{x}$ should be used in combination with other channels. For example, $(\nu x\!:\, \star\, l?^{\mathbf{t}_1}[\,].\,x!^{\mathbf{t}_2}[\tau].\,l!^{\mathbf{t}_3}[\,]\,|\,{*x?^{\mathbf{t}_4}[\tau]})\,P$ means that a new channel x is created and that whenever x is used for output, the lock l is acquired beforehand and l is released afterwards. This extension, however, requires a substantial amount of change in the type system and the proof of subject reduction. First, we need to replace the typing rule for channel creation with something like:

$$\frac{\Gamma_1\,|\,\Gamma_2 \triangleright P \qquad WF(\Gamma_1{\downarrow_{\{\tilde{x}\}}}) \qquad \mathbf{FV}(\Gamma_1{\uparrow_{\{\tilde{x}\}}}\,|\,\Gamma_2) \cap \{\tilde{x}\} = \emptyset}{\Gamma_1{\uparrow_{\{\tilde{x}\}}}\,|\,\Gamma_2 \triangleright (\nu\tilde{x}\!:\!\Gamma_1)\,P} \qquad \text{(T-NEW')}$$

The above rule allows Γ_1 to specify causality between new channels $\tilde{x}$ and other channels. Since Γ_1 may change during reductions, we also need to fix the reduction semantics so that the subject reduction property holds. For example, rule (R-New) should be replaced by the following rule:

$$\frac{P \xrightarrow{x_i^{\mathbf{t}_1,\mathbf{t}_2}} Q \qquad x_i \in \{\tilde{x}\} \qquad \Delta \leq \Delta' \text{ for all } \Delta' \text{ such that } \Gamma \xrightarrow{x_i^{\mathbf{t}_1,\mathbf{t}_2}} \Delta'}{(\nu \tilde{x} : \Gamma)\, P \xrightarrow{\epsilon^{\mathbf{t}_1,\mathbf{t}_2}} (\nu \tilde{x} : \Delta)\, Q}$$

(We show only the case for $x_i \in \{\tilde{x}\}$; The case for $x_i \notin \{\tilde{x}\}$ is uglier.) It is left for future work to formalize these extensions elegantly.

Symmetric treatment of input and output processes As remarked in Section 3.5, the rules (T-In) and (T-Out) are asymmetric in the sense that information about the continuation of a receiver process is transfered to a sender process but not vice versa. Sometimes it is useful to propagate information in the reverse direction. For example, consider the following process:

$$(\nu sync)\, (x?.\, sync? \mid sync!.\, x!)$$

The two sub-processes synchronize on channel *sync*, so that x is first used for input and then used for output. In order to obtain such information, we need to put into the type of process $sync?$ information that x is used for output after the synchronization (so that the process $x?.\, sync?$ has a type of the form $x?.\, sync?[\tau].\, x!$). That is achieved by represent input process types and output process types in the form $x?^{\mathbf{t}}[\tau_I; \tau_O].\, \Gamma$ and $x!^{\mathbf{t}}[\tau_I; \tau_O].\, \Gamma$, where τ_I describes information about the continuation of an input process (which was already present in the type system in Section 3) and τ_O describes information about the continuation of an output process. The new typing rules for output and input are:

$$\frac{\Gamma_1 \mid \Gamma_3 \triangleright P}{x!^{\mathbf{t}}[(\tilde{y})\Gamma_2; ()\Gamma_3].\, (\Gamma_1 \mid [\tilde{z}/\tilde{y}]\Gamma_2) \triangleright x!^{\mathbf{t}}[\tilde{z}].\, P} \qquad \text{(T-Out-Sym)}$$

$$\frac{\Gamma_1 \mid \Gamma_2 \triangleright P \qquad \mathbf{FV}(\Gamma_1) \cap \{\tilde{y}\} = \emptyset}{x?^{\mathbf{t}}[(\tilde{y})\Gamma_2; ()\Gamma_3].\, (\Gamma_1 \mid \Gamma_3) \triangleright x?^{\mathbf{t}}[\tilde{y}].\, P} \qquad \text{(T-In-Sym)}$$

The two sub-processes in the above example is then typed as follows:

$$x?.\, sync?[\tau_1; \tau_2].\, x! \triangleright x?.\, sync?$$

$$sync![\tau_1; \tau_2] \triangleright sync!.\, x!$$

for $\tau_1 = (\,)\mathbf{0}$ and $\tau_2 = (\,)x!$.

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 [41].

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 to study even for specific type systems [30, 40, 41].

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 [37], type systems for other process calculi can also be discussed uniformly.

9 Related Work

General framework of type systems Previous proposals of a general framework [20, 33, 34] are (i) so abstract (e.g., [20, 33]) that only a limited amount of work can be shared for developing concrete type systems, and/or (ii) not general (e.g., [33, 34]) enough to account for recent advanced type systems [10, 27, 31]. Honda's work [20] aims to develop a general theory of type systems for various process calculi, while our work in the present paper focuses on a specific process calculus (the π-calculus) and aims to develop a general theory of various type systems for the π-calculus. Being viewed as a theory of type systems for the π-calculus, Honda's framework [20] seems to be more abstract and restrictive than ours. For example, his framework only deals 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. Honda [21] also developed a type theory for the π-calculus from the viewpoint of denotational semantics (as opposed to our study of the generic type system from an operational point of view); It is left for future work to study how these different approaches are related to each other. In König's framework of type systems [33], type environments do not change during reduction of a process. So, it cannot deal with dynamically changing properties like linearity [30]. Moreover, the target calculus is less expressive than the π-calculus: It cannot express dynamic creation of channels. Conchon and Pottier [4] proposes a framework of type systems for the join-calculus [14]. The main focus of their work is a general study of type inference for the join-calculus in the presence of ML-style polymorphism. Their type system only guarantees a basic type soundness property that there is no arity mismatch error.

Other type systems viewing types as processes As mentioned in Section 1, the idea of expressing types as abstract processes has been inspired by our previous type systems for the π-calculus [28, 31, 46]. The development of the generic type system has been motivated by the observation that the underlying ideas and proofs of those type systems are very similar so that most of them can be shared. Besides the genericity, a technical novelty of our generic type system compared with our previous type systems is that we can express the order of communications on different channels directly using CCS-like processes (while in previous our type systems, we used a smaller process calculus as outlined in Section 1). Thanks to this extension, as observed in Section 6, our generic type system can also be used to analyze properties of concurrent objects.

Inspired by our work, Rehof et al. [3] recently proposed a variant of our type system. In their type system, types are represented as a fragment of CCS [35] which, unlike in our type system, includes the hiding (or channel creation) operator. It is not difficult to extend our types with the hiding operator as in their type system. While the main focus of their type system is on checking whether senders and receivers on each channel agrees on a communication protocol (like in which order senders and receivers use communicated channels), the focus of our type system is on checking channel-wise behavior of processes (like whether some process suffers from a race or deadlock condition on a particular channel). This difference resulted in slightly different formalization of the type systems. In particular, they [3] chooses an open simulation relation as a specific subtyping relation and shows how to use a model checker to check the subtyping relation.

Some previous type systems also use process-like structures to express types. Yoshida's type system [52] (which guarantees a certain deadlock-freedom property) and its successors [23, 53] 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. For example, there is no graph type corresponding to the type environment $x?^{\mathbf{t}}.\, x?^{\mathbf{t}} \,|\, x!^{\mathbf{t}} \,|\, x!^{\mathbf{t}}$ of our type system.

Nielson and Nielson [38] 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 [27] for the reason), race detection, linearity analysis, etc., where the identity of a channel is important.

Process-like terms have been used as types also in type systems for deadlock-freedom [43] and related properties [44] of concurrent objects. As briefly outlined in Section 6, our type system can guarantee such properties without having concurrent objects as primitives.

Gordon and Jeffrey [17, 18] proposed a type system for checking correspondence assertions. Two primitives $\mathbf{begin}(\tilde{x})$ (for asserting the begin of a protocol) and $\mathbf{end}(\tilde{x})$ (for asserting the end of a protocol) are introduced into the π-calculus and their type system checks that every execution of $\mathbf{end}(\tilde{x})$ is preceded by one $\mathbf{begin}(\tilde{x})$. To check that property, they introduced channel types of the form $\mathbf{Ch}(y : T)[\{\tilde{x_1}\}, \ldots, \{\tilde{x_1}\}]$, which expresses information that a receiver on that channel may perform the end actions $\mathbf{end}(\tilde{x_1}), \ldots, \mathbf{end}(\tilde{x_n})$. Their type system can be almost subsumed by our generic type system: By extending the syntax of types with $\mathbf{begin}(\tilde{x})$ and $\mathbf{end}(\tilde{x})$, we can express the channel type $\mathbf{Ch}(y : T)[\{\tilde{x_1}\}, \ldots, \{\tilde{x_1}\}]$ as:

$$(x)(*x?[(y)(y : T \,|\, \mathbf{end}(\tilde{x_1}) \,|\, \cdots \,|\, \mathbf{end}(\tilde{x_n}))] \,|\, *x![(y)(y : T \,|\, \mathbf{end}(\tilde{x_1}) \,|\, \cdots \,|\, \mathbf{end}(\tilde{x_n}))]).$$

(Alternatively, if a begin or end assertion mentions only a single name, the assertion can be encoded into an ordinary output.) However, in order to automatically obtain soundness of the type system from the soundness of our generic type system, we need to extend rules for channel creation as discussed in Section 8.

Abstract interpretation As mentioned in Section 1, our generic type system can be viewed as a kind of abstract interpretation framework [5], 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.

There are some studies of abstract interpretation for the π-calculus [9, 51], but they are quite different from our generic type system. While their abstract interpretation maps a process to one abstract process and analyzes its behavior, our generic type system maps a process to multiple process types (introduced for each $(\nu\tilde{x})\,P$) and analyzes their behavior, so that the analysis is performed in a compositional manner.

Non-standard type systems for sequential languages Unlike standard type systems for functional languages, our type system keeps track of not only the shape of each value but also information about how and in which order each value (a communication channel in our case) is accessed. Non-standard type systems for analyzing such properties have been recently studied to guarantee safe usage of resources such as memory and files [6, 7, 13, 15, 25, 29]. Among them, our type systems for resource usage analysis [25, 29] have been inspired by the generic type system in the present paper.

10 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 study of a more general version of the type soundness theorem in Section 4.2, and extensions of our generic type system discussed in Section 8, to give a complete account of the existing type systems for the π-calculus. Applications of our type system to programming languages and verification systems are also future work. The generic type system would be useful both (1) as a theoretical framework to design a type system for analyzing specific properties of programs (e.g., deadlock and race conditions), and (2) as a basis for constructing a general-purpose program verification tool with which properties to be verified can be specified as a formula of the process logic introduced in Section 4.2. Since the generic type system itself is not suitable for complete type inference (because of the non-determinism of the choice of Γ_1 and Γ_2 in rule (T-In), etc.), we need to either restrict the type system or allow a programmer to explicitly declare type information for the latter goal. We also need to study a model checking algorithm, which, given a type and a formula of the process logic, checks whether the type satisfies the formula.

Acknowledgments

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

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[2] 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.

[3] S. Chaki, S. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 45–57, 2002.

[4] S. Conchon and F. Pottier. JOIN(X): Constraint-based type inference for the join-calculus. In *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 221–236. Springer-Verlag, 2001.

[5] 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.

[6] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[7] R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[8] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Proceedings of SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–126. Springer-Verlag, 1997.

[9] J. Feret. Occurrence counting analysis for the pi-calculus. In *Electronic Notes in Theoretical Computer Science*, volume 39, pages 55–77. Elsevier Science Publishers, 2001.

[10] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.

[11] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP 1999*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, 1999.

[12] 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.

[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[14] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[15] S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[16] 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.

[17] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*. To appear. A preliminary version appeared in Seventeenth Conference on the Mathematical Foundations of Programming Semantics (MFPS 2001), Elsevier ENTCS 45, 2001.

[18] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*, pages 145–159. IEEE Computer Society Press, 2001.

[19] M. Hennessy and J. Riely. Information flow vs. resource access in the information asynchronous pi-calculus. In *Proceedings of ICALP 2000*, volume 1853 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2000.

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

[21] K. Honda. A theory of types for π-calculus. Typescript, November 1998. Available at `http://www.dcs.qmul.ac.uk/~kohei/`.

[22] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

[23] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2002.

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

[25] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.

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

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

[28] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.

[29] N. Kobayashi. Time regions and effects for resource usage analysis. In *Proceedings of ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'03)*, pages 50–61, 2003.

[30] 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.

[31] N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, August 2000.

[32] 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.

[33] B. König. Generating type systems for process graphs. In *Proceedings of CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 1999.

[34] B. König. Analysing input/output-capabilities of mobile processes with a generic type system. In *Proceedings of ICALP2000*, volume 1853 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] 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)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, 1995.

[43] 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.

[44] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *Proceedings of CONCUR2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 474–488, 2000.

[45] C. Stirling. Modal and temporal logics for processes. In *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 149–237, 1996.

[46] 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.

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

[48] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[49] 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.

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

[51] A. Venet. Automatic determination of communication topologies in mobile systems. In *Proceedings of 5th International Symposium on Static Analysis (SAS '98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 152–167. Springer-Verlag, 1998.

[52] N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–387. Springer-Verlag, 1996.

[53] N. Yoshida, M. Berger, and K. Honda. Strong normalization in the π-calculus. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 2001.

Appendix

A Proofs of Theorems

A.1 Proof of Subject Reduction Theorem (Theorem 4.1.1)

Lemma A.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 A.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 $P = x!^{\mathbf{t}}[\tilde{z}].\, P_0$, then there exist Γ_1 and Γ_2 such that $\Gamma_1 \triangleright P_0$ and $\Gamma \,\leq\, x!^{\mathbf{t}}[(\tilde{y})\Gamma_2].\, (\Gamma_1 \,|\, [\tilde{z}/\tilde{y}]\Gamma_2)$.
4. If $P = x?^{\mathbf{t}}[\tilde{y}].\, P_0$, then there exist Γ_1 and Γ_2 such that $\Gamma_1 \,|\, \Gamma_2 \triangleright P_0$ and $\Gamma \,\leq\, x?^{\mathbf{t}}[(\tilde{y})\Gamma_2].\, \Gamma_1$ with $\mathbf{FV}(\Gamma_1) \cap \{\tilde{y}\} = \emptyset$.
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 $ok(\Gamma\downarrow_{\{\tilde{x}\}})$ and $\mathbf{FV}(\Gamma_0\uparrow_{\{\tilde{x}\}}) \cap \{\tilde{x}\} = \emptyset$.
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 A.1.3: If $\Gamma \xrightarrow{L} \Gamma'$, then $\Gamma\uparrow_{\{\tilde{x}\}} \xrightarrow{L'} \Gamma'\uparrow_{\{\tilde{x}\}}$ where L' is defined by:

$$L' = \begin{cases} \epsilon^{\mathbf{t}_1,\mathbf{t}_2} & \text{if } L = y^{\mathbf{t}_1,\mathbf{t}_2} \text{ and } y \in \{\tilde{x}\} \\ L & \text{otherwise} \end{cases}$$

Proof: By induction on the derivation of $\Gamma \xrightarrow{L} \Gamma'$ 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: $\quad \Gamma = \cdots + y!^{\mathbf{t}_1}[\tau_1].\, \Gamma_1 + \cdots \,|\, \cdots + y?^{\mathbf{t}_2}[\tau_2].\, \Gamma_2 + \cdots$
$\quad \Gamma' = \Gamma_1 \,|\, \Gamma_2 \qquad L = y^{\mathbf{t}_1,\mathbf{t}_2} \qquad \tau_1 \,\leq\, \tau_2$

We have two subcases according to whether $y \in \{\tilde{x}\}$ or not. We will show the subcase where $y = x_i$. Since $\Gamma\uparrow_{\{\tilde{x}\}} = \cdots + \mathbf{t}_1.\Gamma_1\uparrow_{\{\tilde{x}\}} + \cdots \,|\, \cdots + \mathbf{t}_2.\Gamma_2\uparrow_{\{\tilde{x}\}} + \cdots$, by rule TER-Com2,

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

finishing the subcase. The other subcase is easy. □

Lemma A.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 A.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 assumptions $WF(\Gamma{\downarrow_S})$ and $WF(\Gamma{\uparrow_S})$. □

Lemma A.1.5 [substitution]: If $\Gamma \rhd P$, then $[\tilde{y}/\tilde{x}]\Gamma \rhd [\tilde{y}/\tilde{x}]P$.

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

Lemma A.1.6: If $\Gamma \rhd P$ and $P \preceq Q$, then $\Gamma \rhd Q$.

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

By Lemma A.1.2, there exist Γ_1, Γ_1', and Γ_2 such that $\Gamma_i \rhd P_i$ 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, $\Gamma_1 \mid \Gamma_2 \rhd P_1 \mid P_2$. Without loss of generality, we can assume $\tilde{x}$ are not free in Γ_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,

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

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, $\Gamma \rhd Q$.

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

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

Proof of Theorem 4.1.1: By induction on the derivation of $P \xrightarrow{l} Q$ with a case analysis on the last rule used.

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

By Lemma A.1.2 and the subtyping rules, there exist Γ_1, Γ_2, Γ_3, and Γ_4 such that

$$\begin{array}{l} \Gamma \leq \cdots + x!^{\mathbf{t}}[(\tilde{w})\Gamma_2].\,(\Gamma_1 \mid [\tilde{z}/\tilde{w}]\Gamma_2) + \cdots \mid \cdots + x?^{\mathbf{t}'}[(\tilde{y})\Gamma_4].\,\Gamma_3 + \cdots \\ \Gamma_1 \rhd P_0 \qquad \Gamma_3 \mid \Gamma_4 \rhd Q_0 \qquad \mathbf{FV}(\Gamma_3) \cap \{\tilde{y}\} = \emptyset. \end{array}$$

Then, since $WF(\Gamma)$, it must be the case that $(\tilde{w})\Gamma_2 \leq (\tilde{y})\Gamma_4$, that is, $\Gamma_2 \leq [\tilde{w}/\tilde{y}]\Gamma_4$. We can show

$$\Gamma \xrightarrow{x^{\mathbf{t},\mathbf{t}'}} \Gamma_1 \mid [\tilde{y}/\tilde{w}](\Gamma_3 \mid \Gamma_4)$$

by the following calculation:

$$\begin{array}{rcl}\Gamma & \xrightarrow{x^{\mathbf{t},\mathbf{t}'}} & \Gamma_1 \,|\, [\tilde{z}/\tilde{w}]\Gamma_2 \,|\, \Gamma_3 \\ & \leq & \Gamma_1 \,|\, [\tilde{z}/\tilde{w}][\tilde{w}/\tilde{y}]\Gamma_4 \,|\, \Gamma_4 \\ & = & \Gamma_1 \,|\, [\tilde{z}/\tilde{y}]\Gamma_4 \,|\, \Gamma_3 \\ & = & \Gamma_1 \,|\, [\tilde{z}/\tilde{y}](\Gamma_4 \,|\, \Gamma_3) \quad (\mathbf{FV}(\Gamma_3) \cap \{\tilde{y}\} = \emptyset)\end{array}$$

By Lemma A.1.5, $[\tilde{z}/\tilde{y}](\Gamma_3 \,|\, \Gamma_4) \triangleright [\tilde{z}/\tilde{y}]Q_0$. Finally, by the rule T-PAR, $\Gamma_1 \,|\, [\tilde{z}/\tilde{y}](\Gamma_3 \,|\, \Gamma_4) \triangleright P_0 \,|\, [\tilde{z}/\tilde{y}]Q_0$, finishing the case.

Case R-PAR**:** $\quad P = P_0 \,|\, R \qquad P_0 \xrightarrow{l} Q_0 \qquad Q = Q_0 \,|\, R$

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

$$\Gamma \leq \Gamma_1 \,|\, \Gamma_2 \quad \Gamma_1 \triangleright P_0 \quad \Gamma_2 \triangleright R$$

To use the induction hypothesis, we show $WF(\Gamma_1)$ by contradiction. Suppose $WF(\Gamma_1)$ does not hold. Then, $WF(\Gamma_1 \,|\, \Gamma_2)$ 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, $\Gamma \xrightarrow{l} \Gamma_1' \,|\, \Gamma_2$. By the rule T-PAR, $\Gamma_1' \,|\, \Gamma_2 \triangleright Q_0 \,|\, R$, finishing the case.

Case R-NEW**:** $\quad P = (\nu\tilde{x})\, P_0 \qquad P_0 \xrightarrow{l'} Q_0 \qquad Q = (\nu\tilde{x})\, Q_0 \qquad l = l'\!\uparrow_{\{\tilde{x}\}}$

We show the subcase where $l' = x_i^{\mathbf{t},\mathbf{t}'}$ for some i, thus $l = \epsilon^{\mathbf{t},\mathbf{t}'}$; The other cases ($l' = \epsilon^{\mathbf{t},\mathbf{t}'}$ or $l' = y^{\mathbf{t},\mathbf{t}'}$ with $y \notin \{\tilde{x}\}$) are similar. By Lemma A.1.2, there exists Γ_0 such that

$$\Gamma_0 \triangleright P_0 \qquad \Gamma \leq \Gamma_0\!\uparrow_{\{\tilde{x}\}} \qquad ok(\Gamma_0\!\downarrow_{\{\tilde{x}\}}) \qquad \mathbf{FV}(\Gamma_0\!\uparrow_{\{\tilde{x}\}}) \cap \{\tilde{x}\} = \emptyset.$$

Since $WF(\Gamma_0\!\downarrow_{\{\tilde{x}\}})$ and $WF(\Gamma_0\!\uparrow_{\{\tilde{x}\}})$ from the assumptions, we have $WF(\Gamma_0)$ by Lemma A.1.4. By the induction hypothesis, there exists Γ_0' such that $\Gamma_0 \xrightarrow{x_i^{\mathbf{t},\mathbf{t}'}} \Gamma_0'$ and $\Gamma_0' \triangleright Q_0$. By Lemma A.1.3, $\Gamma_0\!\uparrow_{\{\tilde{x}\}} \xrightarrow{\epsilon^{\mathbf{t},\mathbf{t}'}} \Gamma_0'\!\uparrow_{\{\tilde{x}\}}$ and $\Gamma_0\!\downarrow_{\{\tilde{x}\}} \xrightarrow{x_i^{\mathbf{t},\mathbf{t}'}} \Gamma_0'\!\downarrow_{\{\tilde{x}\}}$. Then, $ok(\Gamma_0'\!\downarrow_{\{\tilde{x}\}})$. It is easy to show $\Gamma \xrightarrow{L} \Gamma'$ implies $\mathbf{FV}(\Gamma) \supseteq \mathbf{FV}(\Gamma')$, thus $\mathbf{FV}(\Gamma_0'\!\uparrow_{\{\tilde{x}\}}) \cap \{\tilde{x}\} = \emptyset$. Finally, by the rule T-NEW, $\Gamma_0'\!\uparrow_{\{\tilde{x}\}} \triangleright (\nu\tilde{x})\, Q_0$, finishing the case.

Case R-SP**:** $\quad P \preceq P' \qquad P' \xrightarrow{l} Q' \qquad Q' \preceq Q'$

Immediate from Lemma A.1.6 and the induction hypothesis. □

A.2 Proof of Theorem 5.2

We first show properties of $\leq_1$ and $\leq_2$.

Lemma A.2.7: Let $\leq$ be $\leq_i$ ($i \in \{1,2\}$). Then, the following conditions hold:

- $\Gamma \models x!^{\mathbf{t}}\zeta$, then $\Gamma \cong \cdots + x!^{\mathbf{t}}[\tau].\, \Delta_1 + \cdots \,|\, \Delta_2$ for some τ, Δ_1, and Δ_2.
- $\Gamma \models x?^{\mathbf{t}}\zeta$, then $\Gamma \cong \cdots + x?^{\mathbf{t}}[\tau].\, \Delta_1 + \cdots \,|\, \Delta_2$ for some τ, Δ_1, and Δ_2.
- $\Gamma \models \langle\mathbf{t}\rangle\mathtt{true}$, then $\Gamma \cong \cdots + \mathbf{t}.\Delta_1 + \cdots \,|\, \Delta_2$ for some Δ_1 and Δ_2.
- $\Gamma \models x!^{\mathbf{t}}\zeta \,|\, x?^{\mathbf{t}}\zeta$, then $\Gamma \cong \cdots + x!^{\mathbf{t}}[\tau_1].\, \Delta_1 + \cdots \,|\, \cdots + x?^{\mathbf{t}}[\tau_2].\, \Delta_2 + \cdots \,|\, \Delta_3$. for some τ_1, τ_2, Δ_1, Δ_2 and Δ_3.

Proof: By the definition of $\models$, $\Gamma \models x!^{\mathbf{t}}\zeta$ if and only if $\Gamma \leq \cdots + x?^{\mathbf{t}}[\tau].\,\Delta_1 + \cdots \,|\, \Delta_2$ for some Δ_1 and Δ_2. Therefore, the first property follows, for both $\leq_1$ and $\leq_2$, by straightforward induction on derivation of the subtyping relation. Proofs of the other properties are similar. $\Box$

We check the necessary condition for Theorem 4.1.2 below.

Lemma A.2.8: Let $\leq$ be $\leq_i$ ($i \in \{1,2\}$), and let ok be ok_4. If $\Gamma \triangleright P$ and $ok(\Gamma)$, then $p_4(P)$ holds.

Proof: The proof proceeds by well-founded induction on $\mathbf{t}$. Suppose $\Gamma \triangleright P$, $ok(\Gamma)$, and $P \preceq (\widetilde{\nu w_{1..k}})\, Q$ with $Q \models x!^{\mathbf{t}}\zeta \vee x?^{\mathbf{t}}\zeta$ for some x, ζ. It suffices to show that $Q \models \exists x.\exists \mathbf{t}_1, \mathbf{t}_2.(\langle x^{\mathbf{t}_1,\mathbf{t}_2}\rangle\mathtt{true} \vee \langle \epsilon^{\mathbf{t}_1,\mathbf{t}_2}\rangle\mathtt{true})$, that is, $Q \longrightarrow R$ for some R. We show only the case for $Q \models x!^{\mathbf{t}}\zeta$: The case for $Q \models x?^{\mathbf{t}}\zeta$ is similar. By Lemma A.1.6 and Theorem 4.1.3, we have

$$\begin{array}{l} \Delta \triangleright_N Q \\ \Delta \models x!^{\mathbf{t}}\zeta \\ \Gamma \leq \Delta\uparrow_{\{\tilde{w_1},\ldots,\tilde{w_k}\}} \end{array}$$

Moreover, since $ok(\Gamma)$, there exists a set S such that $x \in S$ and $ok(\Delta\downarrow_S)$. (Let S be $\{\tilde{w_i}\}$ if $y \in \{\tilde{w_i}\}$, and $\mathbf{FV}(\Delta)\backslash\{\tilde{w_1},\ldots,\tilde{w_k}\}$ otherwise.) So, by the definition of ok_4, there exist $y, \mathbf{t}_2, \mathbf{t}_3$ such that $\Delta\downarrow_S \models \langle y^{\mathbf{t}_2,\mathbf{t}_3}\rangle\mathtt{true}$ or $\Delta\downarrow_S \models \langle \mathbf{t}_2\rangle\mathtt{true} \wedge \mathbf{t}_2 \prec \mathbf{t}$. In the former case, $\Delta \models y!^{\mathbf{t}_2}\zeta \,|\, y?^{\mathbf{t}_3}\zeta$. By Theorem 4.1.5 and Lemma A.2.7, we have $Q \xrightarrow{y^{\mathbf{t}_2,\mathbf{t}_3}} R$ for some R as required. In the latter case, by Theorem 4.1.5 and Lemma A.2.7, we have $Q' \models y!^{\mathbf{t}_2}\zeta \vee y?^{\mathbf{t}_2}\zeta$ and $Q \preceq (\widetilde{\nu u_{1..l}})\, Q'$ for some $y, \tilde{u_1},\ldots,\tilde{u_l}$. By the induction hypothesis, we have $Q \longrightarrow R$ for some R, as required. $\Box$

Proof of Theorem 5.2: This follows immediately from Lemma A.2.8 and Theorem 4.1.2. $\Box$