An Implicitly-Typed Deadlock-Free Process Calculus*

Naoki Kobayashi Shin Saito Eijiro Sumii
Department of Information Science, University of Tokyo
7-3-1 Hongo,Bunkyo-ku, Tokyo 113-0033, Japan
email:koba@is.s.u-tokyo.ac.jp

June 14, 2000

Abstract

We extend Kobayashi and Sumii's type system for the deadlock-free π-calculus and develop a type reconstruction algorithm. Kobayashi and Sumii's type system helps high-level reasoning about concurrent programs by guaranteeing that communication on certain channels will eventually succeed. It can ensure, for example, that a process implementing a function really behaves like a function. However, because it lacked a type reconstruction algorithm and required rather complicated type annotations, applying it to real concurrent languages was impractical. We have therefore developed a type reconstruction algorithm for an extension of the type system. The key novelties that made it possible are generalization of *usages* (which specifies how each communication channel is used) and a *subusage* relation.

1 Introduction

1.1 General Background

With increasing opportunities of distributed programming, static guarantee of program safety is becoming extremely important, because (i) distributed programs are inherently concurrent and exhibit more complex behavior than sequential programs, (ii) it is hard to debug the whole distributed systems, and (iii) distributed programs usually involve many entities, some of which may be malicious. Lack of static guarantee results in unsafe or slow (due to expensive run-time check) program execution. Among various issues of program safety such as security, this paper focuses on problems caused by concurrency, in particular, deadlock (in a broad sense).

Traditional type systems are insufficient to guarantee the correctness of concurrent/distributed programs. Consider the following program of CML [21]:

```
fun f n = let val ch=channel() in recv(ch)+n+1 end;
```

The function `f` creates a new channel `ch` (by `channel()`), waits for a value v from the channel (by `recv(ch)`), and returns $v + n + 1$. Since there is no sender on the channel `ch`, the application `f(1)` is blocked forever. Thus, `f` actually does not behave like a function, but the type system of CML assigns to `f` a function type $int \to int$.

*A revised version of the technical report TR00-01, Department of Information Science, Univeristy of Tokyo. A summary will appear in Proceedings of CONCUR2000.

1.2 Our Previous Type Systems for Deadlock-freedom and Their Problem

To overcome problems like above, a number of type systems [11, 16, 26] have been studied through π-calculus [14] (just as type systems for functional languages have been studied through λ-calculus). For example, Pierce and Sangiorgi [16] introduced a type system that can guarantee that certain channels are used only for input or output, and Kobayashi, Pierce, and Turner [11] introduced a linear type system that can guarantee that certain channels (called linear channels) are used just once for communication. Among them,advanced type systems for the π-calculus, Our type systems for deadlock-freedom [10, 23] are among the most powerful type systems: They can guarantee partial deadlock-freedom in the sense that communication on certain channels will eventually succeed. In addition to the usual meaning of deadlock-freedom where communications are blocked due to some circular dependencies, it also detects the situation like above, where there exists no communication partner from the beginning. Through the guarantee of deadlock-freedom, they can uniformly ensure that functional processes really behave like functions, that concurrent objects will eventually accept a request for method execution and send a reply, and that binary semaphores are really used like binary semaphores (a process that has acquired a semaphore will eventually release it unless it diverges).

In spite of the attractive features of the deadlock-free type systems, however, their applications to real concurrent programming languages have been limited. The main reason is that there was no reasonable type reconstruction algorithm and therefore programmers had to explicitly annotate programs with rather complex types.

1.3 Contributions of This Paper

To solve the above-mentioned problem, this paper develops an *implicitly-typed* version of the deadlock-free process calculus and its type reconstruction algorithm. Programmers no longer need to write complex type expressions; Instead, they just need to declare which communication they want to succeed. (Programmers may still want to partially annotate programs with types for documentation, etc.: Our algorithm can be easily modified to allow such partial type annotation.) For example, a process that sends a request to a function server or a concurrent object can be written as $(\nu r)\,(s![arg, r] \mid r?^{\mathbf{c}}[x]. \ldots)$. Here, (νr) creates a fresh channel r. $s![arg, r]$ sends a pair $[arg, r]$ to the server through channel s, and in parallel to this, $r?^{\mathbf{c}}[x]. \ldots$ waits on channel r to receive a reply from the server. The **c** attached to ? indicates that this input from r should eventually succeed, i.e., a reply should eventually arrive on r. If the whole system of processes (including the server process) is judged to be well typed in our type system, then it is indeed guaranteed that the input will eventually succeed, unless the whole system diverges.

Our new technical contributions are summarized as follows. (Those who are unfamiliar with our previous type systems can skip the rest of this paragraph.)

- Generalization of the previous type systems for deadlock-freedom — It is not possible to construct a reasonable type reconstruction algorithm for the previous type systems. So, we generalized them by introducing a subusage relation and new usage constructors such as recursive usages and the greatest lower bound of usages (which roughly correspond to the subtype relation, recursive types, and intersection types in the usual type system). A usage [23] is a part of a channel type and describes for which operations (input or output) and in which order channels can and/or must be used. It can be considered an extension of input/output modes [16] and multiplicities [11].

- Constraint-based type reconstruction algorithm — We have developed a type reconstruction algorithm, which inputs an implicitly-typed process and checks whether it is well typed or not. The algorithm

is a non-trivial extension of Igarashi and Kobayashi's type reconstruction algorithm [8] for the linear π-calculus [11], where a principal typing is expressed as a pair of a type environment and a set of constraints on type/usage variables.

1.4 Limitations of This Paper

The type system and type reconstruction algorithm described in this paper have the following limitations.

- Incompleteness of the type reconstruction algorithm — The algorithm is sound but incomplete: Although it never accepts ill-typed processes, it rejects some well-typed processes. This is just because we want to reject some well-typed but bad processes that may livelock (i.e., diverge with keeping some process waiting for communication forever). So, our algorithm is actually preferable to a complete algorithm (if there is any).

- Naive treatment of time tags — The treatment of time tags and tag relations, which are key features of the deadlock-free type systems [10, 23], is very naive in this paper. As a result, the expressive power is very limited. This is just for clarifying the essence of new ideas of this paper. It is easy to replace the naive treatment of time tags in this paper with the sophisticated one in the previous papers [10, 23] and extend the type reconstruction algorithm accordingly. The resulting deadlock-free process calculus is more expressive than the previous calculi [10, 23], which have already been shown to be expressive enough to encode the simply-typed λ-calculus with various evaluation strategies, semaphores, and typical concurrent objects.

1.5 The Rest of This Paper

The rest of this paper is structured as follows. In Section 2, we briefly review the previous type systems for deadlock-freedom [10, 23], discuss why type reconstruction was difficult, and explain key ideas of the new type system and the type reconstruction algorithm. Section 3 introduces the syntax of types and processes and defines the operational semantics of processes. Section 4, 5 and 6 are the main part of this paper: after defining a type system in Section 4, we describe a constraint-based type reconstruction algorithm, which inputs a process expression and outputs a pair of a type judgment containing type variables and a set of constraints on variables, in Section 5. Section 6 describes how to solve those constraints and decide typability of the input process. We discuss extensions of the type system in Section 7 and discuss related work in Section 8.

2 Main Ideas

In this section, we first review basic ideas of Kobayashi and Sumii's type systems for deadlock-freedom [10, 23], and explain why type reconstruction was difficult. After that, we explain key new ideas that enabled type reconstruction.

2.1 Why Type Systems for Deadlock-Freedom

In order to explain the necessity of type systems for deadlock-freedom and ideas of Kobayashi and Sumii's type system, we use the following subset of the polyadic pi-calculus [13]:

$$
\begin{array}{rcll}
P & ::= & P_1 \mid P_2 & \text{(executes } P_1 \text{ and } P_2 \text{ concurrently)} \\
 & & x![v_1, \ldots, v_n] & \text{(sends a tuple } [v_1, \ldots, v_n] \text{ along the channel } x) \\
 & & x?[z_1, \ldots, z_n].\, P & \text{(receives a tuple } [v_1, \ldots, v_n] \text{ along } x \text{ and behaves like } [z_1 \mapsto v_1, \ldots, z_n \mapsto v_n]P) \\
 & & x?^*[z_1, \ldots, z_n].\, P & \text{(repeatedly receives a tuple } [v_1, \ldots, v_n] \text{ along } x \\
 & & & \text{ and spawns the process } [z_1 \mapsto v_1, \ldots, z_n \mapsto v_n]P) \\
 & & (\nu x)\, P & \text{(creates a channel } x \text{ and executes } P) \\
 & & \textbf{if } v \textbf{ then } P \textbf{ else } Q & \text{(executes } P \text{ if } v = \mathit{true} \text{ and } Q \text{ if } v = \mathit{false})
\end{array}
$$

Here, $x?^*[z_1, \ldots, z_n].\, P$ denotes a replication of $x?[z_1, \ldots, z_n].\, P$: it behaves like a (recursive) process definition $x![\tilde{z}] = P$ since, if it is running, the process $[z_1 \mapsto v_1, \ldots, z_n \mapsto v_n]P$ is spawned whenever $x![v_1, \ldots, v_n]$ is executed.

A function $\lambda x.M$ can be implemented as a process of the form $f?^*[x, r].\, \cdots$, which receives an argument x and a reply channel r, evaluates M and returns the result to r. For example, a function that computes the factorial can be implemented as the following process Fact:

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

This process receives a pair of an integer n and a channel r along the channel fact and tests whether $n = 0$. If $n = 0$, then it returns the result 1 (the factorial of 0) to r; otherwise, it computes the factorial of $n - 1$ by sending a request to the channel fact, receives the result m along channel r', and sends the whole result $m \times n$ to channel r. Readers who are familiar with the continuation passing style implementation of functions [1] would notice that reply channels (r and r' above) correspond to continuations.

As in the untyped λ-calculus, one can easily violate the intended use of the above process in the untyped π-calculus. For example, by sending a pair $[0, 2]$ of integers to fact, execution of the above process gets stuck with $2![1]$. In order to overcome such a problem, earlier type systems for process calculi or concurrent languages [5, 21, 25] incorporated into channel types information on values communicated along each channel. Let us write $\updownarrow[\tau_1, \ldots, \tau_n]$ for the type of a channel for communicating a tuple of values of types $\tau_1, \ldots, \tau_n$. Then, the above process Fact is typed as follows:

$$
\mathit{fact} : \updownarrow[\mathit{int}, \updownarrow[\mathit{int}]] \vdash \mathit{Fact}
$$

This type judgment means that Fact is a good process under the assumption that fact is a channel used for communicating pairs of an integer and a channel for exchanging integers. Invalid senders like $\mathit{fact}![0, 2]$ and $\mathit{fact}!["a", r]$ are rejected by this typing.

However, the type system is still not effective enough to enforce that Fact is used as a function. Since the assumption $\mathit{fact} : \updownarrow[\mathit{int}, \updownarrow[\mathit{int}]]$ only says that fact is a channel for communicating a pair of type $[\mathit{int}, \updownarrow[\mathit{int}]]$, it does not forbid an outside process from stealing a request for evaluating the function (i.e., receiving a pair from the channel fact). It is not powerful enough to help reasoning about the behavior of Fact, either: Without looking at the code of Fact, we cannot tell from the type judgment that Fact uses the reply channel r correctly (i.e., it does not try to receive a value from r or send more than one results to r).

In order to overcome these problems, Pierce and Sangiorgi [16] classified channel types according to whether channels can be used for input, output, or both, and Kobayashi, Pierce and Turner [11] further classified channel types according to whether channels can be used just once or an arbitrary number of times. In Pierce and Sangiorgi's type system, the above judgment is refined to:

$$
\mathit{fact} : \updownarrow[\mathit{int}, ![\mathit{int}]] \vdash \mathit{Fact}
$$

Here, $![\tau]$ is the type of channels that can be used *only for sending* values of type τ. Now the judgment asserts that $Fact$ uses a reply channel only for sending integers, not for receiving integers. For the outside processes, $fact$ is given type $![int, ![int]]$, and thus, stealing a request on $fact$ is forbidden. In Kobayashi, Pierce, and Turner's linear type system [11], the type judgment is further refined to:

$$fact : \updownarrow^{\omega}[int, !^{1}[int]] \vdash Fact$$

Here, $\updownarrow^{\omega}[\tau]$ is the type of channels that can be used many times for communication, and $!^{1}[\tau]$ is the type of channels that can be used only once for output, never for input. Thus, the judgment certifies that $Fact$ uses a reply channel only once for sending an integer. For external processes, $fact$ is viewed as a channel of type $!^{\omega}[int, !^{1}[int]]$.

Even with the linear type system, we miss some important properties of functions. In an ordinary typed λ-calculus, if we have $\Gamma \vdash M : int$, then M is guaranteed to evaluate to an integer, unless M diverges; in other words, the evaluation never gets stuck. However, there is no such guarantee in the linear type system for the π-calculus. For example, $(\nu fact)\,(Fact \mid fact![2, r])$ is typed as

$$r : !^{1}[int] \vdash (\nu fact)\,(Fact \mid fact![2, r]),$$

but from this judgment, we cannot tell that an integer will be output on r, because the same judgment holds also for $Fact' = fact?^{*}[n, r].\,(\nu dummy)\,(dummy?[\,].\,r![n])$:

$$r : !^{1}[int] \vdash (\nu fact)\,(Fact' \mid fact![2, r]).$$

Notice that $Fact'$ waits on $dummy$ forever and never returns an integer. This kind of problem is not peculiar to encoding of functions. If a semaphore is implemented by using a channel, the type system cannot guarantee that a process can eventually acquire the semaphore. A method of a concurrent object implemented by using a π-calculus process is not guaranteed to return the result, either.

These situations can all be regarded as *deadlock* in a certain sense: In all the above cases, the problem is that the execution of a process that is required to perform a certain communication gets stuck (i.e., is blocked forever) and that, as a result, a process waiting for the communication also gets stuck. Based on this intuition, Kobayashi and Sumii [10, 23] formally defined the notion of deadlock in the π-calculus and developed type systems to guarantee deadlock-freedom. In their type systems, for example, if $x : !^{1}[int] \vdash P$ holds, then it is guaranteed that P will eventually send an integer on x or diverge. This property is analogous to the property guaranteed by $\vdash M : int$ in an ordinary typed λ-calculus (with recursion). Indeed, the property of the simply-typed λ-calculus is recovered in our type system: If $\vdash M : int$ holds in the simply-typed λ-calculus, $x : !^{1}[int] \vdash [\![M]\!]_x$ holds in Kobayashi's type system [10], where $[\![M]\!]_x$ is a process simulating evaluation of M and returns the result to x. So, we can reason that the evaluation of a well-typed λ-term does not get stuck at the level of the π-calculus. We review ideas of the type systems in the next subsection.

2.2 Ideas of Kobayashi and Sumii's Deadlock-Free Type Systems

The key ideas of Kobayashi and Sumii's deadlock-free type systems are as follows:

Generalization of input/output modes and linearity The linear π-calculus [11] is only concerned with whether channels are used just once or more; it does not care about whether channels are used twice or more, or about the order in which outputs and inputs are performed. This is fine for reasoning about processes implementing functions, but is problematic for reasoning about other processes like those

manipulating channels as binary semaphores. Since a channel implementing a binary semaphore is used many times for input and output, it is given a type $\updownarrow^{\omega}[\,]$ in the linear π-calculus. However, it is actually used in a certain restricted manner: except for the initial output, an output can be performed only after an input. In order to express this kind of restriction in terms of types, Sumii and Kobayashi [23] refined a channel type to $[\tau]/U$, where an expression U, called a *usage*, is given by the following syntax:

$$U ::= 0 \mid I.U \mid O.U \mid (U_1||U_2) \mid *U$$

0 means that the channel cannot be used at all. $I.U$ ($O.U$, resp.) means that the channel can first be used for input (output, resp.) and then used according to U. $U_1||U_2$ means that the channel can be used according to U_1 and U_2 concurrently, and $*U$ (which was written $!U$ in [23]) means that the channel can be used according to U an arbitrary number of times (possibly simultaneously). By using usages, we can express how often and in which order channels should be used for input and output. For example, a channel used as a binary semaphore is given the type $[\,]/(O.0||*I.O.0)$. A linear channel for communicating integers is given the type $[int]/(O.0||I.0)$.

Capabilities and obligations When we implement various mechanisms in terms of π-calculus processes, not all communications are required to eventually succeed. For example, for a channel (like $fact$ above) corresponding to the location of a function, an output on the channel should eventually succeed, but it is fine that an input process (like $Fact$ above) is waiting for a request forever. For another example, consider a channel used as a binary semaphore. An input on the channel should eventually succeed since an input corresponds to the acquisition (a P-operation, in other words) of the semaphore. However, it is fine that an output process is waiting forever, since it means that no process tries to acquire the semaphore. To summarize, some communications are required to succeed (i.e, the process trying to perform those communications can eventually find their communication partners), while others are not, depending on programmers' intention on the use of channels. Similarly, some communications *must* be provided (even if they do not succeed) while others need not: A process implementing a function must output a result to the reply channel, and a process that has acquired the semaphore must release it, while a process need not acquire the semaphore. In order to make these requirements explicit, Sumii and Kobayashi [23] introduced *capabilities* (denoted by $\mathbf{c}$) and *obligations* (denoted by $\mathbf{o}$) as attributes of I and O of the usages. If I (O, resp.) is annotated with $\mathbf{c}$, then an input (output, resp.) on the channel will eventually succeed; in other words, some other process will perform an output (input, resp.). If I (O, resp.) is annotated with $\mathbf{o}$, then an input (output, resp.) on the channel must be performed. The type of a binary semaphore channel is now refined to $[\,]/(O_{\mathbf{o}}.0||*I_{\mathbf{c}}.O_{\mathbf{o}}.0)$. $O_{\mathbf{o}}.0$ indicates that a value must be first put into the channel, and $I_{\mathbf{c}}.O_{\mathbf{o}}.0$ indicates that a process can extract a value from the channel (i.e., can acquire the semaphore), and after having extracted, it must put a value back into the channel. (Note here that the output obligation in $I_{\mathbf{c}}.O_{\mathbf{o}}.0$ arises only after an input capability $I_{\mathbf{c}}$ is consumed.) $*$ indicates that a channel can be used according to $I_{\mathbf{c}}.O_{\mathbf{o}}.0$ an arbitrary number of times by an arbitrary number of processes, possibly in parallel. A linear channel for communicating an integer is now given the type $[int]/(O_{\mathbf{co}}.0||I_{\mathbf{co}}.0)$. The refined usage $O_{\mathbf{co}}.0||I_{\mathbf{co}}.0$ means that the channel must be used once each for input and for output, and that the input and output will eventually succeed.

What we mean by deadlock is now clear: the deadlock is a state where processes cannot be reduced any more and either (1) an input or output operation with the obligation attribute has not been performed, or (2) an input or output operation with the capability attribute is performed, but has not succeeded (the input or output process remains unreduced). Kobayashi and Sumii's type systems guarantee that a well-typed process is free from such deadlock. For example, if $x : [int]/O_{\mathbf{o}}.0 \vdash P$ holds and P is reduced to

some irreducible process Q, then an integer is being sent on x in Q (in technical terms, Q is structurally congruent to a process of the form $x![n] \mid Q'$).

Usage reliability — channel-wise consistency of capabilities and obligations In order to guarantee deadlock-freedom, we need to require some consistency on the capabilities and obligations associated to each channel. For example, creating a channel of usage $I_{\mathbf{c}}.0||I_{\mathbf{c}}.0$ does not make sense, because input operations cannot succeed as no output operations are allowed by the usage. So, we must require that if an input is annotated with the capability attribute $\mathbf{c}$, then there must be a corresponding output annotated with the obligation attribute $\mathbf{o}$. For example, $I_{\mathbf{c}}.0||O_{\mathbf{o}}.0$ and $I_{\mathbf{co}}.0||O_{\mathbf{co}}.0$ are consistent, but neither $I_{\mathbf{c}}.0||O_{\mathbf{c}}.0$ nor $I_{\mathbf{c}}.0||I_{\mathbf{o}}.0$ is. This consistency must be preserved after reductions. Let $I_{\mathbf{c}}.I_{\mathbf{c}}.0||O_{\mathbf{o}}.0$ be the intended usage of a channel. After the channel is used once for communication, it should then be used according to the usage $I_{\mathbf{c}}.0$, obtained by cancelling one I and one O; this usage is inconsistent, since there is no available output operation. Sumii and Kobayashi [23] called these requirements the *reliability* of a usage in [23]. The usage $O_{\mathbf{o}}.0||*I_{\mathbf{c}}.O_{\mathbf{o}}.0$ of a binary semaphore channel is reliable, since an input with capability is matched by an output with obligation, and that condition is kept after reductions (i.e., after cancellation of one I and one O). In Sumii and Kobayashi's type system, it is checked whether U is reliable for each usage U appearing in a channel creation $(\nu x : [\tilde{\tau}]/U)\, P$.

Time tags and tag relations — inter-channel consistency of capabilities and obligations The reliability of each channel usage is not sufficient to guarantee deadlock-freedom. For example, consider the following process:

$$(\nu x : []/(O_{\mathbf{o}}.0||I_{\mathbf{c}}.0))\, (\nu y : []/(O_{\mathbf{o}}.0||I_{\mathbf{c}}.0))\, (x?[].\, y![].\ |\, y?[].\, x![])$$

The usage $O_{\mathbf{o}}.0||I_{\mathbf{c}}.0$ of x and y are reliable, because the input capability $I_{\mathbf{c}}$ is matched by the output capability $O_{\mathbf{o}}$, and the actual usages of x and y look like conforming with the specified usage (since they are used once for input and once for output). However, the process actually deadlocks — why? The subprocess $x?[].\, y![]$ fulfills the output obligation on y only if the input capability on x is guaranteed, while the other subprocess $y?[].\, x![]$ fulfills the output obligation on x only if the input capability on y is guaranteed. However, the input capability on x (y, resp.) is guaranteed only if the output obligation on x (y, resp.) is fulfilled, hence a cyclic dependency among the capabilities and obligations on x and y.

In order to prevent this kind of cyclic dependency, Kobayashi [10] introduced an ordering on usages, so that a capability to communicate on a channel x may be used before an obligation to communicate on a channel y only if the usage of x is *less than* the usage of y with respect to the ordering. In order to specify the ordering, Kobayashi annotated channel types with labels, like $[\tilde{\tau}]^t/U$. Labels are called *time tags* and an ordering was called *tag relations*. The type judgment form is extended by using time tags and tag relations. For example, a process $x?[].\, y![]$ is typed as follows:

$$x : []^{t_x}/I_{\mathbf{c}}.0, y : []^{t_y}/O_{\mathbf{o}}.0; \{(t_x, t_y)\} \vdash x?[].\, y![].$$

The tag ordering $\{(t_x, t_y)\}$ indicates that the process may try to use the capability to receive a null tuple on x before fulfilling the obligation to send a null tuple on y. So, $x?[].\, \mathbf{0} \mid y![]$ is also well typed under the same assumption:

$$x : []^{t_x}/I_{\mathbf{c}}.0, y : []^{t_y}/O_{\mathbf{o}}.0; \{(t_x, t_y)\} \vdash x?[].\, \mathbf{0} \mid y![]$$

However, the judgment:

$$x : []^{t_x}/O_{\mathbf{o}}.0, y : []^{t_y}/I_{\mathbf{c}}.0; \{(t_x, t_y)\} \vdash y?[].\, x![]$$

is invalid, since the process tries to use the capability on y before fulfilling the obligation on x. In order to allow this process, the tag relation must include the pair (t_y, t_x). So, the following judgment for $y?[].\, x![]$ is valid:

$$x : []^{t_x}/O_{\mathbf{o}}.0, y : []^{t_y}/I_{\mathbf{c}}.0; \{(t_y, t_x)\} \vdash y?[].\, x![]$$

In order for a parallel composition $x?[].\, y![].\, |\, y?[].\, x![]$ to be typed, the tag relation must contain both (t_y, t_x) and (t_x, t_y):

$$x : []^{t_x}/(O_{\mathbf{o}}.0||I_{\mathbf{c}}.0), y : []^{t_y}/(O_{\mathbf{o}}.0||I_{\mathbf{c}}.0); \{(t_x, t_y), (t_y, t_x)\} \vdash x?[].\, y![]\, |\, y?[].\, x![]$$

Now, the tag relation means that the process can delay fulfillment of the obligation on x until the capability on y is consumed, and it can also delay the fulfillment of the obligation on y until the capability on x is consumed, implying that a deadlock may occur. In general, if a type judgment $x_1 : \tau_1, \ldots, x_n : \tau_n; \mathcal{T} \vdash P$ holds and the tag relation $\mathcal{T}$ is a strict partial order, then no deadlock can occur (except for that on free channels $x_1, \ldots, x_n$: external processes will provide necessary communication on those channels).

2.3 Ideas of our type inference algorithm

In Kobayashi and Sumii's type systems, time tags and the tag relation can be automatically inferred, but other type information such as usages must be explicitly specified by programmers. For example, in order to create a channel x representing the location of a function of type $(int \to int) \to (int \to int)$, a programmer must write $(\nu x : [[int, [int]/O_{\mathbf{o}}.0]/{*O_{\mathbf{c}}}.0, [[int, [int]/O_{\mathbf{o}}.0]/{*O_{\mathbf{c}}}.0]/O_{\mathbf{o}}.0]/({*O_{\mathbf{c}}}.0||{*I_{\mathbf{o}}}.0)) \cdots$. Although some type annotation is certainly necessary for specifying the programmer's intention, it is often cumbersome to write all type annotations. So, it would be useful if programmers specify type information only when they want to do so, and the unspecified types can be automatically recovered by a type inference algorithm.

The main difficulty of the type reconstruction was that Kobayashi and Sumii's type systems did not have the principal typing property. For example, for a process $x?[].\, y![]$, there are the following valid type judgments, but neither seems more general than the other:

$$x : []^{t_x}/I_{\mathbf{o}}.0, y : []^{t_y}/O_{\mathbf{c}}.0; \emptyset \vdash x?[].\, y![]$$
$$x : []^{t_x}/I_{\mathbf{c}}.0, y : []^{t_y}/O_{\mathbf{o}}.0; \{(t_x, t_y)\} \vdash x?[].\, y![]$$

For another example, consider a process $x?[].\, x![]$: it has the following valid type judgments, but again, neither seems more general than the other.

$$x : []^{t_x}/(I_{\mathbf{o}}.0||O_{\mathbf{c}}.0); \emptyset \vdash x?[].\, x![]$$
$$x : []^{t_x}/(I_{\mathbf{o}}.O_{\mathbf{c}}.0); \emptyset \vdash x?[].\, x![]$$

We summarize below the key new ideas of this paper that enable type reconstruction.

Constraint-based type system One of the key ideas is to use a constraint-based type system, following Igarashi and Kobayashi's type reconstruction algorithm [8] for the linear π-calculus. By introducing variables ranging over types, usages, etc. and constraints on them, we can express all the possible typing for a process P by a generalized judgment of the form $\Gamma; \mathcal{T}; C \vdash P$ where Γ may contain variables and C is the set of constraints on the variables. For example, all the possible typings for the process $x?[].\, y![]$ can be represented by:

$$x : []^{t_x}/(I_{a_x}.\alpha_1||\alpha_2), y : []^{t_y}/(O_{a_y}.\alpha_3||\alpha_4); \mathcal{T}; \{\mathbf{o} \in a_y \Rightarrow (\mathbf{c} \in a_x \wedge t_x \mathcal{T} t_y), noob(\alpha_1), \cdots, noob(\alpha_4)\} \vdash x?[].\, y![]$$

Here, α_is are variables ranging over usages and a_x and a_y are variables ranging over attributes of usages. A constraint $\mathbf{o} \in a$ ($\mathbf{c} \in a$, resp.) means that a contains at least an obligation (a capability, resp.), i.e., a is either $\mathbf{o}$ or $\mathbf{co}$. $noob(\alpha)$ means that the usage α is not annotated with an obligation. Each type judgment is obtained by solving the set of constraints, and substituting a solution for the variables.

Subusage relation To make the above constraint-based system work, we need to introduce an appropriate relation on usages. Recall the type judgments for the process $x?[\,].\,x![\,]$. In a certain sense, $I_{\mathbf{o}}.0||O_{\mathbf{c}}.0$ allows a more liberal usage than $I_{\mathbf{o}}.O_{\mathbf{c}}.0$: the former usage means that the output capability may be used before the input obligation is fulfilled, while the latter one means that the output capability may be used only after the input obligation is fulfilled. So, it is OK to use a channel of the former usage according to the latter usage, but the converse is not allowed. In order to express this kind of relationship between usages, we introduce a *subusage relation* $U_1 \leq U_2$: if $U_1 \leq U_2$ holds, then a channel of the usage U_1 may be used as that of the usage U_2.

By using the subusage relation, the possible typings for the process $x?[\,].\,x![\,]$ is expressed by:

$$x : []^{t_x}/\alpha; \mathcal{T}; \{\alpha \leq I_{a_1}.O_{a_2}.0\} \vdash x?[\,].\,x![\,]$$

Recursive usages In addition to the introduction of the subusage relation, we also need to generalize the usages. Consider, for example, a channel that is used for input an arbitrary number of times sequentially. There are infinitely many usages that can be assigned to the channel: $*I.0$, $I.*I.0$, $I.I.*I.0$,... However, none is the most general (i.e., the greatest with respect to the subusage relation). We, therefore, introduce *recursive* usages of the form $\mathbf{rec}\,\alpha.U$. $\mathbf{rec}\,\alpha.U$ is intended to satisfy $\mathbf{rec}\,\alpha.U = [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$. We can now express the greatest usage of the above channel as $\mathbf{rec}\,\alpha.(I.\alpha)$. In order to deal with recursive usages, we define the subusage relation co-inductively, by using familiar techniques for defining process equivalence in terms of bisimulation [12].

Annotations for specifying programmers' intention By using the above ideas, we can now develop a type reconstruction algorithm, which inputs an unannotated process, outputs the most general type judgment with a set of constraints, and decides whether the constraint set is actually satisfiable. However, this is not so useful, because, without any type annotation, we can always assume that there are no capabilities and obligations — then, the typability of a process breaks down to its typability in the simply-typed π-calculus [5]. What we actually want to do is to let a programmer specify his or her intention on the uses of channels, and to check whether it is indeed observed by the program. One way for doing so would be to let programmers specify the types of channels partially. Instead of doing so, in order to simplify the problem of type reconstruction, we let programmers annotate each input/output operation with a capability and/or an obligation if necessary. For example, $x?^{\mathbf{c}}[\,].\,P$ indicates that this input operation must be guaranteed to succeed (so, some external process must output on x). $y?[\,].\,x!^{\mathbf{o}}[\,].\,0$ indicates that the output on x must be performed (so, the input on y must succeed). We reformalize the type system, so that such annotations are respected by any well-typed processes. In the new type system, if a closed process $(\nu x)\,(x?^{\mathbf{c}}[\,].\,P \mid Q)$ is well typed and Q does not diverge, then Q is guaranteed to eventually send a value on x, so that the input $x?^{\mathbf{c}}[\,].\,P$ can succeed. Recall the example of the factorial function. In order to call the function, a programmer can write $(\nu r)\,(fact![3, r] \mid r?^{\mathbf{c}}[n].\,P)$. Then, it is guaranteed that the result will eventually be received on r. (Note here that $fact![3, r]$ need not be annotated with $\mathbf{c}$. Programmers need to attach annotations only to the output/input operations they are concerned with; obligations and capabilities of the other output/input operations are automatically inferred.)

3 Usages, Types, and Processes

We now introduce the formal syntax of usages, types and processes, and then define an operational semantics.

3.1 Usage

Attributes of usages are defined below. As explained in Section 2, **c** denotes a capability and **o** denotes an obligation. They are used for annotating usages and processes.

Definition 3.1 [usage attributes]: *usage attributes* are subsets of $\mathbf{co} = \{\mathtt{c}, \mathtt{o}\}$. We often write just **c** and **o** for $\{\mathtt{c}\}$ and $\{\mathtt{o}\}$ respectively.

We use a metavariable a for usage attributes.

Definition 3.2 [usages]: The set $\mathcal{U}$ of usages is given by the following syntax.

$$U ::= 0 \mid \alpha \mid O_a.U \mid I_a.U \mid (U_1||U_2) \mid U_1 \sqcap U_2 \mid \mathbf{rec}\,\alpha.U \mid *U$$

Here, α ranges over a countably infinite set of variables called *usage variables*.

Notation 3.3: $\mathbf{rec}\,\alpha.U$ binds α in U. Usage variables that are not bound are called *free* usage variables. We assume that α-conversions are implicitly applied so that bound variables are always different from each other and from free variables. $[\alpha \mapsto U']U$ denotes the usage obtained from U by replacing all free occurrences of α with U'. We write $\mathbf{UV}(U)$ for the set of free usage variables appearing in U. We often omit 0 and write O_a for $O_a.0$ and I_a for $I_a.0$.

We give a higher precedence to prefixes ($I_a.$, $O_a.$, $\mathbf{rec}\,\alpha.$) than to $||$ and $\sqcap$. We also give a higher precedence to $\sqcap$ than to $||$. So, $I_a.U_1||O_a.U_2 \sqcap U_3$ means $(I_a.U_1)||((O_a.U_2) \sqcap U_3)$.

0 denotes the usage of a channel that cannot be used for input or output at all. $O_a.U$ denotes the usage of a channel that can be first used for output, and then used according to U. If the attribute a contains **c**, then the output is a capability: if the output is performed, it will eventually succeed (i.e., some other process will eventually receive the output value). If the attribute a contains **o**, then the input is an obligation: the input *must* be performed. $I_a.U$ denotes the usage of a channel that can be first used for input, and then used according to U. Similarly to the case for output, a indicates whether the input is a capability and/or an obligation. $U_1||U_2$ denotes the usage of a channel that can be used according to U_1 and U_2 concurrently. For example, if x is a channel of the usage $I_\emptyset.0||O_\emptyset.0$, then any of the following uses is allowed: (i) one process performs an output on x while another process performs an input on x in parallel, (ii) one process performs an output on x and then performs an input on x, or (iii) one process performs an input on x and then performs an output on x. Note that in the case of the usage $I_\emptyset.O_\emptyset.0$, only (iii) is allowed. $U_1 \sqcap U_2$ denotes the usage of a channel that can be used according to either U_1 or U_2. For example, if x is a channel of the usage $I_\emptyset.0 \sqcap O_\emptyset.0$, then x can be used either for input or for output, but not for both. $\mathbf{rec}\,\alpha.U$ denotes the usage of a channel that can be used according to the infinite expansion of $\mathbf{rec}\,\alpha.U$ by $\mathbf{rec}\,\alpha.U = [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$. For example, $\mathbf{rec}\,\alpha.I_\emptyset.\alpha$ denotes the usage of a channel that can be used for input an infinite number of times sequentially. $*U$ denotes the usage of a channel that can be used according to U by infinitely many processes. $*U$ is almost the same as $\mathbf{rec}\,\alpha.(\alpha||U)$, but we keep them separate for a subtle technical reason (see Remark 4.5).

Example 3.4: The usage of a binary semaphore is denoted by $O_{\mathbf{o}}.0||*I_{\mathbf{c}}.O_{\mathbf{o}}.0$, meaning that there must be one initial output, and there can be infinitely many input processes on the channel.

Example 3.5: The usage of a channel used as the location of a function is denoted by: $*I_{\mathbf{o}}.0||*O_{\mathbf{c}}.0$.

3.2 Types

As explained in Section 2, time tags introduced below are used for expressing in which order the capability of each channel can be used the obligation must be are fulfilled.

Definition 3.6 [time tags]: $\mathbf{T}$ is a countably infinite set of elements called *time tags*.

We use a metavariable t for a time tag.

Definition 3.7 [types]: The set of types is given by the following syntax.

$$\tau ::= \mathit{bool} \mid [\tau_1, \ldots, \tau_n]^t/U$$

$[\tau_1, \ldots, \tau_n]^t/U$ denotes the type of a channel that can be used for communicating a tuple of values of types $\tau_1, \ldots, \tau_n$. The channel must be used according to the usage U. t expresses in what timing its capabilities can be used and its obligations must be fulfilled. (The timing is determined by the time tags of other channels and a tag ordering introduced later in Section 4.)

3.3 Processes

We use the following subset of the π-calculus processes.

Definition 3.8 [processes]: The set of processes is given by the following syntax.

$$\begin{array}{lll} P ::= & \mathbf{0} \mid x!^a[v_1, \ldots, v_n].\, P \mid x?^a[y_1, \ldots, y_n].\, P \mid (P \mid Q) \mid (\nu x)\, P \\ & \mid \textbf{if } v \textbf{ then } P \textbf{ else } Q \mid *P \\ v ::= & \mathit{true} \mid \mathit{false} \mid x \end{array}$$

Here, x and y_is range over a countably infinite set of variables.

Notation 3.9: As usual, $y_1, \ldots, y_n$ in $x?[y_1, \ldots, y_n].\, P$ and x in $(\nu x)\, P$ 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. $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]P$ denotes a process obtained from P by by replacing all free occurrences of $x_1, \ldots, x_n$ with $v_1, \ldots, v_n$. We write $\tilde{y}$ for a sequence $y_1, \ldots, y_n$. We often write $x!^a[\tilde{y}]$ for $x!^a[\tilde{y}].\, \mathbf{0}$. We often omit the empty attribute $\emptyset$ and just write $x![\tilde{y}].\, P$ and $x?[\tilde{y}].\, P$ for $x!^{\emptyset}[\tilde{y}].\, P$ and $x?^{\emptyset}[\tilde{y}].\, P$ respectively.

$\mathbf{0}$ denotes inaction. $x!^a[v_1, \ldots, v_n].\, P$ denotes a process that sends a tuple $[v_1, \ldots, v_n]$ on x and then (after the tuple is received by some process) behaves like P. If a contains $\mathbf{c}$, then once this process is scheduled and the tuple is sent, the tuple must be received. If a contains $\mathbf{o}$ and if this process is blocked by input or output prefixes on other channels (for example, if it appears in the form $Q = y?[\,].\, x!^a[v_1, \ldots, v_n].\, P$ and the entire process Q is scheduled), then it must be scheduled and the tuple must be sent on x. $x?^a[y_1, \ldots, y_n].\, P$ denotes a process that receives a tuple $[v_1, \ldots, v_n]$ on x and then behaves like $[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n]P$. $P \mid Q$ denotes a concurrent execution of P and Q, and $(\nu x)\, P$ denotes a process that creates a fresh channel x and then behaves like P. **if** v **then** P **else** Q behaves like P if v is *true* and behaves like Q if v is *false*; otherwise it is blocked forever. $*P$ represents infinitely many copies of the process P.

3.4 Operational Semantics

Following the standard reduction semantics for the π-calculus [13], we define an operational semantics by using two relations: a structural congruence relation and a reduction relation.

We first give the structural congruence relation below. For some technical reason, unlike in the standard semantics, we do not have $*P \equiv P \mid *P$. Instead, we will introduce a rule for the reduction relation (the (R-Rep) below) that allows $*P$ to be reduced in the same way as $P \mid *P$.

Definition 3.10: The *structural congruence relation* $\equiv$ is the least congruence relation closed under the following rules:

$$P \mid \mathbf{0} \equiv P \qquad \text{(SCong-Zero)}$$

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

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

$$(\nu x)\,(P \mid Q) \equiv (\nu x)\,P \mid Q \text{ (if } x \text{ is not free in } Q) \qquad \text{(SCong-New)}$$

Following the operational semantics of the linear π-calculus [11], we define the reduction relation as a ternary relation $P \xrightarrow{l} Q$. l indicates on which channel the reduction is performed: l is either ϵ, which means that the reduction is performed by communication on an internal channel or by the reduction of a conditional expression, or x, which means that the reduction is performed by communication on the free channel x.

Definition 3.11: The reduction relation $\xrightarrow{l}$ is the least relation closed under the following rules:

$$x!^{a}[v_1, \ldots, v_n].\,P \mid x?^{a'}[z_1, \ldots, z_n].\,Q \xrightarrow{x} P \mid [z_1 \mapsto v_1, \ldots, z_n \mapsto v_n]Q \qquad \text{(R-Com)}$$

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

$$\frac{P \xrightarrow{x} Q}{(\nu x)\,P \xrightarrow{\epsilon} (\nu x)\,Q} \qquad \text{(R-New1)}$$

$$\frac{P \xrightarrow{l} Q \qquad l \neq x}{(\nu x)\,P \xrightarrow{l} (\nu x)\,Q} \qquad \text{(R-New2)}$$

$$\textbf{if } \mathit{true} \textbf{ then } P \textbf{ else } Q \xrightarrow{\epsilon} P \qquad \text{(R-IfT)}$$

$$\textbf{if } \mathit{false} \textbf{ then } P \textbf{ else } Q \xrightarrow{\epsilon} Q \qquad \text{(R-IfF)}$$

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

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

4 Typing

This section defines a type system for our process calculus. We first redefine the reliability of usages (which was first introduced in [23]) in Section 4.1. As explained in Section 2, the reliability expresses channel-wise consistency of capabilities and obligations. Second, we define a subusage relation $U_1 \leq U_2$ and a subtyping relation $\tau_1 \leq \tau_2$ in Section 4.2. The subusage relation $U_1 \leq U_2$ is defined to hold if a channel of the usage U_1 may be used as a channel of the usage U_2. By using those relations, we give typing rules in Sections 4.3 and 4.4. Finally, we show in Section 4.5 that the type system is sound in the sense that a well-typed process does not deadlock: a input/output process having a capability will eventually find its communication partner and be reduced (unless the whole process diverges).

4.1 Reliability of Usages

We first introduce a relation $U_1 \succeq U_2$, which means that the usage U_2 is obtained by the expansion of a recursive usage, a choice of U_1 or U_2 from $U_1 \sqcap U_2$, etc. Readers who are familiar with the reduction semantics of the π-calculus [13] would notice that $\succeq$ plays a role similar to that played by the structural congruence of processes.

Definition 4.1: $\succeq$ is the least reflexive and transitive relation satisfying the following rules:

$$U \succeq U||0$$

$$U_1||U_2 \succeq U_2||U_1$$

$$(U_1||U_2)||U_3 \succeq U_1||(U_2||U_3)$$

$$U_1||(U_2||U_3) \succeq (U_1||U_2)||U_3$$

$$U_1 \sqcap U_2 \succeq U_i$$

$$\mathbf{rec}\,\alpha.U \succeq [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$$

$$*U \succeq *U||U$$

$$\frac{U_1 \succeq V_1 \qquad U_2 \succeq V_2}{U_1||U_2 \succeq V_1||V_2}$$

Remark 4.2: The rule $U_1||(U_2||U_3) \succeq (U_1||U_2)||U_3$ is actually unnecessary. It is derivable by:

$$U_1||(U_2||U_3) \succeq (U_2||U_3)||U_1 \succeq U_2||(U_3||U_1) \succeq (U_3||U_1)||U_2 \succeq U_3||(U_1||U_2) \succeq (U_1||U_2)||U_3$$

The unary relation $ob_{\mathbf{I}}(U)$ below means that the usage U contains an input obligation and that there is no way to discard the obligation. We need a relation $\succeq'$, which is slightly different from $\succeq$.

Definition 4.3: $\succeq'$ is the least reflexive and transitive relation satisfying the following rules:

$$U \succeq' U||0$$

$$U_1||U_2 \succeq' U_2||U_1$$

$$(U_1||U_2)||U_3 \succeq' U_1||(U_2||U_3)$$

$$U_1||(U_2||U_3) \succeq' (U_1||U_2)||U_3$$

$$U_1 \sqcap U_2 \succeq' U_i$$

$$\mathbf{rec}\,\alpha.U \succeq' [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$$

$$*U \succeq' *U||U$$

$$\frac{U_1 \succeq' V_1 \qquad U_2 \succeq' V_2}{U_1||U_2 \succeq' V_1||V_2}$$

$$\frac{U \succeq' V}{*U \succeq' *V}$$

Definition 4.4: Unary predicates $ob_{\mathbf{I}}, ob_{\mathbf{O}}(\subseteq \mathcal{U})$ on usages are defined by:

$$ob_{\mathbf{I}}(U) \Longleftrightarrow \forall U_1.(U \succeq' U_1 \Rightarrow \exists a, U_2, U_3.((U_1 \succeq I_a.U_2||U_3) \wedge (\mathbf{o} \subseteq a)))$$

$$ob_{\mathbf{O}}(U) \Longleftrightarrow \forall U_1.(U \succeq' U_1 \Rightarrow \exists a, U_2, U_3.((U_1 \succeq O_a.U_2||U_3) \wedge (\mathbf{o} \subseteq a)))$$

For example, $ob_{\mathbf{I}}(I_{\mathbf{o}}.0)$ holds, but $ob_{\mathbf{I}}(I_{\mathbf{o}}.0 \sqcap I_{\emptyset}.0)$ does not hold: $I_{\emptyset}.0$ can be chosen from the usage $I_{\mathbf{o}}.0 \sqcap I_{\emptyset}.0$, so that the input obligation can be discarded. $ob_{\mathbf{I}}(I_{\emptyset}.I_{\mathbf{o}}.0)$ does not hold either, because the input obligation arises only after a channel of the usage is used for input. The definition for recursive usages is subtle: $ob_{\mathbf{I}}(\mathbf{rec}\,\alpha.(I_{\mathbf{o}}.0 \sqcap \alpha)$ is defined to hold, because it contains an input obligation and the obligation remains even after the righthand side of $\sqcap$ is chosen.

Remark 4.5: $ob_{\mathbf{I}}(*U)$ and $ob_{\mathbf{I}}(\mathbf{rec}\,\alpha.(\alpha||U))$ may be different. Let $U = I_{\mathbf{o}}.0 \sqcap 0$. Then, $ob_{\mathbf{I}}(*U)$ does not hold because $*U \succeq' *0$. On the other hand, $ob_{\mathbf{I}}(\mathbf{rec}\,\alpha.(\alpha||U))$ does hold. This is the reason why we keep $*$ as a primitive usage constructor.

Next, we define the consistency of a usage. Intuitively, a usage being consistent means that if the usage is offering an input/output capability, it must be imposing the corresponding output/input obligation.

Definition 4.6 [consistency]: A usage U is *consistent*, written $con(U)$, if it satisfies the following conditions:

1. If $U \succeq I_a.U_1||U_2$ and $\mathbf{c} \subseteq a$, then $ob_{\mathbf{O}}(U_2)$.
2. If $U \succeq O_a.U_1||U_2$ and $\mathbf{c} \subseteq a$, then $ob_{\mathbf{I}}(U_2)$.

The consistency defined above is only concerned with the current consistency between capabilities and obligations; it does not care about the state after the channel is used for communication. For example, the usage $I_{\mathbf{c}}.I_{\mathbf{c}}.0||O_{\mathbf{o}}.0$ is consistent although a channel of the usage is used as that of $I_{\mathbf{c}}.0$ after the channel is used for communication. Following the previous type system [23], we refer to the consistency of a usage during the whole reduction by a term *reliability*. We define the reliability after introducing a reduction relation $U \longrightarrow U'$ on usages. Intuitively, $U \longrightarrow U'$ means that a channel of the usage U may be used as that of U' after the channel is used for communication.

Definition 4.7: $U \longrightarrow U'$ is the least relation closed under the following rules:

$$I_{a_1}.U_1||O_{a_2}.U_2||U_3 \longrightarrow U_1||U_2||U_3$$

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

$\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

Definition 4.8 [reliability]: A usage U is *reliable*, written $rel(U)$, if $con(U')$ for every U' such that $U \longrightarrow^* U'$.

4.2 Subusage and subtyping

A channel of one usage may be used as that of another usage. For example, a channel of the usage $I_\emptyset.0||I_\emptyset.0$ can be used as that of the usage $I_\emptyset.I_\emptyset.0$, because the former usage expresses a more liberal use of the channel. In this section, we define such a relation as a subusage relation $U_1 \leq U_2$, and also define an induced relation on types as a subtype relation $\tau_1 \leq \tau_2$.

We first define a *sub-attribute* relation $a_1 \leq a_2$, meaning that input/output with the attribute a_1 may be viewed as that with the attribute a_2. For the consistency of capabilities and obligations, a capability may be ignored but an obligation must not.

Definition 4.9 [sub-attribute]: The relation $\leq$ is the least partial order satisfying $\mathbf{c} \leq \emptyset$ and $\mathbf{co} \leq \mathbf{o}$.

We now define a subusage relation. We define it co-inductively, using the following *usage simulation*. It is inspired from a standard definition of process equivalence in terms of bisimulation (see [12] for example). It also resembles Jim and Palsberg's definition for subtyping recursive types using simulations [9].

Definition 4.10 [usage simulation]: A binary relation $\mathcal{R}(\subseteq \mathcal{U} \times \mathcal{U})$ on usages is called a *usage simulation* if the following conditions are satisfied for each $(U, U') \in \mathcal{R}$:

1. If $U' \succeq I_{a'}.U'_1||U'_2$, then there exist U_1, U_2, and a such that (i)$U \succeq I_a.U_1||U_2$, (ii)$U_2 \mathcal{R} U'_2$, (iii)$(U_1||U_2)\mathcal{R}(U'_1||U'_2)$, and (iv)$a \leq a'$.
2. If $U' \succeq O_{a'}.U'_1||U'_2$, then there exist U_1, U_2, and a such that (i)$U \succeq O_a.U_1||U_2$, (ii)$U_2 \mathcal{R} U'_2$, (iii)$(U_1||U_2)\mathcal{R}(U'_1||U'_2)$, and (iv)$a \leq a'$.
3. If $U' \longrightarrow U'_1$, then there exists U_1 such that $U \longrightarrow U_1$ and $U_1 \mathcal{R} U'_1$.
4. $ob_{\mathbf{I}}(U)$ implies $ob_{\mathbf{I}}(U')$.
5. $ob_{\mathbf{O}}(U)$ implies $ob(U')$.

The first and second conditions mean that in order for U to simulate U', U must allow any input/output operations that U' allows. The fourth and fifth conditions mean that U' must provide any obligations that U provides. The third condition means that such conditions are preserved even after reductions.

Definition 4.11 [subusage]: A *subusage relation* $\leq$ on usages is $\bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a usage simulation.}\}$.

It is straightforward that $\leq$ is the largest usage simulation. It is also trivial from the definition that $\leq$ is a preorder (i.e., a reflexive and transitive relation).

Example 4.12: $I_{\mathbf{c}}.U \leq 0$ holds because the relation

$$\{(I_{\mathbf{c}}.U, 0)\}$$

is a usage simulation. On the other hand, $I_{\mathbf{o}}.U \leq 0$ does not hold: The pair $(I_{\mathbf{o}}.U, 0)$ does not satisfy the fourth condition of Definition 4.10.

Example 4.13: $I_{\mathbf{c}}.0||I_{\mathbf{c}}.0 \leq I_{\mathbf{c}}.I_{\mathbf{c}}.0$ holds because the relation:

$$\{(I_{\mathbf{c}}.0||I_{\mathbf{c}}.0, I_{\mathbf{c}}.I_{\mathbf{c}}.0)\} \cup \{(0||I_{\mathbf{c}}.0, U) \mid I_{\mathbf{c}}.0 \succeq U\} \cup \{(I_{\mathbf{c}}.0, U) \mid 0 \succeq U\} \cup \{(U_1, U_2) \mid 0 \succeq U_1 \wedge 0 \succeq U_2\}$$

is a usage simulation. On the other hand, $I_{\mathbf{o}}.0||I_{\mathbf{c}}.0 \leq I_{\mathbf{c}}.I_{\mathbf{o}}.0$ does not hold, because the pair $(I_{\mathbf{o}}.0||I_{\mathbf{c}}.0, I_{\mathbf{c}}.I_{\mathbf{o}}.0)$ does not satisfy the fourth condition of Definition 4.10.

Based on the subusage relation, we can also introduce a subtype relation. We could introduce structural subtyping as in [16], but we don't do so for the simplicity; such an extension is discussed in Section 7.

Definition 4.14 [subtyping]: The subtyping relation $\leq$ is the least relation closed under the following rules:

$$bool \leq bool \qquad \text{(SubT-Bool)}$$

$$\frac{U \leq U'}{[\tau_1, \ldots, \tau_n]^t/U \leq [\tau_1, \ldots, \tau_n]^t/U'} \qquad \text{(SubT-Chan)}$$

The unary predicates $noob(U)$ and $noob(\tau)$ below means that U and τ is representing no input/output obligations. In other words, if x has type τ such that $noob(\tau)$, x need not be used.

Definition 4.15: $noob(U)$ is defined to hold if and only if $U \leq 0$. $noob(\tau)$ is defined by:

$$noob(\tau) \Longleftrightarrow (\tau = bool) \vee (\tau = [\tau_1, \ldots, \tau_n]^t/U \wedge noob(U)).$$

We write $ob(U)$ ($ob(\tau)$, resp.) if $noob(U)$ ($noob(\tau)$, resp.) does not hold.

Lemma 4.16: $noob(U)$ holds if and only if neither $ob_{\mathbf{I}}(U)$ nor $ob_{\mathbf{O}}(U)$ holds

Proof: If $noob(U)$, i.e., $U \leq 0$ holds, then the pair $(U, 0)$ must satisfy the conditions of Definition 4.10. Since neither $ob_{\mathbf{I}}(0)$ nor $ob_{\mathbf{I}}(0)$ holds, by the fourth and fifth conditions, it must also be the case that neither $ob_{\mathbf{I}}(U)$ nor $ob_{\mathbf{O}}(U)$ holds.

On the other hand, suppose that neither $ob_{\mathbf{I}}(U)$ nor $ob_{\mathbf{O}}(U)$ holds. Then, $U \leq 0$ holds, because the relation $\{(U, 0)\}$ is a usage simulation. □

Properties of the Subusage Relation $\leq$

In the rest of this subsection, we study properties of the subusage relation. Readers who are not interested in proofs can safely skip to Section 4.3.

We first introduce a proof technique, which corresponds to a familiar proof technique of "bisimulation up to" in process calculi.

Definition 4.17 [usage simulation up to]: A binary relation $\mathcal{R}(\subseteq \mathcal{U} \times \mathcal{U})$ on usages is a *usage simulation up to* $\leq$ if the following conditions are satisfied for each $(P, Q) \in \mathcal{R}$,

1. If $U' \succeq I_{a'}.U_1'||U_2'$, then there exist U_1, U_2, and a such that (i)$U \succeq I_a.U_1||U_2$, (ii)$U_2 \leq \mathcal{R} \leq U_2'$, (iii)$(U_1||U_2) \leq \mathcal{R} \leq (U_1'||U_2')$, and (iv)$a \leq a'$.
2. If $U' \succeq O_{a'}.U_1'||U_2'$, then there exist U_1, U_2, and a such that (i)$U \succeq O_a.U_1||U_2$, (ii)$U_2 \leq \mathcal{R} \leq U_2'$, (iii)$(U_1||U_2) \leq \mathcal{R} \leq (U_1'||U_2')$, and (iv)$a \leq a'$.
3. If $U' \longrightarrow U_1'$, then there exists U_1 such that $U \longrightarrow U_1$ and $U_1 \leq \mathcal{R} \leq U_1'$.
4. $ob_{\mathbf{I}}(U)$ implies $ob_{\mathbf{I}}(U')$.
5. $ob_{\mathbf{O}}(U)$ implies $ob_{\mathbf{O}}(U')$.

The following theorem implies that in order to show $U_1 \leq U_2$, it suffices to prove that there exists a usage simulation up to $\leq$ containing the pair (U_1, U_2).

Theorem 4.18: If $\mathcal{R}$ is a usage simulation up to $\leq$, then $\mathcal{R} \subseteq \leq$.

Proof: This follows from the fact that $\mathcal{R} \cup (\leq \mathcal{R} \leq)$ is a usage simulation. □

We now check several properties of $\leq$ by using Theorem 4.18. Among others, important properties are (1) the subusage relation is closed under usage constructors such as $I_a.$, $O_a.$, $||$, $\sqcap$, and $*$ (Lemmas 4.21 and 4.23), (2) $U_1 \sqcap U_2$ is the least upper bound of U_1 and U_2 with respect to the subusage relation (the 4th and 5th properties of Lemma 4.24), (3) $\mathbf{rec}\,\alpha.U$ is a fixpoint of $\lambda\alpha.U$, with respect to the least equivalence relation containing $\leq$ (the 6th property of Lemma 4.24), and (4) if U is reliable and $U \leq U'$, then U' is also reliable (Lemma 4.27).

Lemma 4.19: If $U \succeq U'$, then $U \leq U'$.

Proof: It suffices to show that $\mathcal{R} = \leq \cup\{(U_1, U_2) \mid U_1 \succeq U_2\}$ is a usage simulation. Assuming $U_1 \mathcal{R} U_2$, we show each condition in Definition 4.10. The case for $U_1 \leq U_2$ is trivial. So, assume $U_1 \succeq U_2$. The conditions 1-3 are trivial. Suppose $ob_{\mathbf{I}}(U_1)$ and $U_2 \succeq U_2'$. Since $U_1 \succeq U_2 \succeq U_2'$, we have $U_2' \succeq O_a.U_3||U_4$ and $\mathbf{o} \leq a$ for some a, U_3, and U_4. So, we have $ob_{\mathbf{I}}(U_2)$. Similarly, the condition 5 also holds. □

Lemma 4.20: $ob_{\mathbf{O}}(I_a.U)$ never holds. $ob_{\mathbf{O}}(O_a.U)$ holds if and only if $\mathbf{o} \subseteq a$. $ob_{\mathbf{O}}(U_1||U_2)$ holds if and only if $ob_{\mathbf{O}}(U_1) \vee ob_{\mathbf{O}}(U_2)$ holds. $ob_{\mathbf{O}}(U_1 \sqcap U_2)$ holds if and only if $ob_{\mathbf{O}}(U_1) \wedge ob_{\mathbf{O}}(U_2)$ holds. $ob_{\mathbf{O}}(*U)$ holds if and only if $ob_{\mathbf{O}}(U)$ holds. Similar properties hold also for $ob_{\mathbf{I}}$.

Proof: Trivial by the definition of $ob_{\mathbf{O}}$ and $ob_{\mathbf{I}}$. □

Lemma 4.21: $\leq$ is closed under the usage constructors $I_a.$, $O_a.$, $||$ and $\sqcap$, i.e.,

- $U \leq U'$ implies $I_a.U \leq I_a.U'$,
- $U \leq U'$ implies $O_a.U \leq O_a.U'$,
- $U_1 \leq U_1'$ and $U_2 \leq U_2'$ imply $U_1||U_2 \leq U_1'||U_2'$, and
- $U_1 \leq U_1'$ and $U_2 \leq U_2'$ imply $U_1 \sqcap U_2 \leq U_1' \sqcap U_2'$.

Proof: It suffices to show that the relation

$$\begin{aligned}\mathcal{R} = \; & \leq \\ & \cup\{(O_a.U, O_a.U') \mid U \leq U'\} \\ & \cup\{(I_a.U, I_a.U') \mid U \leq U'\} \\ & \cup\{(U_1||U_2, U_1'||U_2') \mid U_1 \leq U_1' \text{ and } U_2 \leq U_2'\} \\ & \cup\{(U_1 \sqcap U_2, U_1' \sqcap U_2') \mid U_1 \leq U_1' \text{ and } U_2 \leq U_2'\}\end{aligned}$$

is a usage simulation up to $\leq$. We need to show the conditions in Definition 4.17 for each pair (U, U') in $\mathcal{R}$. Since the cases where (U, U') is in the first, second, or third set is trivial, we show only the other cases.

- Case where (U, U') is in the fourth set: It must be the case that $U = U_1||U_2$, $U' = U_1'||U_2'$, and $U_i \leq U_i'$ for $i = 1, 2$.

– Condition 1: Suppose $U' \succeq I_{a'}.U'_3||U'_4$. Then, $U'_i \succeq I_{a'}.U'_3||U'_5$ and $U'_5||U'_j \succeq U'_4$ for $(i,j) = (1,2)$ or $(2,1)$. Since $U_i \leq U'_i$ and $U_j \leq U'_j$, we have $U \succeq I_a.U_3||(U_5||U_j)$, $U_5 \leq U'_5$, and $U_3||U_5 \leq U'_3||U'_5$. So, we have $U_5||U_j\mathcal{R}U'_5||U'_j \leq U'_4$ and $U_3||(U_5||U_j) \leq (U_3||U_5)||U_j\mathcal{R}(U'_3||U'_5)||U'_j \leq U'_3||U'_4$, as required.

– Condition 2: Similar to the condition 1.

– Condition 3: Suppose $U' \longrightarrow U'_3$. Then, either (i)$U'_1 \longrightarrow U'_4$ and $U'_4||U'_2 \succeq U'_3$, (ii)$U'_2 \longrightarrow U'_4$ and $U'_4||U'_1 \succeq U'_3$, or (iii)$U'_i \succeq I_{a'_1}.U'_4||U'_5$, $U'_j \succeq O_{a'_2}.U'_6||U'_7$, and $U'_4||U'_5||U'_6||U'_7 \succeq U'_3$ for $(i,j) = (1,2)$ or $(2,1)$. Because the case (ii) is similar, we show only the case (i) and (iii).

  * Case (i): By $U_1 \leq U'_1$ and $U'_1 \longrightarrow U'_4$, there must exist U_4 such that $U_1 \longrightarrow U_4$ and $U_4 \leq U'_4$. Let $U_3 = U_4||U_2$. Then, we have $U = U_1||U_2 \longrightarrow U_4||U_2$ and $U_4||U_2\mathcal{R}U'_4||U'_2 \leq U'_3$ as required.
  * Case (iii): By assumptions, it must be the case that:

$$\begin{array}{l} U_i \succeq I_{a_1}.U_4||U_5 \\ U_j \succeq O_{a_2}.U_6||U_7 \\ U_5 \leq U'_5 \\ U_7 \leq U'_7 \\ U_4||U_5 \leq U'_4||U'_5 \\ U_6||U_7 \leq U'_6||U'_7 \\ a_1 \leq a'_1 \\ a_2 \leq a'_2. \end{array}$$

  Let $U_3 = (U_4||U_5)||(U_6||U_7)$. Then, we have $U = U_1||U_2 \longrightarrow U_3$ and $U_3\mathcal{R}(U'_4||U'_5)||(U'_6||U'_7) \leq U'_3$ as required.

– Conditions 4 and 5: Follow immediately from Lemma 4.20.

- Case where (U,U') is in the fifth set: It must be the case that $U = U_1 \sqcap U_2$, $U' = U'_1 \sqcap U'_2$, and $U_i \leq U'_i$ for $i = 1,2$. The conditions 1–3 are trivial (Notice that $U_1 \sqcap U_2 \succeq I_a.U_3||U_4$ if and only if $U_i \succeq I_a.U_3||U_4$ for $i = 1$ or 2. The conditions 4 and 5 follow from Lemma 4.20.

□

Remark 4.22: $\leq$ is not closed under the recursive usage constructor. For example, let $U = \alpha||O_{\mathbf{o}}.0$ and $U' = O_{\mathbf{o}}.0$. Then, $U \leq U'$ holds, but $\mathbf{rec}\,\alpha.U \leq \mathbf{rec}\,\alpha.U'$ does not.

$\leq$ is also closed under the constructor $*$.

Lemma 4.23: $U \leq U'$ implies $*U \leq *U'$.

Proof: It suffices to show that the relation

$$\mathcal{R} = \leq \cup \{(*U||U'', *U'||U'') \mid U, U', U'' \in \mathcal{U} \text{ and } U \leq U'\}$$

is a usage simulation up to $\leq$. For each pair $(U_1, U'_1) \in \mathcal{R}$, we need to show the conditions in Definition 4.17. The case where the pair is in the first set is trivial, we show only the case where the pair is in the second set. In this case, $U_1 = *U||U''$ and $U'_1 = *U'||U''$.

- Condition 1: Suppose $U_1' \succeq I_{a'}.U_2'||U_3'$. Then, it must be the case that (i) $U' \succeq I_{a'}.U_2'||U_4'$ and $U_1'||U_4' \succeq U_3'$, or (ii) $U'' \succeq I_{a'}.U_2'||U_4'$ and $*U'||U_4' \succeq U_3'$. Since the latter case is trivial, we show only the former case. By $U \leq U'$, there exist U_2, U_4, and a such that $U \succeq I_a.U_2||U_4$, $a \leq a'$, $U_4 \leq U_4'$, and $U_2||U_4 \leq U_2'||U_4'$. So, we have $U_1 \succeq U||U_1 \succeq I_a.U_2||(U_1||U_4)$, and

$$\begin{array}{rcl} U_1||U_4 & \leq & *U||(U''||U_4) \\ & \mathcal{R} & *U'||(U''||U_4) \\ & \leq & U_1'||U_4' \\ & \leq & U_3', \end{array}$$

and

$$\begin{array}{rcl} U_2||(U_1||U_4) & \leq & U_1||(U_2||U_4) \\ & \leq & U_1||(U_2'||U_4') \\ & \leq & *U||(U''||U_2'||U_4') \\ & \mathcal{R} & *U'||(U''||U_2'||U_4') \\ & \leq & U_1'||(U_2'||U_4') \\ & \leq & U_2'||U_3'. \end{array}$$

- Condition 2: Similar to the condition 1.
- Condition 3: Suppose $U_1' \longrightarrow U_2'$. Then, one of the following conditions holds.
  (a) $U' \longrightarrow U_3'$ and $U_3'||U_1' \succeq U_2'$.
  (b) $U'' \longrightarrow U_3'$ and $*U'||U_3' \succeq U_2'$.
  (c) $U' \succeq I_{a_1'}.U_3'||U_4'$, $U'' \succeq O_{a_2'}.U_5'||U_6'$, and $*U'||(U_3'||U_4'||U_5'||U_6') \succeq U_3'$.
  (d) $U' \succeq O_{a_1'}.U_3'||U_4'$, $U'' \succeq I_{a_2'}.U_5'||U_6'$, and $*U'||(U_3'||U_4'||U_5'||U_6') \succeq U_3'$.
  (e) $U'||U' \longrightarrow U_3'$ and $U_3'||U_1' \succeq U_2'$.

  Since the case (d) is similar to the case (c) and the case (e) is similar to the case (a), we show only the cases (a), (b), and (c).

  - Case (a): In this case, $U \longrightarrow U_3$ and $U_3 \leq U_3'$ for some U_3. Let $U_2 = U_3||U_1$. Then, we have $U_1 \longrightarrow U_2$ and

$$\begin{array}{rcl} U_2 & \leq & U_3'||U_1 \\ & \leq & *U||(U_3'||U'') \\ & \mathcal{R} & *U'||(U_3'||U'') \\ & \leq & U_3'||U_1' \\ & \leq & U_2' \end{array}$$

    as required.
  - Case (b): Let $U_2 = *U||U_3'$. Then, we have $U_1 \longrightarrow U_2$ and $U_2 \mathcal{R} *U'||U_3' \leq U_2'$ as required.
  - Case (c): In this case, it must be the case that $U \succeq I_{a_1}.U_3||U_4$, $a_1 \leq a_1'$, $U_4 \leq U_4'$, and $U_3||U_4 \leq U_3'||U_4'$. Let $U_2 = *U||(U_3||U_4||U_5'||U_6')$. Then, we have $U_1 \longrightarrow U_2$ and

$$\begin{array}{rcl} U_2 & \leq & *U||(U_3'||U_4'||U_5'||U_6') \\ & \mathcal{R} & *U'||(U_3'||U_4'||U_5'||U_6') \\ & \leq & U_2' \end{array}$$

    as required.

- Conditions 4 and 5: Follow immediately from Lemma 4.20.

□

Lemma 4.24: 1. $U||0 \leq U$ and $U \leq U||0$.

2. $U_1||U_2 \leq U_2||U_1$
3. $U_1||(U_2||U_3) \leq (U_1||U_2)||U_3$ and $(U_1||U_2)||U_3 \leq U_1||(U_2||U_3)$
4. If $U \leq U_1$ and $U \leq U_2$, then $U \leq U_1 \sqcap U_2$.
5. If $U_1 \sqcap U_2 \leq U_i$ for $i = 1, 2$.
6. $\mathbf{rec}\,\alpha.U \leq [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$ and $[\alpha \mapsto \mathbf{rec}\,\alpha.U]U \leq \mathbf{rec}\,\alpha.U$.
7. $*U||U \leq *U$.
8. If $U_2 \leq 0$, then $O_a.U_1||U_2 \leq O_a.(U_1||U_2)$ and $I_a.U_1||U_2 \leq I_a.(U_1||U_2)$.
9. $*U_1||*U_2 \leq *(U_1||U_2)$.

Proof: 1 $U \leq U||0$ follows from Lemma 4.19. $U||0 \leq$ follows from the fact that $\mathcal{R} = \leq \cup\{(U||0, U) \mid U \in \mathcal{U}\}$ is a usage simulation.

2, 3, 5 These follow from Lemma 4.19.

4 Suppose $U \leq U_1$ and $U \leq U_2$. It suffices to show that $\leq \cup\{(U, U_1 \sqcap U_2)\}$ is a usage simulation. We only need to check the pair $(U, U_1 \sqcap U_2)$. The condition 1 follows immediately from the assumption, since $U_1 \sqcap U_2 \succeq I_a.U_3||U_4$ implies $U_i \succeq I_a.U_3||U_4$ for $i = 1$ or 2. The conditions 2 and 3 follows similarly. Suppose that $ob_{\mathbf{I}}(U)$ holds. Then, by the assumption, it must be the case that $ob_{\mathbf{I}}(U_1)$ and $ob_{\mathbf{I}}(U_2)$. Therefore, $ob_{\mathbf{I}}(U_1 \sqcap U_2)$. The condition 5 follows similarly.

6 $\mathbf{rec}\,\alpha.U \leq [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$ follows from Lemma 4.19. $[\alpha \mapsto \mathbf{rec}\,\alpha.U]U \leq \mathbf{rec}\,\alpha.U$ follows from the fact that $\mathcal{R} = \leq \cup\{([\alpha \mapsto \mathbf{rec}\,\alpha.U]U, \mathbf{rec}\,\alpha.U) \mid U \in \mathcal{U}\}$ is a usage simulation. (Notice that $\mathbf{rec}\,\alpha.U \succeq I_a.U_1||U_2$ implies $[\alpha \mapsto \mathbf{rec}\,\alpha.U]U \succeq I_a.U_1||U_2$.

7 Similar to 6.

8 It suffices to show that

$$\begin{aligned}\mathcal{R} = \; & \leq \cup\{(O_a.U_1||U_2, O_a.(U_1||U_2)) \mid U_1, U_2 \in \mathcal{U} \text{ and } U_2 \leq 0\} \\ & \cup\{(I_a.U_1||U_2, I_a.(U_1||U_2)) \mid U_1, U_2 \in \mathcal{U} \text{ and } U_2 \leq 0\}\end{aligned}$$

is a usage simulation. Assuming $U_1 \mathcal{R} U_2$, we need to show each condition in Definition 4.10. Since the case where $U_1 \leq U_2$ is trivial and the case where (U_1, U_2) is in the third set is similar, we show only the case where (U_1, U_2) is in the second set. In this case, $U_1 = O_a.U_3||U_4$, $U_2 = O_a.(U_3||U_4)$, and $U_4 \leq 0$. The conditions except for 2 and 5 are vacuously true. Suppose $U_2 \succeq O_{a'}.U_5||U_6$. Then, by the definition of $\succeq$, it must be the case that $a = a'$, $U_5 = U_3||U_4$, and $0 \succeq U_6$. So, $U_1 \succeq O_a.U_3||U_4$, $U_4 \leq U_6$, and $U_3||U_4 \leq U_5||U_6$. Therefore, the condition 2 holds. The condition 5 follows from the fact that $ob_{\mathbf{O}}(O_a.U_3||U_4)$ implies $\mathbf{o} \leq a$.

9 It suffices to show that

$$\mathcal{R} =\leq \cup\{(*U_1||*U_2||U_3, *(U_1||U_2)||U_3 \mid U_1, U_2, U_3 \in \mathcal{U}\}$$

is a usage simulation up to $\leq$. Assuming $U\mathcal{R}U'$, we show each condition in Definition 4.10. Since the case where $U \leq U'$ is trivial, we show only the case where (U, U') is in the second set of $\mathcal{R}$. In this case, $U = *U_1||*U_2||U_3$ and $U' = *(U_1||U_2)||U_3$.

- Condition 1: Suppose $U' \succeq I_a.U_4||U_5$. Then, one of the following conditions hold:
  (a) $U_1 \succeq I_a.U_4||U_6$ and $U_6||U_2||U' \succeq U_5$.
  (b) $U_2 \succeq I_a.U_4||U_6$ and $U_6||U_1||U' \succeq U_5'$.
  (c) $U_3 \succeq I_a.U_4||U_6$ and $*(U_1||U_2)||U_6 \succeq U_5$.

  We show only the cases (a) and (c): The case (b) is similar to the case (a).
  * Case (a): $U = *U_1||*U_2||U_3 \succeq I_a.U_4||U_6||U \succeq I_a.U_4||(U||(U_6||U_2))$. Let $U_7 = U||(U_6||U_2)$. Then, we have $U_7\mathcal{R}U'||(U_6||U_2) \leq U_5$ and $U_4||U_7 \leq (U||(U_2||U_4||U_6))\mathcal{R}(U'||(U_2||U_4||U_6)) \leq U_4||U_5$ as required.
  * Case (c): $U = *U_1||*U_2||U_3 \succeq I_a.U_4||(*U_1||*U_2||U_6)$. We have

$$\begin{aligned}(*U_1||*U_2||U_6)\mathcal{R}(*(U_1||U_2)||U_6) &\leq U_5\\ U_4||(*U_1||*U_2||U_6) &\leq *U_1||*U_2||(U_4||U_6)\\ \mathcal{R}*(U_1||U_2)||(U_4||U_6) &\leq U_4||U_5\end{aligned}$$

    as required.
- Condition 2: Similar to the condition 1.
- Condition 3: Suppose $U' \longrightarrow U_4$. By the definition of $\longrightarrow$ and the law 7 of this lemma, we have $U_1||U_1||U_2||U_2||U_3 \longrightarrow U_5$ and $*(U_1||U_2)||U_5 \leq U_4$ for some U_5 (since U_4 can be obtained from U' by expanding $*(U_1||U_2)$ twice, reducing it, and then contracting $*(U_1||U_2)||(U_1||U_2)$ to $*(U_1||U_2)$. So, we have

$$U \succeq U_1||U_1||U_2||U_2||U_3||*U_1||*U_2 \longrightarrow *U_1||*U_2||U_5.$$

  We have

$$(*U_1||*U_2||U_5)\mathcal{R}(*(U_1||U_2)||U_5) \leq U_4$$

  as required.
- Condition 4: Suppose $ob_{\mathbf{I}}(U)$. Then, by Lemma 4.20it must be the case that $ob_{\mathbf{I}}(U_i)$ for $i = 1, 2$, or 3. In either case, we have $ob_{\mathbf{I}}(U')$.
- Condition 5: Similar to the condition 5.

□

Lemma 4.25: If $U_1 \leq U_2$ and U_1 is consistent, then U_2 is also consistent.

Proof: We check the conditions in Definition 4.6.

- Condition 1: Suppose $U_2 \succeq I_{a_2}.U_{21}||U_{22}$ and $\mathbf{c} \subseteq a_2$. By the assumption $U_1 \leq U_2$, it must be the case that $U_1 \succeq I_{a_1}.U_{11}||U_{12}$, $U_{12} \leq U_{22}$ and $a_1 \leq a_2$ for some U_{11}, U_{12}, and a_1. From $a_1 \leq a_2$ and $\mathbf{c} \subseteq a_2$, we get $\mathbf{c} \subseteq a_1$. Since U_1 is consistent, it must be the case that $ob_{\mathbf{O}}(U_{12})$. By the fact $U_{12} \leq U_{22}$, we have $ob_{\mathbf{O}}(U_{22})$.

- Condition 2: Similar to the case for the condition 1.

□

Lemma 4.26: If $U_1 \leq U_2$ and $U_2 \longrightarrow U_2'$, then $U_1 \longrightarrow U_1'$ and $U_1' \leq U_2'$ for some U_1'.

Proof: This follows from the fact that $\leq$ is a usage simulation. □

Lemma 4.27: If $rel(U_1)$ and $U_1 \leq U_2$, then $rel(U_2)$ also holds.

Proof: Suppose that $rel(U_1)$, $U_1 \leq U_2$, and $U_2 \longrightarrow^* U_2'$. It suffices to show that U_2' is consistent. By Lemma 4.26, there exists U_1' such that $U_1 \longrightarrow^* U_1'$ and $U_1' \leq U_2'$. By the definition of $rel(U_1)$, U_1' must be consistent, and hence so is U_2' by Lemma 4.25. □

4.3 Type Environments

This subsection defines type environments as well as a few operations and relations on them.

Definition 4.28 [type environments]: A type environment is a mapping from a finite set of variables to types.

Notation 4.29: We use a metavariable Γ for a type environment. $v_1 : \tau_1, \ldots, v_n : \tau_n$ denotes the type environment Γ such that $dom(\Gamma) = \{v_1, \ldots, v_n\} \backslash \{true, false\}$ and $\Gamma(v_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$ satisfying $v_i \notin \{true, false\}$. (So, $x : \tau, true : bool$ denotes the same type environment as $x : \tau$.) We write $\emptyset$ for the type environment whose domain is empty. When $v \notin dom(\Gamma)$, we write $\Gamma, v : \tau$ for the type environment Γ' satisfying $dom(\Gamma') = dom(\Gamma) \cup (\{v\} \backslash \{true, false\})$, $\Gamma'(v) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for $y \in dom(\Gamma)$. $\Gamma \backslash \{x_1, \ldots, x_n\}$ denotes the type environment Γ' such that $dom(\Gamma') = dom(\Gamma) \backslash \{x_1, \ldots, x_n\}$ and $\Gamma'(x) = \Gamma(x)$ for each $x \in dom(\Gamma')$.

The unary predicate *noob* on types is extended to that on type environments.

Definition 4.30: A unary predicate $noob(\Gamma)$ on type environments is defined by:

$$noob(x_1 : \tau_1, \ldots, x_n : \tau_n) \Longleftrightarrow noob(\tau_i) \text{ for each } i \in \{1, \ldots, n\}$$

We write $ob(\Gamma)$ if $noob(\Gamma)$ does not hold.

The subtyping relation is extended to the following relation on type environments. Intuitively, $\Gamma_1 \leq \Gamma_2$ means that a type environment Γ_1 may be regarded as Γ_2.

Definition 4.31: The *sub-environment* relation $\Gamma_1 \leq \Gamma_2$ holds if and only if the following conditions hold:

1. $dom(\Gamma_1) \supseteq dom(\Gamma_2)$
2. $\forall x \in dom(\Gamma_2).(\Gamma_1(x) \leq \Gamma_2(x))$
3. $\forall x \in dom(\Gamma_1) \backslash dom(\Gamma_2).(noob(\Gamma_1(x)))$

Because deadlock-freedom is sensitive to information on how many times each channel is used for input and output, a type environment cannot be shared among concurrent processes; instead, as in the linear π-calculus [11], the type environment of a parallel composition should be a combination of those of its subprocesses. For example, if a process P is well typed under the type environment $x : []^t/I_{\mathbf{c}}.0$ and another process Q is well typed under the type environment $x : []^t/O_{\mathbf{o}}.0$, then $P \mid Q$ should be well typed under the type environment $x : []^t/(I_{\mathbf{c}}.0||O_{\mathbf{o}}.0)$. Such a type environment is computed by the operation + defined below. + can be regarded as extensions of the operation || on usages to the operations on types and type environments.

Definition 4.32 [Summation of types and type environments]: The summation of two types, written $\tau_1 + \tau_2$, is defined by:

$$\begin{array}{l} bool + bool = bool \\ [\tau_1, \ldots, \tau_n]^t/U_1 + [\tau_1, \ldots, \tau_n]^t/U_2 = [\tau_1, \ldots, \tau_n]^t/(U_1||U_2) \end{array}$$

$\tau_1 + \tau_2$ is undefined otherwise. The summation of two type environments, written $\Gamma_1 + \Gamma_2$, is defined only if $\Gamma_1(x) + \Gamma_2(x)$ is defined for each $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$. It is defined by:

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

The usage constructor $*$ is extended to operations on types and type environments.

Definition 4.33: An operation $*$ on types is defined by:

$$\begin{array}{l} *bool = bool \\ *[\tilde{\tau}]^t/U = [\tilde{\tau}]^t/*U \end{array}$$

(Here, α is a fresh variable.) $*$ is pointwise extended to an operation on type environments by:

$$*(x_1 : \tau_1, \ldots, x_n : \tau_n) = x_1 : *\tau_1, \ldots, x_n : *\tau_n.$$

The subtyping relation and sub-environment relation is closed under the operations + and $*$.

Lemma 4.34: Suppose $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$. Then $\tau_1 + \tau_2$ is well defined if and only if $\tau_1' + \tau_2'$ is well defined. Moreover, if they are well defined, $\tau_1 + \tau_2 \leq \tau_1' + \tau_2'$. Similarly, if $\Gamma_1 \leq \Gamma_1'$ and $\Gamma_2 \leq \Gamma_2'$, then $\Gamma_1 + \Gamma_2$ is well defined if and only if $\Gamma_1' + \Gamma_2'$ is, and $\Gamma_1 + \Gamma_2 \leq \Gamma_1' + \Gamma_2'$ holds if they are well defined.

Proof: Trivial by the definitions of + and $\leq$ and Lemmas 4.24 and 4.21. □

Lemma 4.35: If $\tau \leq \tau'$, then $*\tau \leq *\tau'$. If $\Gamma \leq \Gamma'$, then $*\Gamma \leq *\Gamma'$.

Proof: Trivial by Lemma 4.23 and the definitions of $*\tau$ and $*\Gamma$. □

Definition 4.36: The operations $\sqcap$ on types and type environments are extensions of the operation $\sqcap$ on usages. They are defined by:

$$
\begin{array}{l}
\mathit{bool} \sqcap \mathit{bool} = \mathit{bool} \\
[\tilde{\tau}]^t/U_1 \sqcap [\tilde{\tau}]^t/U_2 = [\tilde{\tau}]^t/(U_1 \sqcap U_2) \\
\mathit{dom}(\Gamma_1 \sqcap \Gamma_2) = \mathit{dom}(\Gamma_1) \cup \mathit{dom}(\Gamma_2) \\
(\Gamma_1 \sqcap \Gamma_2)(x) = \left\{ \begin{array}{ll}
\Gamma_1(x) \sqcap \Gamma_2(x) & \text{if } x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2) \\
\mathit{bool} & \text{if } \Gamma_1(x) = \mathit{bool} \text{ and } x \notin \mathit{dom}(\Gamma_2) \\
[\tilde{\tau}]^t/(U \sqcap 0) & \text{if } \Gamma_1(x) = [\tilde{\tau}]^t/U \text{ and } x \notin \mathit{dom}(\Gamma_2) \\
\mathit{bool} & \text{if } \Gamma_2(x) = \mathit{bool} \text{ and } x \notin \mathit{dom}(\Gamma_1) \\
[\tilde{\tau}]^t/(U \sqcap 0) & \text{if } \Gamma_2(x) = [\tilde{\tau}]^t/U \text{ and } x \notin \mathit{dom}(\Gamma_1)
\end{array} \right.
\end{array}
$$

Lemma 4.37: $\Gamma_1 \sqcap \Gamma_2$ is the greatest lower bound of Γ_1 and Γ_2 with respect to $\leq$, i.e.,

1. If $\Gamma \leq \Gamma_1$ and $\Gamma \leq \Gamma_2$, then $\Gamma \leq \Gamma_1 \sqcap \Gamma_2$.
2. If $\Gamma_1 \sqcap \Gamma_2 \leq \Gamma_i$ for $i = 1, 2$.

Proof: Trivial by Lemma 4.24 and the definition of $\Gamma_1 \sqcap \Gamma_2$. □

4.4 Typing Rules

This subsection introduces the type judgment form and the typing rules for deriving type judgments. Before introducing typing rules, we need to define a *tag ordering*, which is used for controlling the order in which capabilities/obligations of multiple channels are used/fulfilled.

Definition 4.38: A *tag ordering*, written $\mathcal{T}$, is a strict partial order (i.e., a transitive and irreflexive binary relation) on $\mathbf{T}$.

Intuitively, $s\mathcal{T}t$ means that capabilities to communicate on a channel with time tag s may be used before obligations to communicate on a channel with time tag t are fulfilled. In other words, fulfilling obligations to communicate on a channel with time tag t can be delayed until capabilities to communicate on a channel with time tag s are used only if $s\mathcal{T}t$ holds.

A tag relation $\mathcal{T}$ is extended to a relation between time tags and types, and that between time tags and type environments as follows.

Definition 4.39: Let $\mathcal{T}$ be a tag ordering. Binary relations $t\mathcal{T}\tau$ and $t\mathcal{T}\Gamma$ are defined as the least relations satisfying the following conditions.

$$
\begin{array}{l}
t\mathcal{T}\,\mathit{bool} \\
t\mathcal{T}[\tilde{\tau}]^{t'}/U \text{ if } t\mathcal{T}t' \vee U \leq 0 \\
t\mathcal{T}\Gamma \text{ if } t\mathcal{T}\Gamma(x) \text{ for each } x \in \mathit{dom}(\Gamma)
\end{array}
$$

Intuitively, $t\mathcal{T}\tau$ means that capabilities to communicate on a channel with time tag t may be used before fulfilling obligations on a value of type τ.

We can now introduce type judgments and typing rules.

Definition 4.40 [type judgments]: A type judgment is a triple $\Gamma; \mathcal{T} \vdash P$ of a type environment Γ, a tag ordering $\mathcal{T}$, and a process P.

$$\emptyset; \mathcal{T} \vdash \mathbf{0} \qquad \text{(T-Zero)}$$

$$\frac{\begin{array}{c}\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U; \mathcal{T} \vdash P \\ a \subseteq a' \qquad t\mathcal{T}(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma) \qquad ob(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma) \Rightarrow \mathbf{c} \subseteq a' \\ v_i \in \{true, false\} \Rightarrow \tau_i = bool \text{ for each } i \in \{1, \ldots, n\}\end{array}}{x : [\tau_1, \ldots, \tau_n]^t/O_{a'}.U + v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma; \mathcal{T} \vdash x!^a[v_1, \ldots, v_n].\, P} \qquad \text{(T-Out)}$$

$$\frac{\begin{array}{c}\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U, y_1 : \tau'_1, \ldots, y_n : \tau'_n; \mathcal{T} \vdash P \\ \tau_i \leq \tau'_i \text{ for each } i \in \{1, \ldots, n\} \\ a \subseteq a' \qquad t\mathcal{T}\Gamma \qquad ob(\Gamma) \Rightarrow \mathbf{c} \subseteq a'\end{array}}{\Gamma, x : [\tau_1, \ldots, \tau_n]^t/I_{a'}.U; \mathcal{T} \vdash x?^a[y_1, \ldots, y_n].\, P} \qquad \text{(T-In)}$$

$$\frac{\Gamma_1; \mathcal{T} \vdash P_1 \qquad \Gamma_2; \mathcal{T} \vdash P_2}{\Gamma_1 + \Gamma_2; \mathcal{T} \vdash P_1 \mid P_2} \qquad \text{(T-Par)}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U; \mathcal{T} \vdash P \qquad rel(U)}{\Gamma; \mathcal{T} \vdash (\nu x)\, P} \qquad \text{(T-New)}$$

$$\frac{\Gamma_1; \mathcal{T} \vdash P \qquad \Gamma_2; \mathcal{T} \vdash Q}{(\Gamma_1 \sqcap \Gamma_2) + v : bool; \mathcal{T} \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q} \qquad \text{(T-If)}$$

$$\frac{\Gamma; \mathcal{T} \vdash P}{*\Gamma; \mathcal{T} \vdash *P} \qquad \text{(T-Rep)}$$

$$\frac{\Gamma; \mathcal{T} \vdash P \qquad noob(\tau)}{\Gamma, x : \tau; \mathcal{T} \vdash P} \qquad \text{(T-Weak)}$$

Figure 1: Typing Rules

Definition 4.41 [typing rules]: The set of typing rules for deriving a type judgment are given in Figure 1. We often just write $\Gamma; \mathcal{T} \vdash P$ to mean that the type judgment $\Gamma; \mathcal{T} \vdash P$ is derivable by using these rules. We also say that $\Gamma; \mathcal{T} \vdash P$ is valid if it is derivable by using the typing rules.

Each typing rule is explained below.

(T-Zero): Because $\mathbf{0}$ uses no variables, it is well typed under the empty type environment.

(T-Out): This is is one of the key rules. The assumption $\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U; \mathcal{T} \vdash P$ implies that P uses x according to U. Because the whole process $x!^a[v_1, \ldots, v_n].\, P$ uses x for output before doing so, the total usage of x is expressed by $O_{a'}.U$. The annotation a on the output indicates that this output operation at least contains attributes (capabilities and/or obligations) in a; we therefore require the condition $a \subseteq a'$. Other variables may be used by P or by a receiver on x, possibly in parallel. The former use is expressed by Γ, while the latter use is by $v_1 : \tau_1 + \cdots + v_n : \tau_n$. Thus, the type environment of the whole process is given by $x : [\tau_1, \ldots, \tau_n]^t/O_{a'}.U + v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma$. (Here, we can assume $\{v_1, \ldots, v_n\}$ do not contain x because we do not have recursive types; see Remark 4.42 below for the changes required when we have recursive types.) We need to require additional conditions regarding capabilities and obligations. If the process P or

the tuple $[v_1, \ldots, v_n]$ being sent contains some obligations, i.e., if $ob(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma)$ holds, then the output on x must be a capability; otherwise this output process may be blocked forever without fulfilling the obligations, causing deadlock. So, we require the condition $ob(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma) \Rightarrow \mathbf{c} \subseteq a'$. The output being a capability is not a sufficient condition. Because this process fulfills the obligations possibly contained in P and $[v_1, \ldots, v_n]$ *only after* using the capability to output on x, the tag ordering $\mathcal{T}$ must explicitly allow such dependency. We therefore require the condition $t\mathcal{T}(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma)$.

(T-In): This is similar to (T-Out). The assumption $\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U, y_1 : \tau'_1, \ldots, y_n : \tau'_n; \mathcal{T} \vdash P$ implies that P uses x according to U. Because the process $x?^a[y_1, \ldots, y_n].P$ uses x and then behaves like P, the total usage of x is expressed by $I_{a'}.U$. The annotation a on the input indicates that this input operation at least contains attributes in a; so, we require the condition $a \subseteq a'$. If P contains some obligations on channels except for $y_1, \ldots, y_n$, then the input on x must be a capability; we therefore require the condition $ob(\Gamma) \Rightarrow \mathbf{c} \subseteq a'$. Because the process fulfills the obligations in Γ only after the input on x succeeds, we require the condition $t\mathcal{T}\Gamma$.

(T-Par): The assumptions imply that P_1 uses variables as described by Γ_1, and in parallel to this, P_2 uses variables as described by Γ_2. So, the type environment of $P_1 \mid P_2$ should be the combination $\Gamma_1 + \Gamma_2$. The tag ordering must be shared between P_1 and P_2, so that there is no disagreement on the order in which capabilities/obligations are used/fulfilled.

(T-New): The usage of the created channel x must be reliable, in the sense that each input/output capability is matched by the corresponding output/input obligation guaranteeing the capability. The condition $rel(U)$ is therefore required.

(T-If): Since **if** v **then** P **else** Q executes either P or Q, the uses of channels by the process is estimated by $\Gamma_1 \sqcap \Gamma_2$.

(T-Rep): Since $*P$ runs infinitely many copies of P, the type environment should also be replicated by $*$.

(T-Weak): This rule allows to add an additonal binding. It is only allowed when the type do not contain any obligations.

Remark 4.42: In the presence of recursive types, a process sending a channel through the channel itself is allowed. For example, $x!^a[x].\mathbf{0}$ is valid if x has a recursive type τ satisfying $\tau = [\tau]^t/U$. In this case, the receiver can use x only after the output on x succeeds. So, we need to generalize the rule (T-Out) as follows:

$$\frac{\begin{array}{c}\Gamma; \mathcal{T} \vdash P \\ [\tau_1, \ldots, \tau_n]^t/U = \Gamma(x) + (v_1 : \tau_1 + \cdots + v_n : \tau_n)(x) \\ a \subseteq a' \qquad t\mathcal{T}(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma)\backslash\{x\} \qquad ob((v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma)\backslash\{x\}) \Rightarrow \mathbf{c} \subseteq a' \\ v_i \in \{true, false\} \Rightarrow \tau_i = bool \text{ for each } i \in \{1, \ldots, n\}\end{array}}{x : [\tau_1, \ldots, \tau_n]^t/O_{a'}.U + (v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma)\backslash\{x\}; \mathcal{T} \vdash x!^a[v_1, \ldots, v_n].P} \quad \text{(T-Out')}$$

Remark 4.43: Originally, we included the following more general rule as (T-Weak):

$$\frac{\Gamma; \mathcal{T} \vdash P \qquad \Gamma' \leq \Gamma}{\Gamma'; \mathcal{T} \vdash P}$$

We removed this because it turned out that this rule has a bad interaction with annotations of input or output processes with obligation attributes.

Example 4.44: Let $\Gamma = x : []^{t_x}/I_{\mathbf{c}}.0, y : []^{t_y}/O_{\mathbf{o}}.0$ and $\mathcal{T} = \{(t_x, t_y)\}$. Then, $\Gamma; \mathcal{T} \vdash x?^{\emptyset}[].\, y!^{\mathbf{o}}[]$ is a valid type judgment.

Example 4.45: A type judgment

$$y : []^{t_y}/O_{\mathbf{o}}.0; \{(t_x, t_y)\} \vdash (\nu x)\,(x?^{\emptyset}[].\, y!^{\mathbf{o}}[] \mid x!^{\emptyset}[])$$

can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset; \{(t_x, t_y)\} \vdash \mathbf{0}}\,(\text{T-Zero})}{y : []^{t_y}/0, x : []^{t_x}/0; \{(t_x, t_y)\} \vdash \mathbf{0}}\,(\text{T-Weak})}{y : []^{t_y}/O_{\mathbf{o}}.0, x : []^{t_x}/0; \{(t_x, t_y)\} \vdash y!^{\mathbf{o}}[]}\,(\text{T-Out})}{y : []^{t_y}/O_{\mathbf{o}}.0, x : []^{t_x}/I_{\mathbf{c}}.0; \{(t_x, t_y)\} \vdash x?[].\, y!^{\mathbf{o}}[]}\,(\text{T-In}) \qquad \dfrac{\dfrac{\dfrac{}{\emptyset; \{(t_x, t_y)\} \vdash \mathbf{0}}\,(\text{T-Zero})}{x : []^{t_x}/0; \{(t_x, t_y)\} \vdash \mathbf{0}}\,(\text{T-Weak})}{x : []^{t_x}/O_{\mathbf{o}}.0; \{(t_x, t_y)\} \vdash x!^{\emptyset}[]}\,(\text{T-Out})}{y : []^{t_y}/O_{\mathbf{o}}.0, x : []^{t_x}/(I_{\mathbf{c}}.0 || O_{\mathbf{o}}.0); \{(t_x, t_y)\} \vdash x?^{\emptyset}[].\, y!^{\mathbf{o}}[] \mid x!^{\emptyset}[]}\,(\text{T-Par})}{y : []^{t_y}/O_{\mathbf{o}}.0; \{(t_x, t_y)\} \vdash (\nu x)\,(x?^{\emptyset}[].\, y!^{\mathbf{o}}[] \mid x!^{\emptyset}[])}\,(\text{T-New})$$

Example 4.46: A process $P = *f?[x, r].\, r![x]$ is considered to implement the identity function, since it just forwards the argument x to the reply address r. A type judgment:

$$f : [bool, [bool]^{t_r}/O_{\mathbf{o}}.0]^{t_f}/*I_{\mathbf{o}}.0; \mathcal{T} \vdash P$$

can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset; \mathcal{T} \vdash \mathbf{0}}\,(\text{T-Zero})}{f : [bool, [bool]^{t_r}/O_{\mathbf{o}}.0]^{t_f}/0, r : [bool]^{t_r}/0; \mathcal{T} \vdash \mathbf{0}}\,(\text{T-Weak})}{f : [bool, [bool]^{t_r}/O_{\mathbf{o}}.0]^{t_f}/0, x : bool, r : [bool]^{t_r}/O_{\mathbf{o}}.0; \mathcal{T} \vdash r![x]}\,(\text{T-Out})}{f : [bool, [bool]^{t_r}/O_{\mathbf{o}}.0]^{t_f}/I_{\mathbf{o}}.0; \mathcal{T} \vdash f?[x, r].\, r![x]}\,(\text{T-In})}{f : [\tau, [bool]^{t_r}/O_{\mathbf{o}}.0]^{t_f}/*I_{\mathbf{o}}.0; \mathcal{T} \vdash P}\,(\text{T-Rep})$$

where $\mathcal{T}$ is an arbitrary time tag. We know from the judgment (without looking at the process expression) that P provides infinitely many inputs on f (since the usage of f is $*I_{\mathbf{o}}.0$), and that each time P receives a pair $[x, r]$ on f, it eventually outputs a boolean value on r (since the usage of the second parameter is $O_{\mathbf{o}}.0$). From the above judgment, we can also obtain the following judgment:

$$\emptyset; \{(t_f, t_y)\} \vdash (\nu f)\,(P \mid (\nu y)\, f![true, y].\, y?^{\mathbf{co}}[z].\, \mathbf{0}).$$

The process $f![true, y]. \cdots$ calls the function located at f and waits for a reply. The judgment indicates that a reply can be eventually received (because the input on y is annotated with the capability attribute $\mathbf{c}$).

4.5 Type Soundness

We now show the correctness of our type system: if a process is well typed, any subprocess currently trying to perform communication with the capability attribute can eventually find its communication partner (unless the process is infinitely reduced). It is formally stated in Theorem 4.49.

In the proof of Theorem 4.49, the following subject reduction theorem plays an important role. It implies that the well-typedness of a process is preserved by reductions. As in the linear π-calculus [11], the type environment changes if the reduction occurs by communication on a free channel. For example, if the process $x?[].\, x?[].\, \mathbf{0} \mid x![].\, x![].\, \mathbf{0}$ is well typed under $x : []^t/(I.I.0||O.O.0)$, but the process is reduced to $x?[].\, \mathbf{0} \mid x![].\, \mathbf{0}$, which is well typed under $x : []^t/(I.0||O.0)$, not under $x : []^t/(I.I.0||O.O.0)$.

Theorem 4.47 [Subject Reduction]:

1. If $\Gamma; \mathcal{T} \vdash P$ and $P \xrightarrow{c} Q$, then $\Gamma'; \mathcal{T} \vdash Q$ and $\Gamma \leq \Gamma'$ for some Γ'.
2. If $\Gamma, x : [\tilde{\tau}]^t/U; \mathcal{T} \vdash P$ and $P \xrightarrow{x} Q$, then $\Delta; \mathcal{T} \vdash Q$, $U \longrightarrow U'$, and $(\Gamma, x : [\tilde{\tau}]^t/U') \leq \Delta$ for some Δ and U'.

Proof: See Appendix A. □

To state the deadlock-freedom theorem, we introduce the following predicate *Waiting*. Intuitively, $Waiting(P)$ means that P is trying to use a capability to input or output on some channel.

Definition 4.48: A predicate *Waiting* on processes is the least unary relation satisfying the following conditions: (i) $\mathbf{c} \subseteq a$ implies $Waiting(x!^a[\tilde{v}].\, P)$ and $Waiting(x?^a[\tilde{y}].\, P)$, and (ii) $Waiting(P)$ implies $Waiting(P \mid Q)$, $Waiting(Q \mid P)$, $Waiting(*P)$, and $Waiting((\nu x)\, P)$.

Theorem 4.49 [Deadlock Freedom]: If $\emptyset; \mathcal{T} \vdash P$ and $Waiting(P)$, then there exists Q such that $P \longrightarrow Q$.

Proof: The proof proceeds in the same way as the proof of the deadlock-freedom property of the previous type system [10]. Basically, we can find a sub-process waiting on a channel with a minimal time tag among those trying to use an input or output capability and show that there must exists a process fulfilling the corresponding output or input obligation on the same channel. Because we use Theorem 5.1 in the next section, we defer a complete proof until Appendix B. □

The above theorem states only about closed process expressions. As for a process containing free variables, even if it is well typed, an input/output process annotated with $\mathbf{c}$ may not be reduced by itself. For example, $x : []^t/I_{\mathbf{c}}.0; \emptyset \vdash x?^{\mathbf{c}}[].\, \mathbf{0}$ is a valid type judgment, but the process cannot be reduced by itself. This is just because the input on x being a capability depends on the assumption that some external process fulfills an obligation to perform an output on x. So, if the above process is correctly composed with external processes so that the whole process is closed and well typed, the input on x will eventually succeed.

We omit statements about the properties guaranteed by the obligation annotation ($!^{\mathbf{o}}$ and $?^{\mathbf{o}}$).

5 Type Reconstruction

We now turn to the main goal of this paper: type reconstruction. The type system has been reformalized in the previous section for this goal, but there still exists a little hurdle to develop a type reconstruction

algorithm: The typing rules in Figure 1 are not syntax-directed (i.e., there are more than one applicable rules for each process expression). We make the typing rules syntax-directed by eliminating the rule (T-Weak) in Section 5.1. Then, we define the notion of a principal typing in Section 5.2. As usual, the principal typing of a process expresses all the possible typings of the process. It will be defined as a pair of a type environment containing variables and a set of constraints on the variables. After that, we give an algorithm to obtain a principal typing. The typability of a process is decided by checking the satisfiability of the constraint set. It is deferred until Section 6.

5.1 Syntax-directed typing rules

We can eliminate applications of the rule (T-Weak) from a type derivation (except for those at the bottom of the derivation) by moving applications of (T-Weak) downwards as far as possible, and then combining the remaining applications of (T-Weak) with applications of other rules.

For example, consider the following type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\mathcal{T}\vdash\mathbf{0}}\ \text{(T-Zero)}}{x:[]^t/0;\mathcal{T}\vdash\mathbf{0}}\ \text{(T-Weak)}}{x:[]^t/0,y:\mathit{bool};\mathcal{T}\vdash\mathbf{0}}\ \text{(T-Weak)}}{x:[]^t/I_a.0,y:\mathit{bool};\mathcal{T}\vdash x?[].\mathbf{0}}\ \text{(T-In)}$$

The second application of (T-Weak) is unnecessary for the application of (T-In). So, we can move it downwards and obtain:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset;\mathcal{T}\vdash\mathbf{0}}\ \text{(T-Zero)}}{x:[]^t/0;\mathcal{T}\vdash\mathbf{0}}\ \text{(T-Weak)}}{x:[]^t/I_a.0;\mathcal{T}\vdash x?[].\mathbf{0}}\ \text{(T-In)}}{x:[]^t/I_a.0,y:\mathit{bool};\mathcal{T}\vdash x?[].\mathbf{0}}\ \text{(T-Weak)}$$

The first application of (T-Weak) is necessary for the application of (T-In), but we can eliminate it by generalizing (T-In) to:

$$\dfrac{\begin{array}{c}\Gamma;\mathcal{T}\vdash_{\mathcal{STR}}P\\ \tau_i\le\Gamma(y_i)\text{ for each }i\text{ such that }y_i\in\mathit{dom}(\Gamma)\\ \mathit{noob}(\tau_i)\text{ for each }i\text{ such that }y_i\notin\mathit{dom}(\Gamma)\\ \Gamma(x)=[\tau_1,\ldots,\tau_n]^t/U\text{ or }x\notin\mathit{dom}(\Gamma)\wedge U=0\\ a\subseteq a'\qquad t\mathcal{T}\Gamma\backslash\{x,y_1,\ldots,y_n\}\qquad \mathit{ob}(\Gamma\backslash\{x,y_1,\ldots,y_n\})\Rightarrow\mathbf{c}\subseteq a'\end{array}}{\Gamma\backslash\{x,y_1,\ldots,y_n\},x:[\tau_1,\ldots,\tau_n]^t/I_{a'}.U;\mathcal{T}\vdash_{\mathcal{STR}}x?^a[y_1,\ldots,y_n].P}\qquad\text{(ST-In)}$$

This rule allows the type environment of the body P not to contain a type binding on x. By using the new rule, we can replace the derivation with:

$$\dfrac{\dfrac{\dfrac{}{\emptyset;\mathcal{T}\vdash\mathbf{0}}\ \text{(T-Zero)}}{x:[]^t/I_a.0;\mathcal{T}\vdash x?[].\mathbf{0}}\ \text{(T-In')}}{x:[]^t/I_a.0,y:\mathit{bool};\mathcal{T}\vdash x?[].\mathbf{0}}\ \text{(T-Weak)}$$

Thus, the applications of (T-Weak) except for the last one have been removed.

$$\emptyset; \mathcal{T} \vdash_{\mathcal{STR}} \mathbf{0} \qquad \text{(ST-Zero)}$$

$$\frac{\begin{array}{c}\Gamma; \mathcal{T} \vdash P \\ \Gamma(x) = [\tau_1, \ldots, \tau_n]^t/U \vee (x \notin dom(\Gamma) \wedge U = 0) \\ a \subseteq a' \qquad t\mathcal{T}(v_1 : \tau_1 + \cdots + v_n : \tau_n + (\Gamma \backslash \{x\})) \qquad ob(v_1 : \tau_1 + \cdots + v_n : \tau_n + (\Gamma \backslash \{x\})) \Rightarrow \mathbf{c} \subseteq a' \\ v_i \in \{true, false\} \Rightarrow \tau_i = bool \text{ for each } i \in \{1, \ldots, n\}\end{array}}{x : [\tau_1, \ldots, \tau_n]^t/O_{a'}.U + v_1 : \tau_1 + \cdots + v_n : \tau_n + (\Gamma \backslash \{x\}); \mathcal{T} \vdash x!^a[v_1, \ldots, v_n].\, P} \qquad \text{(ST-Out)}$$

$$\frac{\begin{array}{c}\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} P \\ \tau_i \leq \Gamma(y_i) \text{ for each } i \text{ such that } y_i \in dom(\Gamma) \\ noob(\tau_i) \text{ for each } i \text{ such that } y_i \notin dom(\Gamma) \\ \Gamma(x) = [\tau_1, \ldots, \tau_n]^t/U \text{ or } x \notin dom(\Gamma) \wedge U = 0 \\ a \subseteq a' \qquad t\mathcal{T}\Gamma \backslash \{x, y_1, \ldots, y_n\} \qquad ob(\Gamma \backslash \{x, y_1, \ldots, y_n\}) \Rightarrow \mathbf{c} \subseteq a'\end{array}}{\Gamma \backslash \{x, y_1, \ldots, y_n\}, x : [\tau_1, \ldots, \tau_n]^t/I_{a'}.U; \mathcal{T} \vdash_{\mathcal{STR}} x?^a[y_1, \ldots, y_n].\, P} \qquad \text{(ST-In)}$$

$$\frac{\Gamma_1; \mathcal{T} \vdash_{\mathcal{STR}} P_1 \qquad \Gamma_2; \mathcal{T} \vdash_{\mathcal{STR}} P_2}{\Gamma_1 + \Gamma_2; \mathcal{T} \vdash_{\mathcal{STR}} P_1 \mid P_2} \qquad \text{(ST-Par)}$$

$$\frac{\begin{array}{c}\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} P \\ (\Gamma(x) = [\tau_1, \ldots, \tau_n]^t/U \wedge rel(U)) \vee x \notin dom(\Gamma)\end{array}}{\Gamma \backslash \{x\}; \mathcal{T} \vdash_{\mathcal{STR}} (\nu x)\, P} \qquad \text{(ST-New)}$$

$$\frac{\Gamma_1; \mathcal{T} \vdash_{\mathcal{STR}} P_1 \qquad \Gamma_2; \mathcal{T} \vdash_{\mathcal{STR}} P_2}{(\Gamma_1 \sqcap \Gamma_2) + v : bool; \mathcal{T} \vdash_{\mathcal{STR}} \mathbf{if}\ v\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2} \qquad \text{(ST-If)}$$

$$\frac{\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} P}{*\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} *P} \qquad \text{(ST-Rep)}$$

Figure 2: Syntax-Directed Typing Rules

Based on the above ideas, we can reformalize the typing rules as shown in Figure 2. By using those rules and using (T-Weak) only at the bottom of derivations, we can derive the same type judgments as those obtained by the rules in Figure 1. We write $\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} P$ if $\Gamma; \mathcal{T} \vdash S$ is derivable by the rules in Figure 2.

Each rule (ST-xx) can be considered a combination of (T-xx) with minimal weakening required to make (T-xx) applicable. The rule (ST-Out) can be considered a combination of (T-Out) with weakening on x. (ST-In) is a combination of (T-In) with weakening on $x, y_1, \ldots, y_n$. The rule (ST-New) is a combination of (T-New) with weakening on x. The rule (ST-If) can be considered a combination of (T-If) with minimal weakening required to make the type environments of P_1 and P_2 coincide.

The syntax-directed rules are essentially equivalent to the typing rules in Section 4 in the following sense.

Theorem 5.1 [Correctness of Syntax-Directed Rules]:

1. If $\Gamma; \mathcal{T} \vdash P$ holds, then there exists Γ' such that $\Gamma'; \mathcal{T} \vdash_{\mathcal{STR}} P$ and $\Gamma = \Gamma', x_1 : \tau_1, \ldots, x_n : \tau_n$.
2. If $\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} P$ holds, then $\Gamma; \mathcal{T} \vdash P$ also holds.

Proof: Straightfoward induction on type derivation. □

5.2 Principal Typing

Since the typing rules in Section 5.1 are syntax-directed (i.e., there is only one rule that matches each process expression), by introducing variables expressing types and usage attributes, we can express all the possible typings as a pair of a type judgment containing those variables and a set of constraints on the variables. We call the pair a principal typing. The concrete definition of a principal typing follows the definitions of extended type judgments and constraints.

Definition 5.2 [extended usage attributes, usages, and types]: The sets of extended usage attributes, extended usages, and extended types are given by the following syntax.

$$
\begin{array}{l}
a ::= \zeta \mid \emptyset \mid \mathbf{c} \mid \mathbf{o} \mid \mathbf{co} \\
U ::= \mathit{Uof}(\tau) \mid 0 \mid \alpha \mid O_a.U \mid I_a.U \mid (U_1||U_2) \mid U_1 \sqcap U_2 \mid \mathbf{rec}\,\alpha.U \mid *U \\
\tau ::= \rho \mid \mathit{bool} \mid [\tau_1, \ldots, \tau_n]^t/U \mid \tau_1 + \tau_2 \mid \tau_1 \sqcap \tau_2 \mid *\tau
\end{array}
$$

Here, ζ and ρ denote variables ranging over attributes and types respectively. Actually, t above is also a variable ranging over time tags, but in order to avoid introducing so many meta-variables, we do not distinguish between variables ranging over time tags and time tags.

$\mathit{Uof}(\tau)$ is an expression representing the outermost usage of τ if τ is a channel type. We identify an expression $\mathit{Uof}([\tilde{\tau}]^t/U)$ with U. An extended type judgment is obtained by replacing types in a type environment with extended types. Operations $+, \sqcap, *$ on type environments are naturally extended to those on extended type environments. For example, $(x : \rho) + (y : \mathit{bool}, x : [\tilde{\tau}]^t/U)$ is defined as $x : (\rho + [\tilde{\tau}]^t/U), y : \mathit{bool}$.

We do not distinguish between two extended types which are instantiated to the same type for any substitution: for example, we identify $[\tilde{\tau}]^t/U_1 + [\tilde{\tau}]^t/U_2$ and $[\tilde{\tau}]^t/(U_1||U_2)$. In the rest of this section, we use metavariables a, U, τ, and Γ for extended attributes, extended usages, extended types, and extended type environments.

Definition 5.3 [constraints]: The set of constraints, ranged over by c, is defined by:

$$
\begin{array}{l}
c ::= \mathbf{false} \mid c_\tau \mid c_U \mid c_a \mid c_t \\
c_\tau \text{ (constraints on types)} ::= \tau_1 \approx \tau_2 \mid \tau_1 \sim \tau_2 \mid \tau_1 \leq \tau_2 \mid \mathit{noob}(\tau) \\
\qquad\qquad\qquad\qquad \mid \mathit{rel}(\tau) \mid (o_1 \vee \cdots \vee o_n) \Rightarrow c_a \mid t \mathcal{T} \tau \\
o \text{ (obligation predicates)} ::= \mathit{ob}(\tau) \mid \mathit{ob}(U) \\
c_U \text{ (constraints on usages)} ::= U_1 \leq U_2 \mid \mathit{noob}(U) \mid (\mathit{ob}(U_1) \vee \cdots \vee \mathit{ob}(U_n)) \Rightarrow c_a \mid \mathit{ob}(U) \Rightarrow c_t \mid \mathit{rel}(U) \\
c_a \text{ (constraints on attributes)} ::= a_1 \leq a_2 \mid a_1 \subseteq a_2 \\
c_t \text{ (constraints on time tags)} ::= t_1 \mathcal{T} t_2
\end{array}
$$

We write C for a set of constraints.

$\tau_1 \approx \tau_2$ means that τ_1 and τ_2 must be identical. $\tau_1 \sim \tau_2$ means that τ_1 and τ_2 must be identical except for the outermost usages (so, when variables in τ_1 and τ_2 must be instantiated, $\tau_1 + \tau_2$ and $\tau_1 \sqcap \tau_2$ are well defined). $\mathit{rel}(\tau)$ means that τ is a channel type $[\tau_1, \ldots, \tau_n]^t/U$ and $\mathit{rel}(U)$ holds.

Notation 5.4: We write θ for a substitution of types, usages, and attributes for type, usage, and attribute variables. We write $FV(\Gamma)$ and $FV(C)$ for the sets of variables appearing free in Γ and C respectively.

Definition 5.5 [principal typing]: A pair (Γ, C) of an extended type environment and a set of constraints is a *principal typing* of a process P if it satisfies the following conditions:

1. If θ and a tag relation $\mathcal{T}$ is chosen so that $dom(\theta) \supseteq FV(\Gamma) \cup FV(C)$ and θC is satisfied, then $\theta\Gamma; \mathcal{T} \vdash P$.
2. If $\Gamma'; \mathcal{T} \vdash P$, then there exists a substitution θ such that θC and $\Gamma' \leq \theta\Gamma$ hold.

5.3 Algorithm for Computing a Principal Typing

By reading syntax-directed typing rules in a bottom-up manner, we can easily construct an algorithm for computing a principal typing. It is shown in Figure 3. In the figure, $Rep(\tau)$, $ob(\Gamma)$, $t\mathcal{T}\Gamma$, and $\Gamma_1 \sim \cdots \sim \Gamma_n$ are defined by:

$$
\begin{array}{l}
Rep(\rho) = \rho \\
Rep(bool) = bool \\
Rep([\tilde{\tau}]^t/U) = [\tilde{\tau}]^t/0 \\
Rep(\tau_1 + \tau_2) = Rep(\tau_1) \\
Rep(\tau_1 \sqcap \tau_2) = Rep(\tau_1) \\
Rep(*(\tau)) = Rep(\tau)
\end{array}
$$

$$
\begin{array}{l}
ob(\emptyset) = \mathbf{false} \\
ob(x_1 : \tau_1, \ldots, x_n : \tau_n) = ob(\tau_1) \vee \cdots \vee ob(\tau_n)
\end{array}
$$

$$
t\mathcal{T}(x_1 : \tau_1, \ldots, x_n : \tau_n) = \{t\mathcal{T}\tau_1, \ldots, t\mathcal{T}\tau_n\}
$$

$$
\begin{array}{l}
\Gamma_1 \sim \cdots \sim \Gamma_n = \\
\quad \{Rep(\tau_1) \sim Rep(\tau_2) \mid \Gamma_i(x) = \tau_1 \text{ and } \Gamma_j(x) = \tau_2 \text{ for some } x, i, j \text{ such that } 1 \leq i < j \leq n\}
\end{array}
$$

Notice that the above definition does not depend on the particular representation of extended types: Although we identify $[\tilde{\tau}]^t/U_1 + [\tilde{\tau}]^t/U_2$ with $[\tilde{\tau}]^t/(U_1||U_2)$, *Rep* returns the same type for both representations.

It is trivial that PT outputs a principal typing.

Theorem 5.6: Let P be a process expression. Then $PT(P)$ is a principal typing of P.

Proof: The first condition follows by fairly straightforward induction on the structure of P. The second condition follows by Theorem 5.1 and induction on derivation of $\Gamma; \mathcal{T} \vdash_{\mathcal{STR}} P$. □

Note that PT outputs a principal typing even for an ill-typed process. Since the constraint set of the output principal typing is unsatisfiable in that case, the conditions for principal typings are vacuously true. For example, for an ill-typed process $P = x?^a[].(\mathbf{if}\ x\ \mathbf{then\ 0\ else\ 0}\ )$, $PT(P)$ is $(x : \cdots, \{[]^t/\alpha \sim bool, \ldots\})$, whose constraint part is unsatisfiable. This kind of process is rejected in the constraint solving phase described in Section 6. In practice, it would be better to interleave the phase of computing $PT(P)$ and the constraint solving phase in order to reject ill-typed processes as early as possible.

Note that not all the constraints introduced in Definition 5.3 can be solved. For example, we don't have an effective way to solve a type equality constraint $\rho_1 + \rho_2 \approx \rho_3 + \rho_4$. Fortunately, however, we know that PT outputs only a certain restricted form of constraint set, so that we can effectively solve it. In fact, the output of PT satisfies the conditions below. We say C implies c when θc holds for every substitution θ such that $dom(\theta) \supseteq FV(C)$ and θC holds. We write $well\_defined(\tau)$ when τ is well defined.

$$
\begin{array}{l}
PT(\mathbf{0}) = (\emptyset, \emptyset) \\
PT(x!^a[v_1, \ldots, v_n].\, P) = \\
\quad \textbf{let } (\Gamma, C_1) = PT(P) \\
\qquad (C_2, U) = \textbf{if } x \in dom(\Gamma) \textbf{ then } (\{[\rho_1, \ldots, \rho_n]^t/\alpha \sim Rep(\Gamma(x))\},\, Uof(\Gamma(x))) \\
\qquad\qquad \textbf{else } (\emptyset, 0) \\
\quad \textbf{in } (x : [\rho_1, \ldots, \rho_n]^t/O_\zeta.U + v_1 : \rho_1 + \cdots + v_n : \rho_n + (\Gamma \backslash \{x\}), \\
\qquad C_1 \cup C_2 \cup \{a \subseteq \zeta, ob(v_1 : \rho_1 + \cdots + v_n : \rho_n + (\Gamma \backslash \{x\})) \Rightarrow \mathbf{c} \subseteq \zeta\} \\
\qquad \cup (x : [\rho_1, \ldots, \rho_n]^t/O_\zeta.U \sim v_1 : \rho_1 \sim \cdots \sim v_n : \rho_n \sim (\Gamma \backslash \{x\})) \\
\qquad \cup (t \mathcal{T} v_1 : \rho_1 + \cdots + v_n : \rho_n + (\Gamma \backslash \{x\}) \\
\qquad \cup \{\rho_i \approx bool \mid i \in \{1, \ldots, n\}, v_i \in \{true, false\}\}) \\
\quad (\text{where } \rho_1, \ldots, \rho_n, \alpha, \zeta, t \text{ are fresh}) \\
PT(x?^a[y_1, \ldots, y_n].\, P) = \\
\quad \textbf{let } (\Gamma, C_1) = PT(P) \\
\qquad (C_2, U) = \textbf{if } x \in dom(\Gamma) \textbf{ then } (\{[\rho_1, \ldots, \rho_n]^t/\alpha \sim Rep(\Gamma(x))\},\, Uof(\Gamma(x))) \\
\qquad\qquad \textbf{else } (\emptyset, 0) \\
\quad \textbf{in } ((\Gamma \backslash \{x, y_1, \ldots, y_n\}, x : [\rho_1, \ldots, \rho_n]^t/I_\zeta.U), \\
\qquad C_1 \cup C_2 \cup \{a \subseteq \zeta, ob(\Gamma \backslash \{x, y_1, \ldots, y_n\}) \Rightarrow \mathbf{c} \subseteq \zeta\} \\
\qquad \cup \{\rho_i \leq \Gamma(y_i), \rho_i \sim Rep(\Gamma(y_i)) \mid y_i \in dom(\Gamma)\} \cup \{noob(\rho_i) \mid y_i \notin dom(\Gamma)\} \\
\qquad \cup t \mathcal{T} \Gamma \backslash \{x, y_1, \ldots, y_n\} \\
\quad (\text{where } \rho_1, \ldots, \rho_n, \alpha, \zeta, t'' \text{ are fresh}) \\
PT(P_1 \mid P_2) = \\
\quad \textbf{let } (\Gamma_1, C_1) = PT(P_1) \\
\qquad (\Gamma_2, C_2) = PT(P_2) \\
\quad \textbf{in } (\Gamma_1 + \Gamma_2, C_1 \cup C_2 \cup (\Gamma_1 \sim \Gamma_2)) \\
PT((\nu x)\, P) = \\
\quad \textbf{let } (\Gamma, C) = PT(P) \\
\quad \textbf{in if } x \in dom(\Gamma) \textbf{ then } (\Gamma \backslash \{x\}, C \cup \{rel(\Gamma(x))\}) \textbf{ else } (\Gamma, C) \\
PT(\textbf{if } v \textbf{ then } P_1 \textbf{ else } P_2\ ) = \\
\quad \textbf{let } (\Gamma_1, C_1) = PT(P_1) \\
\qquad (\Gamma_2, C_2) = PT(P_2) \\
\quad \textbf{in } ((\Gamma_1 \sqcap \Gamma_2) + v : bool, C_1 \cup C_2 \cup (\Gamma_1 \sim \Gamma_2 \sim v : bool)) \\
PT(*P) = \\
\quad \textbf{let } (\Gamma, C) = PT(P) \\
\quad \textbf{in } (*\Gamma, C)
\end{array}
$$

Figure 3: Algorithm for computing a principal typing

Lemma 5.7: If $(\Gamma, C) = PT(P)$, then

1. For any type expression of the form $[\tau_1, \ldots, \tau_n]^t/U$ appearing in Γ and C, all the extended usages in $\tau_1, \ldots, \tau_n$ are usage variables.

2. For any constraint $\tau_1 \approx \tau_2 \in C$, all the extended usages in τ_1 and τ_2 are usage variables.

3. For any constraint $\tau_1 \leq \tau_2 \in C$, all the extended usages in τ_1 are usage variables. Moreover, τ_1 contains no operators $+$, $\sqcap$, and $*$ on types. Similarly, for any constraint $noob(\tau) \in C$, all the extended usages in τ are usage variables and τ contains no operators $+$, $\sqcap$, and $*$ on types.

4. For any constraint $\tau_1 \approx \tau_2$ or $\tau_1 \sim \tau_2$ contained in C, τ_1 and τ_2 contain no operators $+$, $\sqcap$, and $*$ on types.

5. Let $C' = \{c \in C \mid c \text{ is of the form } \tau_1 \approx \tau_2 \text{ or } \tau_1 \sim \tau_2\}$. C' implies (i) $well\_defined(\tau)$ for every type τ appearing in Γ and C, (ii) $\tau \sim \tau'$ for any $\tau_1 \leq \tau_2 \in C$, and (iii) $\tau \sim \tau'$ for some channel type τ' for any usage expression of the form $Uof(\tau)$ appearing in in C.

Proof: 1. Trivial by the fact that all the channel types constructed in PT are of the form $[\rho_1, \ldots, \rho_n]^t/U$, and the fact that the operations $+, \sqcap$, and $*$ on type environments only compose existing type expressions with $+, \sqcap$, and $*$.

2. Constraints of the form $\tau_1 \approx \tau_2$ are only introduced in the case for output expressions, and they are of the form $\rho \approx bool$.

3. A constraint of the form $\tau_1 \leq \tau_2$ or $noob(\tau_1)$ is only introduced in the case for input processes, and τ_1 in this case is a fresh type variable.

4. PT introduces no constraint of the form $\tau_1 \approx \tau_2$, except for those of the form $\rho \approx bool$. Constraints of the form $\tau_1 \sim \tau_2$ can be introduced only by either the function $\sim$ on type environments or by $[\rho_1, \ldots, \rho_n]^t/\alpha \sim Rep(\Gamma(x))$ or $\rho_i \sim Rep(\Gamma(y_i))$ in the cases for output or input processes. In all the cases, the produced constraints $\tau_1 \sim \tau_2$ cannot contain operators $+, \sqcap, *$, by the definition of $Rep(\tau)$ and $\Gamma_1 \sim \Gamma_2$.

5. This follows from the fact that whenever a new expression is constructed by applying $+, \sqcap, *, Uof$ in PT, appropriate constraints of the form $\tau_1 \sim \tau_2$ are added to the constraint set.

□

Example 5.8: Let $P = (\nu x)\,(\nu y)\,(x![y] \mid x?^{\mathbf{c}}[z].\, z?[w].\, \mathbf{0})$. Then, $PT(P)$ is computed as follows (constraints

of the form $\mathbf{false} \Rightarrow c_a$ are omitted).

$$
\begin{array}{l}
PT(x![y]) = ((x:[\rho_y]^{t_x}/O_{\zeta_x}.0, y:\rho_y), \{\emptyset \subseteq \zeta_x, ob(\rho_y) \Rightarrow \mathbf{c} \subseteq \zeta_x, t_x \mathcal{T} \rho_y\}).\\
PT(z?[w].\,\mathbf{0}) = (z:[\rho_w]^{t_z}/I_{\zeta_z}.0, \{\emptyset \subseteq \zeta_z, noob(\rho_w)\})\\
PT(x?^{\mathbf{c}}[z].\,z?[w].\,\mathbf{0}) =\\
\quad (x:[\rho'_z]^{t'_x}/I_{\zeta'_x}.0, \{\emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0\})\\
PT(x![y]\mathbf{0} \mid x?^{\mathbf{c}}[z].\,z?[w].\,\mathbf{0}) =\\
\quad ((x:([\rho_y]^{t_x}/O_{\zeta_x}.0 + [\rho'_z]^{t'_x}/I_{\zeta'_x}.0), y:\rho_y),\\
\quad \{\emptyset \subseteq \zeta_x, ob(\rho_y) \Rightarrow \mathbf{c} \subseteq \zeta_x, t_x \mathcal{T} \rho_y,\\
\quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0,\\
\quad [\rho_y]^{t_x}/0 \sim [\rho'_z]^{t'_x}/0\})\\
PT(P) = (\emptyset,\\
\quad \{\emptyset \subseteq \zeta_x, ob(\rho_y) \Rightarrow \mathbf{c} \subseteq \zeta_x, t_x \mathcal{T} \rho_y,\\
\quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0,\\
\quad [\rho_y]^{t_x}/0 \sim [\rho'_z]^{t'_x}/0, rel([\rho_y]^{t_x}/O_{\zeta_x}.0 + [\rho'_z]^{t'_x}/I_{\zeta'_x}.0), rel(\rho_y)\})
\end{array}
$$

6 Deciding Typability by Constraint Solving

The existence of a principal typing does not imply that there is a valid type judgment, because the set of constraints may not be satisfiable. Indeed, the algorithm described in Section 5.3 always outputs a principal typing, even for an ill-typed process $x?^a[y_1, y_2, y_3].\, x!^a[y_1, y_2].\, \mathbf{0}$. The typability of a process is decided by reducing the set of constraints in its principal typing and checking its satisfiability.

We describe below how to reduce the set of constraints step by step. We reduce constraints on types, those on usages, those on attributes and those on time tags in this order. The algorithm for reducing constraints on usages is incomplete. As mentioned in Section 1, this is just because we add some extra constraints when reducing usage constraints in order to reject some well-typed but bad processes.

The typability can be completely decided only when the whole process is given or type information on the external processes is given. For example, whether or not a process $(\nu y)\,(x!^{\mathbf{c}}[y].\,\mathbf{0} \mid y?^{\mathbf{c}}[\,].\,\mathbf{0})$ is well-typed depends on the behavior of input processes on the channel x. Therefore, when we need to incrementally check the typability of processes, we can only partially apply the transformation rules given below. Although some processes may be found to be ill-typed and rejected during partial reduction of constraints, the complete decision of typability of some processes must be deferred until the whole process is given or type information on all the free variables is given. For simplicity, we assume below that the input process P is closed.

6.1 Reducing Constraints on Types

We reduce constraints on types in two steps: we first reduce constraints of the form $\tau_1 \approx \tau_2$ or $\tau_1 \sim \tau_2$, and then reduce constraints of the other forms.

We can transform constraints of the form $\tau_1 \approx \tau_2$ or $\tau_1 \sim \tau_2$ by using the transformation rules in Figure 4.

Example 6.1: Let $P = (\nu x)\,(\nu y)\,(x![y] \mid x?^{\mathbf{c}}[z].\, z?[w].\, \mathbf{0})$. As given in Example 5.8, $PT(P) = (\emptyset, C)$ for

$$
\begin{array}{l}
C = \{\emptyset \subseteq \zeta_x, ob(\rho_y) \Rightarrow \mathbf{c} \subseteq \zeta_x, t_x \mathcal{T} \rho_y,\\
\quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0,\\
\quad [\rho_y]^{t_x}/0 \sim [\rho'_z]^{t'_x}/0, rel([\rho_y]^{t_x}/O_{\zeta_x}.0 + [\rho'_z]^{t'_x}/I_{\zeta'_x}.0), rel(\rho_y)\}
\end{array}
$$

$$(\Gamma, C \cup \{\rho \approx \tau\}) \rightsquigarrow \begin{cases} (\Gamma, \{\mathbf{false}\}) & \text{if } \rho \neq \tau \text{ and } \rho \in FV(\tau) \\ [\rho \mapsto \tau](\Gamma, C) & \text{otherwise} \end{cases}$$

$$(\Gamma, C \cup \{\tau \approx \rho\}) \rightsquigarrow \begin{cases} (\Gamma, \{\mathbf{false}\}) & \text{if } \rho \neq \tau \text{ and } \rho \in FV(\tau) \\ [\rho \mapsto \tau](\Gamma, C) & \text{otherwise} \end{cases}$$

$$(\Gamma, C \cup \{\mathit{bool} \approx \mathit{bool}\}) \rightsquigarrow (\Gamma, C)$$

$$(\Gamma, C \cup \{\mathit{bool} \approx [\tau_1, \ldots, \tau_n]^t/U\}) \rightsquigarrow (\Gamma, \{\mathbf{false}\})$$

$$(\Gamma, C \cup \{[\tau_1, \ldots, \tau_n]^t/\alpha \approx [\tau'_1, \ldots, \tau'_n]^{t'}/\alpha'\}) \rightsquigarrow [\alpha \mapsto \alpha', t \mapsto t'](\Gamma, C \cup \{\tau_1 \approx \tau'_1, \ldots, \tau_n \approx \tau'_n\})$$

$$(\Gamma, C \cup \{[\tau_1, \ldots, \tau_n]^t/\alpha \approx \mathit{bool}) \rightsquigarrow (\Gamma, \{\mathbf{false}\})$$

$$(\Gamma, C \cup \{\rho \sim \mathit{bool}\}) \rightsquigarrow [\rho \mapsto \mathit{bool}](\Gamma, C)$$

$$(\Gamma, C \cup \{\rho \sim [\tau_1, \ldots, \tau_n]^t/U\}) \rightsquigarrow \begin{cases} (\Gamma, \{\mathbf{false}\}) & \text{if } \rho \in FV(\tau_1) \cup \cdots \cup FV(\tau_n) \\ [\rho \mapsto [\tau_1, \ldots, \tau_n]^t/\alpha](\Gamma, C) & \text{otherwise } (\alpha \text{ fresh}) \end{cases}$$

$$(\Gamma, C \cup \{\mathit{bool} \sim \rho\}) \rightsquigarrow [\rho \mapsto \mathit{bool}](\Gamma, C)$$

$$(\Gamma, C \cup \{[\tau_1, \ldots, \tau_n]^t/U \sim \rho\}) \rightsquigarrow \begin{cases} (\Gamma, \{\mathbf{false}\}) & \text{if } \rho \in FV(\tau_1) \cup \cdots \cup FV(\tau_n) \\ [\rho \mapsto [\tau_1, \ldots, \tau_n]^t/\alpha](\Gamma, C) & \text{otherwise } (\alpha \text{ fresh}) \end{cases}$$

$$(\Gamma, C \cup \{\mathit{bool} \sim \mathit{bool}\}) \rightsquigarrow (\Gamma, C)$$

$$(\Gamma, C \cup \{\mathit{bool} \sim [\tau_1, \ldots, \tau_n]^t/U\}) \rightsquigarrow (\Gamma, \{\mathbf{false}\})$$

$$(\Gamma, C \cup \{[\tau_1, \ldots, \tau_n]^t/U \sim [\tau'_1, \ldots, \tau'_n]^{t'}/U'\}) \rightsquigarrow [t \mapsto t'](\Gamma, C \cup \{\tau_1 \approx \tau'_1, \ldots, \tau_n \approx \tau'_n\})$$

$$(\Gamma, C \cup \{[\tau_1, \ldots, \tau_n]^t/U \sim \tau) \rightsquigarrow (\Gamma, \{\mathbf{false}\})$$

if τ is *bool* or a channel type of the form $[\tau'_1, \ldots, \tau'_m]^{t'}/\alpha'$ where $m \neq n$

Figure 4: Rules for reducing type constraints (1)

$$
\begin{array}{l}
(\Gamma, C \cup \{\rho \leq \rho\}) \rightsquigarrow (\Gamma, C) \\
(\Gamma, C \cup \{bool \leq bool\}) \rightsquigarrow (\Gamma, C) \\
(\Gamma, C \cup \{[\tau_1, \ldots, \tau_n]^t/U \leq [\tau_1, \ldots, \tau_n]^t/U'\}) \rightsquigarrow (\Gamma, C \cup \{U \leq U'\}) \\
\\
(\Gamma, C \cup \{noob(bool)\}) \rightsquigarrow (\Gamma, C) \\
(\Gamma, C \cup \{noob([\tau_1, \ldots, \tau_n]^t/U)\}) \rightsquigarrow (\Gamma, C \cup \{noob(U)\}) \\
\\
(\Gamma, C \cup \{rel(bool)\}) \rightsquigarrow (\Gamma, \{\mathbf{false}\}) \\
(\Gamma, C \cup \{rel([\tau_1, \ldots, \tau_n]^t/U)\}) \rightsquigarrow (\Gamma, C \cup \{rel(U)\}) \\
\\
(\Gamma, C \cup \{(o_1 \vee \cdots \vee o_{i-1} \vee ob(bool) \vee o_{i+1} \vee \cdots \vee o_n) \Rightarrow c_a\}) \\
\qquad \rightsquigarrow (\Gamma, C \cup \{(o_1 \vee \cdots \vee o_{i-1} \vee o_{i+1} \vee \cdots \vee o_n) \Rightarrow c_a\}) \\
(\Gamma, C \cup \{(o_1 \vee \cdots \vee o_{i-1} \vee ob([\tilde{\tau}]^t/U) \vee o_{i+1} \vee \cdots \vee o_n) \Rightarrow c_a\}) \\
\qquad \rightsquigarrow (\Gamma, C \cup \{(o_1 \vee \cdots \vee o_{i-1} \vee ob(U) \vee o_{i+1} \vee \cdots \vee o_n) \Rightarrow c_a\}) \\
\\
(\Gamma, C \cup \{t \mathcal{T} bool\}) \rightsquigarrow (\Gamma, C) \\
(\Gamma, C \cup \{t \mathcal{T} [\tilde{\tau}]^{t'}/U\}) \rightsquigarrow (\Gamma, C \cup \{ob(U) \Rightarrow t \mathcal{T} t'\})
\end{array}
$$

Figure 5: Rules for reducing type constraints (2)

It is transformed by the rules in Figure 4 as follows.

$$
\begin{array}{rcl}
(\emptyset, C) & \rightsquigarrow & [t_x \mapsto t'_x](\emptyset, \{\emptyset \subseteq \zeta_x, ob(\rho_y) \Rightarrow \mathbf{c} \subseteq \zeta_x, t_x \mathcal{T} \rho_y, \\
& & \qquad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0, \\
& & \qquad rel([\rho_y]^{t_x}/O_{\zeta_x}.0 + [\rho'_z]^{t'_x}/I_{\zeta'_x}.0), rel(\rho_y), \rho_y \approx \rho'_z\}) \\
& & \qquad\qquad \text{(rewriting on } [\rho_y]^{t_x}/0 \sim [\rho'_z]^{t'_x}/0) \\
& \rightsquigarrow & [\rho_y \mapsto \rho'_z][t_x \mapsto t'_x](\emptyset, \{\emptyset \subseteq \zeta_x, ob(\rho_y) \Rightarrow \mathbf{c} \subseteq \zeta_x, t_x \mathcal{T} \rho_y, \\
& & \qquad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0, \\
& & \qquad rel([\rho_y]^{t_x}/O_{\zeta_x}.0 + [\rho'_z]^{t'_x}/I_{\zeta'_x}.0), rel(\rho_y)\}) \\
& & \qquad\qquad \text{(rewriting on } \rho_y \approx \rho'_z) \\
& = & (\emptyset, \{\emptyset \subseteq \zeta_x, ob(\rho'_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, t'_x \mathcal{T} \rho'_z, \\
& & \qquad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \rho'_z \sim [\rho_w]^{t_z}/0, \\
& & \qquad rel([\rho'_z]^{t'_x}/(O_{\zeta_x}.0||I_{\zeta'_x}.0)), rel(\rho'_z)\}) \\
& \rightsquigarrow & [\rho'_z \mapsto [\rho_w]^{t_z}/\alpha_z](\emptyset, \{\emptyset \subseteq \zeta_x, ob(\rho'_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, t'_x \mathcal{T} \rho'_z, \\
& & \qquad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \rho'_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \\
& & \qquad rel([\rho'_z]^{t'_x}/(O_{\zeta_x}.0||I_{\zeta'_x}.0)), rel(\rho'_z)\}) \\
& & \qquad \text{(rewriting on } \rho'_z \sim [\rho_w]^{t_z}/0) \\
& = & (\emptyset, \{\emptyset \subseteq \zeta_x, ob([\rho_w]^{t_z}/\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, t'_x \mathcal{T} [\rho_w]^{t_z}/\alpha_z, \\
& & \qquad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, [\rho_w]^{t_z}/\alpha_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \\
& & \qquad rel([[\rho_w]^{t_z}/\alpha_z]^{t'_x}/(O_{\zeta_x}.0||I_{\zeta'_x}.0)), rel([\rho_w]^{t_z}/\alpha_z)\})
\end{array}
$$

Next, we can transform the other constraints on types by using the rules in Figure 5. We implicitly assume that expressions like $[\tilde{\tau}]^t/U_1 + [\tilde{\tau}]^t/U_2$ and $Uof([\tilde{\tau}]^t/U)$ are simplified as necessary into expressions like $[\tilde{\tau}]^t/(U_1||U_2)$ and U. So, for example, if $bool \leq bool + bool$ is in C, then it is first simplified into $bool \leq bool$

and the second rule of Figure 5 is applied.

Example 6.2: Let C_1 be the last set of constraints in Example 6.1. $(\emptyset, C_1)$ is further rewritten as follows.

$$\begin{array}{rll}
(\emptyset, C_1) & \leadsto & (\emptyset, \{\emptyset \subseteq \zeta_x, ob(\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, t'_x \mathcal{T} [\rho_w]^{t_z}/\alpha_z, \\
& & \quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, [\rho_w]^{t_z}/\alpha_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \\
& & \quad rel([[\rho_w]^{t_z}/\alpha_z]^{t'_x}/(O_{\zeta_x}.0 || I_{\zeta'_x}.0)), rel([\rho_w]^{t_z}/\alpha_z)\}) \\
& & \qquad \text{(rewriting on } ob([\rho_w]^{t_z}/\alpha_z)) \\
& \leadsto & (\emptyset, \{\emptyset \subseteq \zeta_x, ob(\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, ob(\alpha_z) \Rightarrow t'_x \mathcal{T} t_z, \\
& & \quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, [\rho_w]^{t_z}/\alpha_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0, \\
& & \quad rel([[\rho_w]^{t_z}/\alpha_z]^{t'_x}/(O_{\zeta_x}.0 || I_{\zeta'_x}.0)), rel([\rho_w]^{t_z}/\alpha_z)\}) \\
& & \qquad \text{(rewriting on } t'_x \mathcal{T} [\rho_w]^{t_z}/\alpha_z) \\
& \leadsto & (\emptyset, \{\emptyset \subseteq \zeta_x, ob(\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, ob(\alpha_z) \Rightarrow t'_x \mathcal{T} t_z, \\
& & \quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \alpha_z \leq I_{\zeta_z}.0, \\
& & \quad rel([[\rho_w]^{t_z}/\alpha_z]^{t'_x}/(O_{\zeta_x}.0 || I_{\zeta'_x}.0)), rel([\rho_w]^{t_z}/\alpha_z)\}) \\
& & \qquad \text{(rewriting on } [\rho_w]^{t_z}/\alpha_z \leq [\rho_w]^{t_z}/I_{\zeta_z}.0) \\
& \leadsto^* & (\emptyset, \{\emptyset \subseteq \zeta_x, ob(\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, ob(\alpha_z) \Rightarrow t'_x \mathcal{T} t_z, \\
& & \quad \emptyset \subseteq \zeta_z, noob(\rho_w), \mathbf{c} \subseteq \zeta'_x, \alpha_z \leq I_{\zeta_z}.0, rel(O_{\zeta_x}.0 || I_{\zeta'_x}.0), rel(\alpha_z)\}) \\
& & \qquad \text{(rewriting on } rel(\tau))
\end{array}$$

Rewriting by using the rules in Figures 4 and 5 always terminates.

Lemma 6.3 [termination]: Let P be a process expression. There is no infinite sequence $PT(P) \leadsto (\Gamma_1, C_1) \leadsto (\Gamma_2, C_2) \leadsto \cdots$.

Proof: We define the size $size_\tau(C)$ of a constraint set C by:

$$size_\tau(\{c_1, \ldots, c_n\}) = size_\tau(c_1) + \cdots + size_\tau(c_n)$$

$$\begin{array}{l}
size_\tau(\tau_1 \approx \tau_2) = size_\tau(\tau_1 \sim \tau_2) = size_\tau(\tau_1 \leq \tau_2) = size_\tau(\tau_1) + size_\tau(\tau_2) \\
size_\tau(noob(\tau)) = size_\tau(rel(\tau)) = size_\tau(\tau) \\
size_\tau((o_1 \vee \cdots \vee o_n) \Rightarrow c_a) = size_\tau(o_1) + \cdots + size_\tau(o_n) \\
size_\tau(t \mathcal{T} \tau) = size_\tau(\tau) \\
size_\tau(ob(\tau)) = size_\tau(\tau) \\
size_\tau(ob(U)) = 0 \\
size_\tau(c_U) = size_\tau(c_a) = size_\tau(c_t) = 0
\end{array}$$

$$\begin{array}{l}
size_\tau(\rho) = 1 \\
size_\tau(bool) = 1 \\
size_\tau([\tau_1, \ldots, \tau_n]^t/U) = size_\tau(\tau_1) + \cdots + size_\tau(\tau_n) + 1
\end{array}$$

Intuitively, $size_\tau(C)$ denotes the number of type constructors (channel types, *bool*, and type variables) appearing in C. The lemma follows from the fact that each rewriting step monotonically decreases the pair of (1) the number of type variables appearing in C and (2) $size_\tau(C)$, with respect to the lexicographic ordering of pairs (i.e., with respect to the partial order $\leq$ such that $(m, n) \leq (m', n')$ if and only if $m \leq m' \vee (m = m' \wedge n \leq n')$). □

It is easy to see that if (Γ, C) is a principal typing and $(\Gamma, C) \rightsquigarrow^* (\Gamma', C')$ by the rules in Figures 4 and 5, then (Γ', C') is also a principal typing. Moreover, all the rules in Figures 4 and 5 preserve the conditions in Lemma 5.7, unless **false** is added to the constraint. Therefore, the following conditions hold for (Γ, C) obtained by reducing $PT(P)$.

Lemma 6.4: Suppose $PT(P) \rightsquigarrow^* (\Gamma, C) \not\rightsquigarrow$. If **false** $\in C$, then P is not typable. Otherwise, (Γ, C) is a principal typing of P and the following conditions hold (if all expressions in C have been simplified as far as possible).

1. C contains no constraint of the form $\tau_1 \approx \tau_2$.
2. For every constraint $\tau_1 \sim \tau_2 \in C$, both τ_1 and τ_2 are type variables.
3. Every extended type appearing in C is composed of only type variables and operators $+, \sqcap, *$ (i.e., it contains no type expressions of the form *bool* or $[\tilde{\tau}]^t/U$).
4. For every constraint of the form $\tau_1 \leq \tau_2 \in C$ or $noob(\tau_1)$, τ_1 is a type variable.
5. An extended usage of the form $Uof(\tau)$ does not appear in C.
6. If $U_1 \leq U_2 \in C$ or $noob(U_1) \in C$, then U_1 is a usage variable.

Proof: By Theorem 5.6 and Lemma 5.7, $PT(P)$ is a principal typing of P and it satisfies the conditions of Lemma 5.7. Because each rule in Figures 4 and 5 preserves the principal typing property (each rule preserves the satisfiability of the constraint set, and changes the type environment only by applying a substitution θ such that the constraint set implies $\theta\rho \approx \rho$, $\theta\alpha = \alpha$ and $\theta t = t$), (Γ, C) is also a principal typing. By the definition of the principal typing, the unsatisfiability of C implies that there is no valid type judgment $\Gamma'; \mathcal{T} \vdash P$.

Each rule also preserves the conditions of Lemma 5.7. Therefore, the conditions 1–5 can be checked as follows.

1. If $\tau_1 \approx \tau_2 \in C$, then both τ_1 and τ_2 are composed only of type variables, *bool*, and channel type constructors (the fourth condition of Lemma 5.7). So, one of the rules in Figure 4 must be applicable, which contradicts with the assumption $(\Gamma, C) \not\rightsquigarrow$.
2. Similar to the above proof of the condition 1.
3. By the above conditions 1 and 2, and the fifth condition of Lemma 5.7, each type expression appearing in C is either a type expression composed only of type variables, *bool* and channel type constructors or a type expression composed only of type variables and operators $+, \sqcap, *$. The latter must be the case since $(\Gamma, C) \not\rightsquigarrow$.
4. This follows from the above condition 3 and the third condition of Lemma 5.7.
5. By the above conditions 1 and 2, and the fifth condition of Lemma 5.7, $Uof(\tau) \in C$ implies that τ is a channel type, which contradicts with the assumption $(\Gamma, C) \not\rightsquigarrow$.
6. $PT(P)$ generates no constraint of the form $U_1 \leq U_2$ or $noob(U_1)$. It is introduced only by the third or fifth rule in Figure 5. By the third condition of Lemma 5.7, it must be the case that U_1 is a usage variable.

□

Now, if **false** $\notin C$, then the remaining constraints on types are clearly satisfiable: if we instantiate remaining type variables with $[]^t/0$ for a fresh time tag t, then all the constraints on types are satisfied. So, we only need to check the satisfiability of constraints on usages, attributes, and time tags.

Example 6.5: By substituting $[]^t/0$ for the remaining type variable ρ_w in Example 6.2, we can eliminate the remaining constraint $noob(\rho_w)$ on types and obtain the following constraints:

$$\{\emptyset \subseteq \zeta_x, ob(\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, ob(\alpha_z) \Rightarrow t'_x \mathcal{T} t_z, \emptyset \subseteq \zeta_z, \mathbf{c} \subseteq \zeta'_x, \alpha_z \leq I_{\zeta_z}.0, rel(O_{\zeta_x}.0||I_{\zeta'_x}.0), rel(\alpha_z)\}$$

6.2 Reducing Constraints on Usages, Attributes, and Time tags

Now we describe how to reduce constraints on usages, attributes and time tags. Unlike the transformations presented so far, the transformation on usages is sound but incomplete: Although an unsatisfiable constraint set is never accepted, a satisfiable constraint set may be rejected. Because the reduction itself and the reason why it is not complete are rather complicated, we first give an overview in Section 6.2.1, and then describe details in the succeeding subsections.

6.2.1 Overview

By applying the rules in Section 6.1, we have constraints on usages of the form $\alpha \leq U$, $noob(\alpha)$, $(ob(U_1) \vee \cdots \vee ob(U_n)) \Rightarrow c_a$, $ob(U) \Rightarrow c_t$, or $rel(U)$. Because $noob(\alpha)$ is equivalent to $\alpha \leq 0$, and $\alpha \leq U_1 \wedge \alpha \leq U_2$ is equivalent to $\alpha \leq U_1 \sqcap U_2$, we can assume that the constraint set is of the form:

$$\begin{array}{l}\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n, \\ ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m, \\ rel(V_1), \ldots, rel(V_l)\}\end{array}$$

Here, $c_1, \ldots, c_m$ is either a constraint of the form $\mathbf{c} \subseteq a$ or $t_1 \mathcal{T} t_2$. $\alpha_1, \ldots, \alpha_n$ are mutually different usage variables. We can also assume that $\{\alpha_1, \ldots, \alpha_n\}$ contains all usage variables appearing in the constraint set, because otherwise we can add a constraint $\alpha \leq \alpha$ for variables not in $\{\alpha_1, \ldots, \alpha_n\}$.

The basic strategy for checking the satisfiability of the above constraint set is to choose a *normal solution* (the exact definition will be given later) of $\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$, obtained from replacing $\alpha_i \leq U_i$ with $\alpha_i = U_i$ (if $\alpha_i \notin FV(U_i)$) or $\alpha_i = \mathbf{rec}\, \alpha_i.U_i$ (if $\alpha_i \in FV(U_i)$), and checking the satisfiability of the other constraints for the normal solution. If the constraint set is satisfiable for the normal solution, then we can accept the input process as well-typed. Otherwise, the process is rejected. For the constraint in Example 6.5, for example, we check

$$\{\emptyset \subseteq \zeta_x, ob(\alpha_z) \Rightarrow \mathbf{c} \subseteq \zeta_x, ob(\alpha_z) \Rightarrow t'_x \mathcal{T} t_z, \emptyset \subseteq \zeta_z, \mathbf{c} \subseteq \zeta'_x, rel(O_{\zeta_x}.0||I_{\zeta'_x}.0), rel(\alpha_z)\}$$

for $\alpha_z = I_{\zeta_z}.0$. Because this is satisfiable (let $\zeta_x = \mathbf{o}$, $\zeta'_x = \mathbf{c}$, and $\zeta_z = \emptyset$), the input process $(\nu x)\,(\nu y)\,(x![y] \mid x?^c[z].\, z?[w].\, \mathbf{0})$ (which was given in Example 5.8) is accepted as well-typed.

The above strategy, however, is sound but incomplete. Even if a constraint set is unsatisfiable for a normal solution, it may be satisfiable for other solutions. There are the following two cases for this:

1. $rel(V_i)$ holds not for the normal solution but for other solutions.

$$(C \cup \{\alpha \leq U\}, \theta) \rightsquigarrow_{U1} ([\alpha \mapsto U]C, \theta \circ [\alpha \mapsto U]) \qquad \text{if } \alpha \notin FV(U)$$
$$(C \cup \{\alpha \leq U\}, \theta) \rightsquigarrow_{U1} ([\alpha \mapsto \mathbf{rec}\,\alpha.U]C, \theta \circ [\alpha \mapsto \mathbf{rec}\,\alpha.U]) \qquad \text{if } \alpha \in FV(U)$$

Figure 6: Rules for reducing usage constraints (1)

2. $ob(V_{ij})$ holds for the normal solution, but not for other solutions (so, c_i need not be satisfied for them).

Actually, the first case is not problematic in practice. Rather, rejecting the first case is preferable: If $rel(V_i)$ does not hold for a normal solution, then even if the input process is well typed, it is a bad process that may fall into a livelock (i.e., diverge without fulfilling some obligations). So, rather than seeking for a complete transformation method, we add extra conditions to the constraint set and solve them. It is discussed in Section 6.2.3.

The second case is indeed problematic. For example, consider the constraint set:

$$\{\alpha \leq I_a.0, ob(\alpha) \Rightarrow t\mathcal{T}t, rel(I_a.0||O_{\mathbf{c}}.0)\}.$$

Then, the third constraint implies that a must contain $\mathbf{o}$. So, if we choose a normal solution $\alpha = I_a.0$, then $ob(\alpha)$ holds. The second condition then implies $t\mathcal{T}t$, which is unsatisfiable since the tag ordering $\mathcal{T}$ must be irreflexive. If we take a non-normal solution $\alpha = 0 \sqcap I_a.0$ instead, then the constraint set is satisfiable since $ob(\alpha)$ does not hold.

In order to solve the above problem, we modify the basic strategy so that α constrained by $\alpha \leq U$ is instantiated with a normal solution $\mathbf{rec}\,\alpha.U$ only when $ob(\alpha)$ turns out to be true, and it is instantiated with a non-normal solution $\mathbf{rec}\,\alpha.(U \sqcap 0)$ otherwise. This refined strategy is complete with respect to the strengthened constraint (for rejecting the first case). We describe the refined strategy in Section 6.2.4.

6.2.2 Normal solutions

We first define a normal solution for a system of inequalities $\{\alpha_1 \leq U_1, \ldots, \alpha \leq U_n\}$.

Notation 6.6: If θ_1 and θ_2 are substitutions, we write $\theta_1 \circ \theta_2$ for the substitution such that $(\theta_1 \circ \theta_2)e = \theta_1(\theta_2 e)$ for any expression e.

Definition 6.7 [normal solutions]: Let $\alpha_1, \ldots, \alpha_n$ be mutually different variables and $\{\alpha_1, \ldots, \alpha_n\} \supseteq FV(U_1) \cup \cdots \cup FV(U_n)$. Then, a substitution θ is a *normal solution* of $\{\alpha_1 \leq U_1, \ldots, \alpha \leq U_n\}$ if $(\{\alpha_1 \leq U_1, \ldots, \alpha \leq U_n\}, id) \rightsquigarrow_{U1}^* (\emptyset, \theta)$ holds. Here, $\rightsquigarrow_{U1}$ is the least relation closed under the rules in Figure 6.

Example 6.8:

$$\begin{array}{l}(\{\alpha_1 \leq I_a.0||\alpha_2, \alpha_2 \leq \alpha_1\}, id)\\ \rightsquigarrow_{U1} (\{\alpha_1 \leq I_a.0||\alpha_1\}, [\alpha_2 \mapsto \alpha_1])\\ \rightsquigarrow_{U1} (\emptyset, [\alpha_1 \mapsto \mathbf{rec}\,\alpha_1.(I_a.0||\alpha_1), \alpha_2 \mapsto \mathbf{rec}\,\alpha_1.(I_a.0||\alpha_1)])\end{array}$$

It is easy to see that a normal solution is indeed a solution of the system of inequalities.

Lemma 6.9: Let $\alpha_1, \ldots, \alpha_n$ be mutually different variables and $\{\alpha_1, \ldots, \alpha_n\} \supseteq FV(U_1) \cup \cdots \cup FV(U_n)$. If $(\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}, id) \rightsquigarrow_{U1}^* (\emptyset, \theta)$, then $\theta\alpha_i \leq \theta U_i$ holds for each i.

Proof: Trivial by the fact that if θ is a solution of $[\alpha \mapsto \mathbf{rec}\,\alpha.U]C$, then $\theta \circ [\alpha \mapsto \mathbf{rec}\,\alpha.U]$ is a solution of $C \cup \{\alpha \leq U\}$. □

6.2.3 Adding extra conditions

The reason why $rel(V_i)$ holds not for a normal solution but for some non-normal solution is that the non-normal solution *over*-estimates the obligations fulfilled by a process. For example, consider the constraint set $\{\alpha \leq \alpha||I_{\mathbf{c}}.0, rel(\alpha)\}$. $rel(\alpha)$ does not hold for a normal solution $\alpha = \mathbf{rec}\,\alpha.(\alpha||I_{\mathbf{c}}.0)$, but it holds for a non-normal solution $\alpha = \mathbf{rec}\,\alpha.(\alpha||I_{\mathbf{c}}.0)||*O_{\mathbf{o}}.0$. As explained below, however, no process can actually fulfill output obligations in this case. Because an over-estimation of obligations results in accepting livelocking processes, we add extra conditions to avoid it. Let

$$\begin{array}{l}\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n,\\ ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m,\\ rel(V_1), \ldots, rel(V_l)\}\end{array}$$

be the original constraint set. Then, we compute a normal solution θ for $\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$, and add the constraints $ob_{\mathbf{I}}(\alpha_i) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_i)$ and $ob_{\mathbf{O}}(\alpha_i) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_i)$ for each i. The resulting constraint set is:

$$\begin{array}{l}\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n,\\ ob_{\mathbf{O}}(\alpha_1) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_1), \ldots, ob_{\mathbf{O}}(\alpha_n) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_n),\\ ob_{\mathbf{I}}(\alpha_1) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_1), \ldots, ob_{\mathbf{I}}(\alpha_n) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_n),\\ ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m,\\ rel(V_1), \ldots, rel(V_l)\}\end{array}$$

More justification for the addition of extra conditions is given below.

First, we note that it is only when α is not guarded by $I_a.$ or $O_a.$ in U of $\alpha \leq U$ that $rel(V_i)$ holds not for a normal solution but for a non-normal solution. It is implied by Lemma 6.11 and Lemma 4.27.

Definition 6.10: The set of top-level usage variables $\mathbf{TFV}(U)$ of a usage U is defined by:

$$\begin{array}{l}\mathbf{TFV}(0) = \mathbf{TFV}(I_a.U) = \mathbf{TFV}(O_a.U) = \emptyset\\ \mathbf{TFV}(\alpha) = \{\alpha\}\\ \mathbf{TFV}(U_1||U_2) = \mathbf{TFV}(U_1 \sqcap U_2) = \mathbf{TFV}(U_1) \cup \mathbf{TFV}(U_2)\\ \mathbf{TFV}(\mathbf{rec}\,\alpha.U) = \mathbf{TFV}(U)\backslash\{\alpha\}\\ \mathbf{TFV}(*U) = \mathbf{TFV}(U)\end{array}$$

We say α is guarded in U if $\alpha \notin \mathbf{TFV}(U)$.

Lemma 6.11: If α is guarded in U, then $U_1 \leq \mathbf{rec}\,\alpha.U$ for every usage U_1 such that $U_1 \leq [\alpha \mapsto U_1]U$.

Proof sketch: This follows from the fact that the relation $\leq \cup\{([\alpha \mapsto U_1]U_2, [\alpha \mapsto \mathbf{rec}\,\alpha.U]U_2) \mid U_2 \in \mathcal{U}\}$ is a usage simulation up to $\leq$. Details are omitted. □

By the above lemma, the case where $rel(V_i)$ holds for some non-normal solution but not for a normal solution is only when α appears in U at the top level in the constraint $\alpha \leq U$.

Looking carefully into the procedure PT, we know that problematic inequalities like $\alpha \leq \alpha||U$ are produced only when there is some process that receives a channel of the usage α and forwards it to itself, like the following:

$$*x?[y].\,(x![y] \mid P)$$

For example, the constraint

$$\{rel(U), \alpha \leq \alpha||I_{\mathbf{c}}.0\}$$

is produced from the following process

$$(\nu x)\,(\nu z)\,(x!\,[z] \mid *x?[y].\,(x![y] \mid y?^{\mathbf{c}}[\,].\,\mathbf{0}))$$

Let $[[\,]^{t_y}/\alpha]^{t_x}/\beta$ be the type of x. Then, the usage of y in $x![y] \mid y?^{\mathbf{c}}[\,].\,\mathbf{0}$ is expressed by $\alpha||I_{\mathbf{c}}.0$. So, we get the constraint $\alpha \leq \alpha||I_{\mathbf{c}}.0$ from $x?[y].\,(x![y] \mid y?^{\mathbf{c}}[\,].\,\mathbf{0})$. Although we can derive a valid type judgment for the whole process by substituting $\mathbf{rec}\,\beta.(\beta||O_{\mathbf{o}}.0)||\mathbf{rec}\,\beta.(\beta||I_{\mathbf{c}}.0)$ for α, this is clearly an over-estimation: The output obligation $\mathbf{rec}\,\beta.(\beta||O_{\mathbf{o}}.0)$ is actually delegated to itself through the channel x infinitely, and never fulfilled. Another solution $\alpha = \mathbf{rec}\,\beta.(\beta||I_{\mathbf{c}}.0)$, which means that no output obligation is performed, gives a better estimation of the actual usage of the channel y. Although the above process is rejected as a result, it is indeed a bad process that livelocks and never enables the input capability on y.

In general, when there is an inequality $\alpha \leq U$, we can regard $\mathbf{rec}\,\alpha.U$ as giving a minimal estimation of the obligations that are actually fulfilled, among the solutions of $\alpha \leq U$. This intuition is justified by Theorem 6.12 below, which means that if there is a valid type judgment $\Gamma, x : [\tilde{\tau}]^{t_x}/U; \mathcal{T} \vdash P$, then P can perform an input or an output with an attribute a on a channel x only if such a usage is specified by U. For example, there cannot be the case where $U = I_{\emptyset}.0$ but P can be reduced to $x?^{\mathbf{o}}[\tilde{y}].\,Q$. To put it in another way, if there is a valid type judgment $\Gamma, x : [\tilde{\tau}]^{t_x}/U; \mathcal{T} \vdash P$, then there is no danger that a use of x by P is overlooked in U. So, although there may be many solutions for a constraint $\alpha \leq U$, it is better to choose $\mathbf{rec}\,\alpha.U$, as it gives a minimal estimation of the actual usage.

Theorem 6.12: If $\Gamma, x : [\tilde{\tau}]^{t_x}/U; \mathcal{T} \vdash P$ and $P \longrightarrow^* (\nu\tilde{y})\,(x!^a[\tilde{v}].\,Q \mid R)$, then $U \longrightarrow^* \succeq O_{a'}.U_1||U_2$ for some U_1, U_2, a' such that $a \subseteq a'$. Similarly, If $\Gamma, x : [\tilde{\tau}]^{t_x}/U; \mathcal{T} \vdash P$ and $P \longrightarrow^* (\nu\tilde{y})\,(x?^a[\tilde{z}].\,Q \mid R)$, then $U \longrightarrow^* \succeq I_{a'}.U_1||U_2$ for some U_1, U_2, a' such that $a \subseteq a'$.

Proof: Suppose $\Gamma, x : [\tilde{\tau}]^{t_x}/U; \mathcal{T} \vdash P$ and $P \longrightarrow^* (\nu\tilde{y})\,(x!^a[\tilde{v}].\,Q \mid R)$. By the subject reduction theorem (Theorem 4.47), $\Gamma', x : [\tilde{\tau}]^{t_x}/U'; \mathcal{T} \vdash (\nu\tilde{y})\,(x!^a[\tilde{v}].\,Q \mid R)$ for some Γ' and U' such that $U \longrightarrow^* U'$. By typing rules, it must be the case that $U' \succeq O_{a'}.U_1||U_2$ and $a \subseteq a'$ for some a', U_1, and U_2. The case for input is similar. □

In the example above, because the process $Q = x!\,[z] \mid *x?[y].\,(x![y] \mid y?^{\mathbf{c}}[\,].\,\mathbf{0})$ is typed by:

$$x : [[\,]^{t_y}/U]^{t_x}/(O_{\mathbf{c}}.\mathbf{0} \mid *I_{\mathbf{o}}.\mathbf{0}), z : [\,]^{t_y}/U; \{(t_x, t_y)\} \vdash Q$$

for $U = \mathbf{rec}\,\alpha.(\alpha||I_{\mathbf{c}}.0)$, we know that Q never fulfills the output obligation on z.

Remark 6.13: One may expect that all processes that are well typed but rejected by our algorithm fall into a livelock. However, this is not the case. Consider the following process:

$$\begin{array}{l}(\nu x)\,(y![x] \mid x?^{\mathbf{c}}[\,].\,\mathbf{0}\\ \quad \mid *y?[z].\,(y![z] \mid w![z])\\ \quad \mid *w?[z].\,z![\,]\\ \quad \mid *w'?[z].\,\mathbf{0}\\ \quad \mid u![w] \mid u![w'])\end{array}$$

It creates a fresh channel x, sends it on y, and waits to input on x. Because x is forwarded to channel w by the process in the second line and received by the process in the third line, a null tuple is output on x. So, an input on x in the first line always succeeds.

However, if we choose $\mathbf{rec}\,\alpha.U$ as a solution of $\alpha \leq U$, the above process is rejected as ill-typed. From the second line, we get the constraint $\alpha \leq \alpha||\beta$, where α is the usage of a channel received by the process

of the second line, and β is the usage of a channel received by the process of the third line. Although the third process always performs an output on the received channel, our type system can only estimate β as $O_{\mathbf{o}}.0 \sqcap 0$: Because w and w' are sent on the same channel (the fifth line), w and w' must have the same parameter type. We therefore obtain the inequality $\alpha \leq \alpha||(O_{\mathbf{o}}.0 \sqcap 0)$. If $\mathbf{rec}\,\alpha.(\alpha||(O_{\mathbf{o}}.0 \sqcap 0))$ is chosen as its solution, then it is inferred that the output obligation may not be fulfilled and the process is rejected by our algorithm. The process is still well typed, because if we choose $O_{\mathbf{o}}.0||\mathbf{rec}\,\alpha.(\alpha||(O_{\mathbf{o}}.0 \sqcap 0))$ as a solution, it is judged that the output obligation may be fulfilled.

Note, however, that the reason for the above process being rejected should be attributed to the limit of the type system's ability to capture flow information: If the type system could infer that x is never received on w', then it could estimate the usage α as $\mathbf{rec}\,\alpha.(\alpha||O_{\mathbf{o}}.0)$, and therefore, infer that the output obligation on x is always fulfilled. (Note that although the problem of this particular example is solved by introducing subtyping discussed in Section 7.3, it is in general undecidable to statically obtain the exact flow information.) It is just a coincidence that the process can be judged to be well typed by choosing another solution $O_{\mathbf{o}}.0||\mathbf{rec}\,\alpha.(\alpha||(O_{\mathbf{o}}.0 \sqcap 0))$. Choosing this solution means allowing the output obligation $O_{\mathbf{o}}.0$ to be delegated infinitely through channel y, which is undesired.

Remark 6.14: Even if we always choose $\mathbf{rec}\,\alpha.U$ as a solution of $\alpha \leq U$, we cannot guarantee freedom from livelock. For example, consider the following process

$$f![y]\,|\,*f?[x].\,\mathbf{if}\ \mathit{true}\ \mathbf{then}\ f![x]\ \mathbf{else}\ x!^{\mathbf{o}}[\,]$$

The usage α of x is inferred as $\mathbf{rec}\,\alpha.(\alpha \sqcap O_{\mathbf{o}}.0)$, so it is inferred that the output obligation on y will be eventually fulfilled (unless the process diverges). It is true, but only because the process diverges. See Section 7.8 for more discussions on livelocks.

6.2.4 Reducing inequalities and reliability constraints

Now we can assume that the remaining constraint set is:

$$\begin{array}{l}\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n,\\ \ \mathit{ob}_{\mathbf{O}}(\alpha_1) \Rightarrow \mathit{ob}_{\mathbf{O}}(\theta\alpha_1), \ldots, \mathit{ob}_{\mathbf{O}}(\alpha_n) \Rightarrow \mathit{ob}_{\mathbf{O}}(\theta\alpha_n)\\ \ \mathit{ob}_{\mathbf{I}}(\alpha_1) \Rightarrow \mathit{ob}_{\mathbf{I}}(\theta\alpha_1), \ldots, \mathit{ob}_{\mathbf{I}}(\alpha_n) \Rightarrow \mathit{ob}_{\mathbf{I}}(\theta\alpha_n)\\ \ \mathit{ob}(V_{11}) \vee \cdots \vee \mathit{ob}(V_{1k_1}) \Rightarrow c_1, \cdots, \mathit{ob}(V_{m1}) \vee \cdots \vee \mathit{ob}(V_{mk_m}) \Rightarrow c_m,\\ \ \mathit{rel}(V_1), \ldots, \mathit{rel}(V_l)\}\end{array}$$

Moreover, by the definition of PT and the transformations described so far, we can assume that $\theta\alpha_i$ above are closed usages (i.e., $FV(\theta\alpha_i) = \emptyset$), and that the other usages do not contain the recursive usage constructor $\mathbf{rec}\,\alpha.$.

The next step is to eliminate reliability constraints of the form $\mathit{rel}(V_i)$.

In order to clarify the essence, we first obtain necessary and sufficient conditions for the reliability of a closed usage. By definition, $\mathit{rel}(U)$ holds if $\mathit{con}(U')$ holds for every U' such that $U \longrightarrow^* U'$. Let us define a constraint $\mathit{con}'(U)$ as follows.

Definition 6.15: A binary relation $\cong$ on usages is the least equivalence relation satisfying the following laws for the commutative monoid $(\mathcal{U}, ||, 0)$:

1. $0||U \cong U$
2. $U_1||U_2 \cong U_2||U_1$

3. $U_1||(U_2||U_3) \cong (U_1||U_2)||U_3$

Definition 6.16: $con'(U)$ is defined to hold if and only if (i) $U \cong I_a.U_1||U_2$ and $\mathbf{c} \subseteq a$ imply $ob_{\mathbf{O}}(U_2)$ and (ii) $U \cong O_a.U_1||U_2$ and $\mathbf{c} \subseteq a$ imply $ob_{\mathbf{I}}(U_2)$.

Because $con'(U_1||U_2)$ holds if and only if $con'(U_2||U_1)$ holds, we extend con' to a predicate on sets of usages by: $con'(\{U_1, \ldots, U_n\}) \iff con'(U_1|| \cdots ||U_n)$.

Then, $rel(U)$ can be reduced to a set of constraints of the form $con'(U')$ by the following lemma.

Lemma 6.17: $rel(U)$ if and only if every constraint in $\{con'(U') \mid U \longrightarrow^* \succeq U'\}$ holds.

Proof: Note that by the definition of rel, $rel(U)$ holds if and only if $con(U')$ holds for every U' such that $U \longrightarrow^* U'$.

Suppose $rel(U)$ holds and $U \longrightarrow^* \succeq U'$. Then, there exists U'' such that $U \longrightarrow^* U''$ and $U'' \succeq U'$. By the assumption, $con(U'')$ holds. So, $con'(U)$ must hold by the definition of con.

On the other hand, suppose every constraint in $\{con'(U') \mid U \longrightarrow^* \succeq U'\}$ holds and $U \longrightarrow^* U'$. It suffices to show that $con(U')$ holds. If $U' \succeq I_a.U_1||U_2$, then $con'(I_a.U_1||U_2)$. So, $\mathbf{c} \subseteq a$ implies $ob_{\mathbf{O}}(U_2)$. Similarly, $U' \succeq O_a.U_1||U_2$ and $\mathbf{c} \subseteq a$ imply $ob_{\mathbf{I}}(U_2)$. Therefore, we have $con(U')$. □

If U contains no recursive usages, the set $\{U' \mid U \longrightarrow^* \succeq U'\}$ is finite, and therefore, we can straightforwardly reduce $rel(U)$ to constraints of the form $con'(U')$, which can further be reduced to constraints of the forms $\mathbf{c} \subseteq a \Rightarrow ob_{\mathbf{O}}(U')$ and $\mathbf{c} \subseteq a \Rightarrow ob_{\mathbf{I}}(U')$. If U contains recursive usages, however, the set $\{U' \mid U \longrightarrow^* \succeq U'\}$ may be infinite. The following lemma helps reducing the search space.

Lemma 6.18: For any usages U_1 and U_2, $con'(U_1||U_2)$ if and only if $con'(U_1||U_2||U_2)$.

Proof: First, note that $ob_{\mathbf{O}}(U_1||U_2)$ ($ob_{\mathbf{I}}(U_1||U_2)$, resp.) holds if and only if $ob_{\mathbf{O}}(U_1) \vee ob_{\mathbf{O}}(U_2)$ ($ob_{\mathbf{I}}(U_1) \vee ob_{\mathbf{I}}(U_2)$, resp.) holds.

$\Rightarrow$: Suppose $con'(U_1||U_2)$. Suppose also that $U_1||U_2||U_2 \cong I_a.U_3||U_4$, and $\mathbf{c} \subseteq a$. Then, either (i)$U_1 \cong I_a.U_3||U_5$ and $U_5||U_2||U_2 \cong U_4$, or (ii)$U_2 \cong I_a.U_3||U_5$ and $U_1||U_5||U_2 \cong U_4$. In the former case, $U_1||U_2 \cong I_a.U_3||(U_5||U_2)$. So, by the assumption $con'(U_1||U_2)$, it must be the case that $ob_{\mathbf{O}}(U_5||U_2)$, which implies $ob_{\mathbf{O}}(U_5||U_2||U_2)$. We therefore have $ob_{\mathbf{O}}(U_4)$. In the latter case, $U_1||U_2 \cong I_a.U_3||(U_1||U_5)$. By the assumption $con'(U_1||U_2)$, it must be the case that $ob_{\mathbf{O}}(U_1||U_5)$, which implies $ob_{\mathbf{O}}(U_1||U_5||U_2)$. So, we have $ob_{\mathbf{O}}(U_4)$. Similarly, $U_1||U_2||U_2 \cong O_a.U_3||U_4$, and $\mathbf{c} \subseteq a$ imply $ob_{\mathbf{I}}(U_4)$. Therefore, we have $con'(U_1||U_2||U_2)$.

$\Leftarrow$: Suppose $con'(U_1||U_2||U_2)$. Suppose also that $U_1||U_2 \equiv I_a.U_3||U_4$, and $\mathbf{c} \subseteq a$. Then, either (i)$U_1 \cong I_a.U_3||U_5$ and $U_4 \cong U_5||U_2$ or (ii)$U_2 \cong I_a.U_3||U_5$ and $U_4 \cong U_5||U_1$. In the former case, $U_1||U_2||U_2 \cong I_a.U_3||(U_5||U_2||U_2)$. By the assumption $con'(U_1||U_2||U_2)$, it must be the case that $ob_{\mathbf{O}}(U_5||U_2||U_2)$ holds. It also implies $ob_{\mathbf{O}}(U_5||U_2)$ by:

$$ob_{\mathbf{O}}(U_5||U_2||U_2) \iff ob_{\mathbf{O}}(U_5) \vee ob_{\mathbf{O}}(U_2) \vee ob_{\mathbf{O}}(U_2) \iff ob_{\mathbf{O}}(U_5) \vee ob_{\mathbf{O}}(U_2) \iff ob(U_5||U_2).$$

So, $ob_{\mathbf{O}}(U_4)$. In the latter case, $U_1||U_2||U_2 \cong I_a.U_3||(U_5||U_1||U_2)$. By the assumption $con'(U_1||U_2||U_2)$, it must be the case that $ob_{\mathbf{O}}(U_5||U_1||U_2)$. It implies $ob_{\mathbf{O}}(U_4)$ by:

$$ob_{\mathbf{O}}(U_5||U_1||U_2) \iff ob_{\mathbf{O}}(U_5||(I_a.U_3||U_5)||U_2) \iff ob_{\mathbf{O}}(U_5) \vee ob_{\mathbf{O}}(I_a.U_3) \vee ob_{\mathbf{O}}(U_5) \vee ob_{\mathbf{O}}(U_2)$$
$$\iff ob_{\mathbf{O}}(U_5) \vee ob_{\mathbf{O}}(U_2) \iff ob_{\mathbf{O}}(U_5||U_2) \iff ob_{\mathbf{O}}(U_4)$$

Similarly, $U_1||U_2 \cong O_a.U_3||U_4$, and $\mathbf{c} \subseteq a$ imply $ob_{\mathbf{I}}(U_4)$. Therefore, we have $con'(U_1||U_2)$.

□

The above lemma implies that multiple occurrences of the same usage do not matter. For example, $con'(I_{a_1}.U_1||I_{a_1}.U_1||O_{a_2}.U_2||O_{a_2}.U_2||O_{a_2}.U_2)$ holds if and only if $con'(I_{a_1}.U_1||O_{a_2}.U_2)$ holds. So, let $Comp(U)$ be the set of usages defined below, from which U can be constructed by using $||$ and 0. Then, $rel(U)$ is reduced to the set $\{con'(Comp(U')) \mid U \longrightarrow^* \succeq U'\}$ of constraints.

Definition 6.19 [atomic usages]: A usage is *atomic* if it is not of the form 0 or $U_1||U_2$. We write $\mathcal{U}_{atom}$ for the set of atomic usages.

Definition 6.20: A mapping $Comp$ from $\mathcal{U}$ to $2^{\mathcal{U}_{atom}}$ is defined by:

$$\begin{array}{l} Comp(0) = \emptyset \\ Comp(\alpha) = \{\alpha\} \\ Comp(I_a.U) = \{I_a.U\} \\ Comp(O_a.U) = \{O_a.U\} \\ Comp(U_1||U_2) = Comp(U_1) \cup Comp(U_2) \\ Comp(U_1 \sqcap U_2) = \{U_1 \sqcap U_2\} \\ Comp(\mathbf{rec}\,\alpha.U) = \{\mathbf{rec}\,\alpha.U\} \\ Comp(*U) = \{*U\} \end{array}$$

For example, $Comp(I_{a_1}.U_1||I_{a_1}.U_1||O_{a_2}.U_2||O_{a_2}.U_2||O_{a_2}.U_2) = \{I_{a_1}.U_1, O_{a_2}.U_2\}$.

Lemma 6.21: $rel(U)$ holds if and only if every constraint in $\{con'(Comp(U')) \mid U \longrightarrow^* \succeq U'\}$ holds.

Proof: Suppose $rel(U)$ holds. Then, by Lemma 6.17, $con'(U')$ holds for every U' such that $U \longrightarrow^* \succeq U'$. By Lemma 6.18, $con'(Comp(U'))$ holds for every U' such that $U \longrightarrow^* \succeq U'$.

On the other hand, suppose $\{con'(Comp(U')) \mid U \longrightarrow^* \succeq U'\}$ holds. Then, by Lemma 6.18, $con'(U')$ holds for every U' such that $U \longrightarrow^* \succeq U'$. Therefore, $rel(U)$ must hold by the definition of rel. □

By the above lemma, to reduce $rel(U)$, it suffices to find the set $\{Comp(U') \mid U \longrightarrow^* \succeq U'\}$.

Example 6.22: Let $U = I_{a_1}.0||\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha)$. Then, $\{Comp(U') \mid U \longrightarrow^* \succeq U'\}$ is:

$$\{\{I_{a_1}.0, \mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha)\}, \{I_{a_1}.0, \mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha), O_{a_2}.0\}, \{\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha)\}, \{\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha), O_{a_2}.0\}\}.$$

So, $rel(U)$ is reduced to the following set of constraints:

$$\begin{array}{l} \{\mathbf{c} \subseteq a_1 \Rightarrow ob_{\mathbf{O}}(\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha)), \\ \ \mathbf{c} \subseteq a_1 \Rightarrow ob_{\mathbf{O}}(\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha)||O_{a_2}.0), \\ \ \mathbf{c} \subseteq a_2 \Rightarrow ob_{\mathbf{I}}(I_{a_1}.0||\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha)), \\ \ \mathbf{c} \subseteq a_2 \Rightarrow ob_{\mathbf{I}}(\mathbf{rec}\,\alpha.(O_{a_2}.0||\alpha))\}, \end{array}$$

which can further be simplified to $\{\mathbf{c} \subseteq a_1 \Rightarrow \mathbf{o} \subseteq a_2, \mathbf{c} \subseteq a_2 \Rightarrow \mathbf{false}\}$ by reducing $ob_{\mathbf{O}}$ and $ob_{\mathbf{I}}$.

Now we turn to the problem of reducing reliability constraints on usages containing free usage variables $\alpha_1, \ldots, \alpha_n$ constrained by $\alpha \leq U_1, \ldots, \alpha \leq U_n$. Because the set $\{Comp(U') \mid \theta U \longrightarrow^* \succeq U'\}$ depends on the substitution θ for free usage variables, we cannot directly use Lemma 6.21 to reduce the satisfiability of $rel(U)$. We use the following slightly different set.

Definition 6.23: Let $C = \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$. $\succeq_C$ is the least relation closed under the rules for $\succeq$ and the rules $\alpha_i \succeq_C U_i$ for $i \in \{1, \ldots, n\}$. $Deriv(U, C)$ is the set $\{Comp(U') \mid U \succeq_C (\longrightarrow \succeq_C)^* U'\}$.

The following lemma gives necessary conditions for the satisfiability of reliability constraints.

Lemma 6.24: Suppose $FV(V) \subseteq \{\alpha_1, \ldots, \alpha_n\}$. If a substitution θ satisfies $\{rel(V), \alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$, then θ must also satisfy $con'(S)$ for each $S \in Deriv(V, \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\})$.

Proof: Trivial from Lemma 6.21 and Lemma 4.27. □

Actually, the satisfiability of $C \cup Deriv(V, \alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n)$ is a not only necessary and but also sufficient condition for $C \cup \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n, rel(V)\}$.

Theorem 6.25: Suppose that θ is a normal solution of $C = \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$. Let C_1 be a set of constraints:

$$\begin{array}{l}\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n, \\ ob_{\mathbf{O}}(\alpha_1) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_1), \ldots, ob_{\mathbf{O}}(\alpha_n) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_n) \\ ob_{\mathbf{I}}(\alpha_1) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_1), \ldots, ob_{\mathbf{I}}(\alpha_n) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_n) \\ ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m, \\ rel(V_1), \ldots, rel(V_l)\}\end{array}$$

and C_2 be a set of constraints:

$$\begin{array}{l}\{ob_{\mathbf{O}}(\alpha_1) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_1), \ldots, ob_{\mathbf{O}}(\alpha_n) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_n) \\ ob_{\mathbf{I}}(\alpha_1) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_1), \ldots, ob_{\mathbf{I}}(\alpha_n) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_n) \\ ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m\} \\ \cup\{con'(S) \mid S \in Deriv(V_1, C)\} \\ \cup \cdots \cup \{con'(S) \mid S \in Deriv(V_l, C)\}.\end{array}$$

Then, C_1 is satisfiable if and only if C_2 is satisfiable.

Proof: The "only if" part follows immediately from Lemma 6.24.

Suppose C_2 is satisfiable. Let θ_1 be a solution of C_2. Then, let us define $U_1', \ldots, U_n'$ by:

$$U_i' = \begin{cases} U_i & \text{if } ob(\theta\alpha_i) \\ U_i \sqcap 0 & \text{otherwise} \end{cases}$$

Let θ_2 be a normal solution of $\{\alpha_1 \leq U_1', \ldots, \alpha_n \leq U_n'\}$ and θ_3 be $(\theta_1 \backslash \{\alpha_1, \ldots, \alpha_n\}) \circ \theta_2$. We show that θ_3 is a solution of C_1, which proves the theorem. First, θ_3 satisfies $\alpha_i \leq U_i$, since $\theta_3\alpha_i \leq \theta_3 U_i' \leq \theta_3 U_i$. Because $ob_{\mathbf{O}}(U \sqcap 0)$ implies $ob_{\mathbf{O}}(U)$, θ_3 also satisfies $ob_{\mathbf{O}}(\alpha_i) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_i)$. Similarly, θ_3 satisfies $ob_{\mathbf{I}}(\alpha_i) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_i)$.

Next, we show that θ_3 satisfies the set of constraints:

$$\{ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m\}$$

Because $c_1, \ldots, c_m$ are constraints on attributes and time tags, $\theta_1 c_i$ holds if and only if $\theta_3 c_i$ holds. Moreover, the value of $ob(V_{ij})$ is monotonic with respect to the values of $ob(\alpha_1), \ldots, ob(\alpha_n)$ (Theorem 4.20 in Section 6.2.5). So, it suffices to show that $ob(\theta_3\alpha_i)$ implies $ob(\theta_1\alpha_i)$. Suppose $ob(\theta_3\alpha_i)$ holds. Because θ_3 is a solution of $\alpha_i \leq U_i'$, it must be the case that $ob(\theta_3 U_i')$ must also hold. Then, by the definition of U_i', it must be the case that $U_i' = U_i$ and $ob(\theta_1\alpha_i)$.

Finally, we check θ_3 satisfies $rel(V_i)$ by contraposition. Suppose that $\theta V_i \longrightarrow^* V$, $Comp(V) = \{I_a.V', W_1, \ldots, W_k\}$, and $\mathbf{c} \subseteq a$, but that $ob_{\mathbf{O}}(W_1) \vee \cdots \vee ob_{\mathbf{O}}(W_k)$ does not hold. Without loss of generality, we can assume that none of $W_1, \ldots, W_k$ is of the form $W' \sqcap W''$. By the construction of U_i' and θ_3, it must be the case that there exist $V'', W_1', \ldots, W_p', \theta_{31}, \ldots, \theta_{3p}$ such that

$$
\begin{array}{l}
\{I_{a'}.V'', W_1', \ldots, W_p'\} \in Deriv(V_i, \alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n) \\
\theta_3 a' = a \\
\theta_3 V'' = V' \\
Comp(\theta_{31} W_1' || \cdots || \theta_{3p} W_p') = \{W_1, \ldots, W_k\} \\
\theta_{3j}\alpha_i = \theta_3 U_i \text{ if } ob(\theta\alpha_i) \text{ holds} \\
\theta_{3j}\alpha_i \in \{0, \theta_3 U_i\} \text{ if } ob(\theta\alpha_i) \text{ does not hold} \\
\theta_{3j}\zeta = \theta_3\zeta \text{ for each attribute variable } \zeta
\end{array}
$$

Because θ_1 satisfies $con'(\{I_{a'}.V'', W_1', \ldots, W_p'\})$ and $\theta_1 a' = \theta_3 a' = a \supseteq \mathbf{c}$, $ob_{\mathbf{O}}(\theta_1 W_1') \vee \cdots \vee ob_{\mathbf{O}}(\theta_1 W_p')$ must hold. Because $ob_{\mathbf{O}}(\theta_{31} W_1' || \cdots || \theta_{3p} W_p')$ does not hold, it must be the case that there exists W_j' such that $W_j' = \alpha_i$, $ob_{\mathbf{O}}(\theta_1\alpha_i)$, and $\neg ob_{\mathbf{O}}(\theta_{3j}\alpha_i)$. Because θ_1 satisfies $ob_{\mathbf{O}}(\alpha_i) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_i)$, we also have $ob_{\mathbf{O}}(\theta\alpha_i)$. By $\neg ob_{\mathbf{O}}(\theta_{3j}\alpha_i)$ and $ob_{\mathbf{O}}(\theta\alpha_i)$, it must be the case that

$$
\begin{array}{l}
\{\alpha_{i1}, \ldots, \alpha_{iq}, I_{a_{i1}}..W_{i1}', \ldots, I_{a_{ir}}.W_{ir}', O_{a_{i1}'}.W_{i1}'', \ldots, O_{a_{is}'}.W_{is}''\} \in Deriv(\alpha_i, C) \\
\mathbf{o} \not\sqsubseteq \theta_1 a_{i1}', \ldots, \theta_1 a_{is}' \\
\neg ob_{\mathbf{O}}(\theta_1\alpha_{i1}), \ldots, \neg ob_{\mathbf{O}}(\theta_1\alpha_{iq})
\end{array}
$$

By replacing each $W_j' = \alpha_i$ such that $ob_{\mathbf{O}}(\theta_1\alpha_i)$ with the above set $\{\alpha_{i1}, \ldots, \alpha_{iq}, I_{a_{i1}}..W_{i1}', \ldots, I_{a_{ir}}.W_{ir}', O_{a_{i1}'}.W_{i1}'', \ldots, O_{a_{is}'}.W_{is}''\}$, we obtain $\{I_{a'}.V'', W_1', \ldots, W_{p'}'\} \in Deriv(V_i, C)$ such that $\mathbf{c} \subseteq \theta_1 a'$ but $ob_{\mathbf{O}}(W_1') \vee \cdots \vee ob_{\mathbf{O}}(W_{p'}')$ does not hold, which contradicts with the assumption that θ_1 is a solution of C_2. □

Example 6.26: Let $V = I_{a_1}.0||\alpha$ and $C = \{\alpha \leq O_{a_2}.0||\alpha, ob_{\mathbf{O}}(\alpha) \Rightarrow ob_{\mathbf{O}}(\mathbf{rec}\,\alpha.(I_{a_1}.0||\alpha)), ob_{\mathbf{I}}(\alpha) \Rightarrow ob_{\mathbf{I}}(\mathbf{rec}\,\alpha.(I_{a_1}.0||\alpha)), rel(V)\}$. Then, $Deriv(V, \{\alpha \leq O_{a_2}.0||\alpha\})$ is:

$$\{\{I_{a_1}.0, \alpha\}, \{I_{a_1}.0, \alpha, O_{a_2}.0\}, \{\alpha\}, \{\alpha, O_{a_2}.0\}\}.$$

So, C can be transformed into:

$$
\begin{array}{l}
\{ob_{\mathbf{O}}(\alpha) \Rightarrow ob_{\mathbf{O}}(\mathbf{rec}\,\alpha.(I_{a_1}.0||\alpha)), ob_{\mathbf{I}}(\alpha) \Rightarrow ob_{\mathbf{I}}(\mathbf{rec}\,\alpha.(I_{a_1}.0||\alpha)), \\
\mathbf{c} \subseteq a_1 \Rightarrow ob_{\mathbf{O}}(\alpha), \\
\mathbf{c} \subseteq a_1 \Rightarrow ob_{\mathbf{O}}(\alpha||O_{a_2}.0), \\
\mathbf{c} \subseteq a_2 \Rightarrow ob_{\mathbf{I}}(I_{a_1}.0||\alpha), \\
\mathbf{c} \subseteq a_2 \Rightarrow ob_{\mathbf{I}}(\alpha)\}.
\end{array}
$$

By Theorem 6.25, we can eliminate inequalities and reliability constraints on usages and obtain a finite set of constraints on obligation constraints and constraints on attributes and time tags if the set $Deriv(U, \alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n)$ is finite. Fortunately, this is always the case, as shown in Lemma 6.30 below.

Definition 6.27: A mapping $SubUExp$ from $\mathcal{U}$ to $2^{\mathcal{U}_{atom}}$ is defined by:

$$
\begin{array}{l}
SubUExp(0) = \emptyset \\
SubUExp(\alpha) = \{\alpha\} \\
SubUExp(I_a.U) = \{I_a.U\} \cup SubUExp(U) \\
SubUExp(O_a.U) = \{O_a.U\} \cup SubUExp(U) \\
SubUExp(U_1||U_2) = SubUExp(U_1) \cup SubUExp(U_2) \\
SubUExp(U_1 \sqcap U_2) = \{U_1 \sqcap U_2\} \cup SubUExp(U_1) \cup SubUExp(U_2) \\
SubUExp(\mathbf{rec}\,\alpha.U) = \{\mathbf{rec}\,\alpha.U\} \cup [\alpha \mapsto \mathbf{rec}\,\alpha.U]SubUExp(U) \\
SubUExp(*U) = \{*U\} \cup SubUExp(U)
\end{array}
$$

Lemma 6.28: For any usage U, $SubUExp(U)$ is a finite set.

Proof: Straightforward induction on the structure of U. □

Lemma 6.29: Suppose $C = \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$. If $U \succeq_C U'$, then $SubUExp(U') \subseteq (SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n))$. Also, if $U \longrightarrow U'$, then $SubUExp(U') \subseteq SubUExp(U)$.

Proof: Straightforward induction on derivations of $U \succeq_C U'$ and $U \longrightarrow U'$. □

Lemma 6.30: For any usage U, the set $Deriv(U, \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\})$ is finite.

Proof: Let $C = \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$. By Lemma 6.29, $U \succeq_C (\longrightarrow \succeq_C)^* U'$ implies $SubUExp(U') \subseteq SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n)$. Because $Comp(U') \subseteq SubUExp(U')$, we have $Deriv(U, C) \subseteq 2^{SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n)}$. The result follows, since $SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n)$ is finite (Lemma 6.28). □

We can compute the set $Deriv(U, C)$ by reducing it to the reachability problem of Petri nets [4]. It is explained later in Section 6.2.7.

6.2.5 Reducing obligation constraints

The remaining constraints on usages are of the form

$$
\begin{array}{l}
\{ob_{\mathbf{O}}(\alpha_1) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_1), \ldots, ob_{\mathbf{O}}(\alpha_n) \Rightarrow ob_{\mathbf{O}}(\theta\alpha_n) \\
\ ob_{\mathbf{I}}(\alpha_1) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_1), \ldots, ob_{\mathbf{I}}(\alpha_n) \Rightarrow ob_{\mathbf{I}}(\theta\alpha_n) \\
\ ob(V_{11}) \vee \cdots \vee ob(V_{1k_1}) \Rightarrow c_1, \cdots, ob(V_{m1}) \vee \cdots \vee ob(V_{mk_m}) \Rightarrow c_m, \\
\ \mathbf{c} \subseteq a_1 \Rightarrow (ob_{\mathbf{O}}(U_{11}) \vee \cdots \vee ob_{\mathbf{O}}(U_{1l_1})), \ldots, \mathbf{c} \subseteq a_p \Rightarrow (ob_{\mathbf{O}}(U_{p1}) \vee \cdots \vee ob_{\mathbf{O}}(U_{pl_p})), \\
\ \mathbf{c} \subseteq a'_1 \Rightarrow (ob_{\mathbf{I}}(U'_{11}) \vee \cdots \vee ob_{\mathbf{O}}(U'_{1l'_1})), \ldots, \mathbf{c} \subseteq a'_{p'} \Rightarrow (ob_{\mathbf{I}}(U'_{p'1}) \vee \cdots \vee ob_{\mathbf{O}}(U'_{p'l'_{p'}}))\}
\end{array}
$$

where $\theta\alpha_i$ is a closed usage, and in other usages, recursive usage constructors appears only in the form $\mathbf{rec}\,\alpha.(\alpha||U)$.

The next step is to reduce obligation constraints into those of the form $ob_{\mathbf{O}}(\alpha)$ or $ob_{\mathbf{I}}(\alpha)$ and constraints on attributes and time tags.

Since $ob(U)$ is equivalent to $ob_{\mathbf{I}}(U) \vee ob_{\mathbf{O}}(U)$, we show how to reduce constraints of the form $ob_{\mathbf{I}}(U)$ and $ob_{\mathbf{O}}(U)$. U is either a closed usage or a usage containing no recursive usage constructor.

If U contains no recursive usage constructor, $ob_{\mathbf{O}}(U)$ and $ob_{\mathbf{O}}(U)$ can be decomposed by using Lemma 4.20.

For a closed usage U, $ob_{\mathbf{I}}(U)$ and $ob_{\mathbf{O}}(U)$ can be reduced to the constraints $ob^*_{\mathbf{I}}(U, id)$ and $ob^*_{\mathbf{O}}(U, id)$ (recall that id is the identity substitution) defined in Figure 7.

$$
\begin{aligned}
&ob^*_{\mathbf{O}}(0,\theta) = \mathbf{false}\\
&ob^*_{\mathbf{O}}(\alpha,\theta) = \mathbf{true}\\
&ob^*_{\mathbf{O}}(O_a.U,\theta) = \mathbf{o} \subseteq a\\
&ob^*_{\mathbf{O}}(I_a.U,\theta) = \mathbf{false}\\
&ob^*_{\mathbf{O}}(U_1||U_2,\theta) = ob^*_{\mathbf{O}}(U_1,\theta) \vee ob^*_{\mathbf{O}}(U_2,\theta)\\
&ob^*_{\mathbf{O}}(U_1 \sqcap U_2,\theta) = ob^*_{\mathbf{O}}(U_1,\theta) \wedge ob^*_{\mathbf{O}}(U_2,\theta)\\
&ob^*_{\mathbf{O}}(\mathbf{rec}\,\alpha.U,\theta) = mayob^*_{\mathbf{O}}(\theta U) \wedge ob^*_{\mathbf{O}}(U, \theta \circ [\alpha \mapsto \mathbf{rec}\,\alpha.\theta U])\\
&ob^*_{\mathbf{O}}(*U,\theta) = ob^*_{\mathbf{O}}(U,\theta)
\end{aligned}
$$

$$
\begin{aligned}
&mayob^*_{\mathbf{O}}(0) = \mathbf{false}\\
&mayob^*_{\mathbf{O}}(\alpha) = \mathbf{false}\\
&mayob^*_{\mathbf{O}}(O_a.U) = \mathbf{o} \subseteq a\\
&mayob^*_{\mathbf{O}}(I_a.U) = \mathbf{false}\\
&mayob^*_{\mathbf{O}}(U_1||U_2) = mayob^*_{\mathbf{O}}(U_1) \vee mayob^*_{\mathbf{O}}(U_2)\\
&mayob^*_{\mathbf{O}}(U_1 \sqcap U_2) = mayob^*_{\mathbf{O}}(U_1) \vee mayob^*_{\mathbf{O}}(U_2)\\
&mayob^*_{\mathbf{O}}(\mathbf{rec}\,\alpha.U) = mayob^*_{\mathbf{O}}(U)\\
&mayob^*_{\mathbf{O}}(*U) = mayob^*_{\mathbf{O}}(U)
\end{aligned}
$$

$$
\begin{aligned}
&ob^*_{\mathbf{I}}(0,\theta) = \mathbf{false}\\
&ob^*_{\mathbf{I}}(\alpha,\theta) = \mathbf{true}\\
&ob^*_{\mathbf{I}}(O_a.U,\theta) = \mathbf{false}\\
&ob^*_{\mathbf{I}}(I_a.U,\theta) = \mathbf{o} \subseteq a\\
&ob^*_{\mathbf{I}}(U_1||U_2,\theta) = ob^*_{\mathbf{I}}(U_1,\theta) \vee ob^*_{\mathbf{I}}(U_2,\theta)\\
&ob^*_{\mathbf{I}}(U_1 \sqcap U_2,\theta) = ob^*_{\mathbf{I}}(U_1,\theta) \wedge ob^*_{\mathbf{I}}(U_2,\theta)\\
&ob^*_{\mathbf{I}}(\mathbf{rec}\,\alpha.U,\theta) = mayob^*_{\mathbf{I}}(\theta U) \wedge ob^*_{\mathbf{I}}(U, \theta \circ [\alpha \mapsto \mathbf{rec}\,\alpha.\theta U])\\
&ob^*_{\mathbf{I}}(*U,\theta) = ob^*_{\mathbf{I}}(U,\theta)
\end{aligned}
$$

$$
\begin{aligned}
&mayob^*_{\mathbf{I}}(0) = \mathbf{false}\\
&mayob^*_{\mathbf{I}}(\alpha) = \mathbf{false}\\
&mayob^*_{\mathbf{I}}(O_a.U) = \mathbf{false}\\
&mayob^*_{\mathbf{I}}(I_a.U) = \mathbf{o} \subseteq a\\
&mayob^*_{\mathbf{I}}(U_1||U_2) = mayob^*_{\mathbf{I}}(U_1) \vee mayob^*_{\mathbf{I}}(U_2)\\
&mayob^*_{\mathbf{I}}(U_1 \sqcap U_2) = mayob^*_{\mathbf{I}}(U_1) \vee mayob^*_{\mathbf{I}}(U_2)\\
&mayob^*_{\mathbf{I}}(\mathbf{rec}\,\alpha.U) = mayob^*_{\mathbf{I}}(U)\\
&mayob^*_{\mathbf{I}}(*U) = mayob^*_{\mathbf{I}}(U)
\end{aligned}
$$

Figure 7: Functions for reducing obligation constraints

Theorem 6.31: Let U be a closed usage (i.e., a usage containing no free usage variables) and id be the identity substitution (i.e., the substitution whose domain is empty). Then, $ob_{\mathbf{O}}(U)$ ($ob_{\mathbf{I}}(U)$, resp.) holds if and only if $ob^*_{\mathbf{O}}(U, id)$ ($ob^*_{\mathbf{I}}(U, id)$, resp.) holds.

We prove the above theorem after introducing several lemmas. Readers who are not interested in the proof can safely skip to Section 6.2.6.

Lemma 6.32: Let U_1 and U_2 be usages. If $mayob^*_{\mathbf{O}}(U_2)$ does not hold, then $mayob^*_{\mathbf{O}}([\alpha \mapsto U_2]U_1)$ holds if and only if $mayob^*_{\mathbf{O}}(U_1)$ holds.

Proof: Straightforward induction on the structure of U_1. Note that the base case for $U_1 = \alpha$ follows from the fact that both $mayob^*_{\mathbf{O}}(U_2)$ and $mayob^*_{\mathbf{O}}(\alpha)$ are false. □

Definition 6.33: $mayob_{\mathbf{I}}(U)$ holds if and only if there exist a, U_1, and U_2 such that $U \succeq I_a.U_1||U_2$ and $\mathbf{o} \subseteq a$. $mayob_{\mathbf{O}}(U)$ holds if and only if there exist a, U_1, and U_2 such that $U \succeq O_a.U_1||U_2$ and $\mathbf{o} \subseteq a$.

Lemma 6.34: Let U be a usage (not an extended usage). $mayob^*_{\mathbf{O}}(U)$ ($mayob^*_{\mathbf{I}}(U)$, resp.) holds if and only if $mayob_{\mathbf{O}}(U)$ ($mayob_{\mathbf{I}}(U)$, resp.) holds.

Proof: We show only the case for $mayob^*_{\mathbf{O}}$. The case for $mayob^*_{\mathbf{I}}$ is similar.

⇒: The proof proceeds by induction on the structure of U.

- Case for U is 0, α, or $I_a.U'$: Since $mayob^*_{\mathbf{O}}(U)$ is **false**, the proposition is vacuously true.
- Case for $U = O_{a'}.U'$: If $mayob^*_{\mathbf{O}}(U)$ holds, then it must be the case that $\mathbf{o} \subseteq a'$. So, the required result holds for $a = a'$, $U_1 = U'$, and $U_2 = 0$.
- Case for $U = U'||U''$: Suppose $mayob^*_{\mathbf{O}}(U'||U'')$ holds. By the definition of $mayob^*_{\mathbf{O}}$, either $mayob^*_{\mathbf{O}}(U')$ or $mayob^*_{\mathbf{O}}(U'')$ holds. Suppose $mayob^*_{\mathbf{O}}(U')$ holds. Then, by induction hypothesis, there must exist a', U'_1, and U'_2 such that $U_1 \succeq O_{a'}.U'_1||U'_2$ and $\mathbf{o} \subseteq a'$. Therefore, the required result holds for $a = a'$, $U_1 = U'_1$, and $U_2 = U'_2||U''$. The case where $mayob^*_{\mathbf{O}}(U'')$ holds is similar.
- Case for $U = U' \sqcap U''$: Similar to the above case.
- Case for $U = \mathbf{rec}\,\alpha.U'$: Suppose $mayob^*_{\mathbf{O}}(U)$ holds. Then, by the definition of $mayob^*_{\mathbf{O}}$, $mayob^*_{\mathbf{O}}(U')$ also holds. By induction hypothesis, there must exist a', U'_1, and U'_2 such that $U' \succeq O_{a'}.U'_1||U'_2$. Since U is $\mathbf{rec}\,\alpha.U'$, we have $U \succeq [\alpha \mapsto U]U' \succeq O_{a'}.[\alpha \mapsto U]U'_1||[\alpha \mapsto U]U'_2$. The required result therefore holds for $a = a'$, $U_1 = [\alpha \mapsto U]U'_1$, and $U_2 = [\alpha \mapsto U]U'_2$.
- Case for $U = *U'$: Suppose $mayob^*_{\mathbf{O}}(U)$ holds. Then, by the definition of $mayob^*_{\mathbf{O}}$, $mayob^*_{\mathbf{O}}(U')$ also hold. By induction hypothesis, we have $mayob_{\mathbf{O}}(U')$, which implies $mayob_{\mathbf{O}}(*U)$.

⇐: Because $mayob^*_{\mathbf{O}}(O_a.U_1||U_2)$ holds, the result follows if we show that $U \succeq U'$ and $mayob^*_{\mathbf{O}}(U')$ imply $mayob^*_{\mathbf{O}}(U)$. We prove it by induction on derivation of $U \succeq U'$ with case analysis on the last rule used. Since the other cases are trivial or similar, we show only the case for the rule $\mathbf{rec}\,\alpha.U_1 \succeq [\alpha \mapsto \mathbf{rec}\,\alpha.U_1]U_1$. Suppose $mayob^*_{\mathbf{O}}([\alpha \mapsto \mathbf{rec}\,\alpha.U_1]U_1)$ holds but $mayob^*_{\mathbf{O}}(\mathbf{rec}\,\alpha.U_1)$ does not hold. Then, by Lemma 6.32, $mayob^*_{\mathbf{O}}(U_1)$ also holds. By the definition of $mayob^*_{\mathbf{O}}$, $mayob^*_{\mathbf{O}}(\mathbf{rec}\,\alpha.U_1)$ holds, hence a contradiction.

□

Lemma 6.35: Suppose that U is a usage and that θ is a substitution of closed usages for usage variables such that $FV(U) \subseteq dom(\theta)$. Then, $ob_{\mathbf{O}}(\theta U)$ imply $ob^*_{\mathbf{O}}(U, \theta)$.

Proof: The proof proceeds by induction on the structure of U.

- Case U is 0 or $I_a.U$: Vacuously true, since $ob_{\mathbf{O}}(\theta U)$ cannot hold.
- Case $U = \alpha$: Trivial, since $ob^*_{\mathbf{O}}(\alpha, \theta) = \mathbf{true}$ by the definition of $ob^*_{\mathbf{O}}$.
- Case $U = O_a.U'$: Suppose $ob_{\mathbf{O}}(\theta U)$ holds. Then, it must be the case that $\mathbf{o} \subseteq a$. By the definition of $ob^*_{\mathbf{O}}$, $ob^*_{\mathbf{O}}(U, \theta)$ holds.
- Case $U = U_1||U_2$: Suppose $ob_{\mathbf{O}}(\theta U)$ holds. Then, it must be the case that $ob_{\mathbf{O}}(\theta U_1)$ or $ob_{\mathbf{O}}(\theta U_2)$. By induction hypothesis, $ob^*_{\mathbf{O}}(U_1, \theta)$ or $ob^*_{\mathbf{O}}(U_2, \theta)$ must hold. So, we have $ob^*_{\mathbf{O}}(U, \theta)$ by the definition of $ob^*_{\mathbf{O}}$.
- Case $U = U_1 \sqcap U_2$: Suppose $ob_{\mathbf{O}}(\theta U)$ holds. Then, it must be the case that $ob_{\mathbf{O}}(\theta U_1)$ and $ob_{\mathbf{O}}(\theta U_2)$. By induction hypothesis, $ob^*_{\mathbf{O}}(U_1, \theta)$ and $ob^*_{\mathbf{O}}(U_2, \theta)$ must hold. So, we have $ob^*_{\mathbf{O}}(U, \theta)$ by the definition of $ob^*_{\mathbf{O}}$.
- Case $U = \mathbf{rec}\,\alpha.U'$: Suppose $ob_{\mathbf{O}}(\theta U)$ holds. Then, there must exist a, U_1, and U_2 such that $\theta U \succeq O_a.U_1||U_2$ and $\mathbf{o} \subseteq a$. By Lemma 6.34, $mayob^*_{\mathbf{O}}(\theta U)$ holds, which implies $mayob^*_{\mathbf{O}}(\theta U')$ also holds by the definition of $mayob^*_{\mathbf{O}}$. Also, by $ob_{\mathbf{O}}(\theta U)$ and $\theta U \succeq [\alpha \mapsto \theta U]\theta U' = (\theta \circ [\alpha \mapsto \theta U])U'$ (we can assume without loss of generality that $\alpha \notin dom(\theta)$), it must be the case that $ob_{\mathbf{O}}(\theta' U')$ for $\theta' = \theta \circ [\alpha \mapsto \theta U]$. Since θ' is a substitution of closed usages for usage variables and $dom(\theta') = dom(\theta) \cup \{\alpha\} \supseteq FV(U') = FV(U) \cup \{\alpha\}$, we have $ob^*_{\mathbf{O}}(U', \theta')$ by induction hypothesis. Therefore, $ob^*_{\mathbf{O}}(U, \theta)$ holds.
- Case for $U = *U'$: Suppose $ob_{\mathbf{O}}(\theta U)$ holds. Then, by Lemma 4.20, it must be the case that $ob_{\mathbf{O}}(\theta U')$. By induction hypothesis, we have $ob^*_{\mathbf{O}}(U', \theta)$, which implies $ob^*_{\mathbf{O}}(U, \theta)$.

□

Definition 6.36: $\mathcal{U}_{O_\mathbf{o}}$ is the least set closed under the following rules:

$$(FV(U) = \emptyset \wedge \exists \theta, U'.(U = \theta U' \wedge ob^*_{\mathbf{O}}(U', \theta) \wedge \forall \alpha \in FV(U').\theta\alpha \in \mathcal{U}_{O_\mathbf{o}})) \Rightarrow U \in \mathcal{U}_{O_\mathbf{o}}$$

Lemma 6.37: Let U_1 and U_2 are closed usages, i.e., $FV(U_1) = FV(U_2) = \emptyset$. If $U_1 \in \mathcal{U}_{O_\mathbf{o}}$ or $U_2 \in \mathcal{U}_{O_\mathbf{o}}$, then $U_1||U_2 \in \mathcal{U}_{O_\mathbf{o}}$.

Proof: If $U_1 \in \mathcal{U}_{O_\mathbf{o}}$, then U_1 is decomposed into $\theta U_1'$ such that $ob^*_{\mathbf{O}}(U_1', \theta)$ and $\theta\alpha \in \mathcal{U}_{O_\mathbf{o}}$ for each $\alpha \in FV(U_1')$. Then, $U_1||U_2$ can be decomposed into $\theta(U_1'||U_2)$. Moreover, $ob^*_{\mathbf{O}}(U_1'||U_2, \theta)$ holds by the definition of $ob^*_{\mathbf{O}}$. □

Lemma 6.38: If $U \in \mathcal{U}_{O_\mathbf{o}}$ and $U \succeq' U'$, then $U' \in \mathcal{U}_{O_\mathbf{o}}$.

Proof: The proof proceeds by induction on derivation of $U \succeq' U'$, with case analysis on the last rule used. Because the other cases are similar or trivial, we show only the cases for the rules for $\sqcap$, $\mathbf{rec}\,\alpha.$, and congruence on $||$.

- Case for the rule $U_1 \sqcap U_2 \succeq' U_1$: In this case, there exist θ and U_3 such that

$$\begin{array}{l}\theta U_3 = U_1 \sqcap U_2 \\ ob^*_{\mathbf{O}}(U_3, \theta) \\ \forall \alpha \in FV(U_3).(\theta\alpha \in \mathcal{U}_{O_\circ})\end{array}$$

  Without loss of generality, we can assume that U_3 is not a variable. (If U_3 is a variable β, we can decompose $\theta\beta$ into (U'_3, θ') by the construction of the set $\mathcal{U}_{O_\circ}$.) So, $U_3 = U'_1 \sqcap U'_2$ and $\theta U'_i = U_i$ for $i = 1, 2$. From $ob^*_{\mathbf{O}}(U_3, \theta)$, we get $ob^*_{\mathbf{O}}(U'_1, \theta)$ and $ob^*_{\mathbf{O}}(U'_2, \theta)$. So, $U_1 \in \mathcal{U}_{O_\circ}$ as required.

- Case for the rule $\mathbf{rec}\,\alpha.U_1 \succeq' [\alpha \mapsto \mathbf{rec}\,\alpha.U_1]U_1$: In this case, there exist θ, U_2 such that

$$\begin{array}{l}\theta U_2 = \mathbf{rec}\,\alpha.U_1 \\ ob^*_{\mathbf{O}}(U_2, \theta) \\ \forall \alpha \in FV(U_2).(\theta\alpha \in \mathcal{U}_{O_\circ})\end{array}$$

  Without loss of generality, we can assume that U_2 is not a variable. So, $U_2 = \mathbf{rec}\,\alpha.U'_1$ and $\theta U'_1 = U_1$. From $ob^*_{\mathbf{O}}(U_2, \theta)$, we get $ob^*_{\mathbf{O}}(U'_1, \theta \circ [\alpha \mapsto \theta'\mathbf{rec}\,\alpha.U'_1])$. Because $\theta'\mathbf{rec}\,\alpha.U'_1 = \mathbf{rec}\,\alpha.U_1 \in \mathcal{U}_{O_\circ}$ and $(\theta \circ [\alpha \mapsto \theta'\mathbf{rec}\,\alpha.U'_1])U'_1 = [\alpha \mapsto \mathbf{rec}\,\alpha.U_1]U_1$, we have $[\alpha \mapsto \mathbf{rec}\,\alpha.U_1]U_1 \in \mathcal{U}_{O_\circ}$ as required.

- Case for the rule:

$$\frac{U_1 \succeq' U'_1}{U_1||U_2 \succeq' U'_1||U_2}$$

  In this case, there exist θ, U_3 such that

$$\begin{array}{l}\theta U_3 = U_1||U_2 \\ ob^*_{\mathbf{O}}(U_3, \theta) \\ \forall \alpha \in FV(U_3).(\theta\alpha \in \mathcal{U}_{O_\circ})\end{array}$$

  Without loss of generality, we can assume that U_3 is not a variable. So, $U_3 = U'_1||U'_2$ and $\theta U'_i = U_i$ for $i = 1, 2$. By the definition of $ob^*_{\mathbf{O}}$, it must be the case that $ob^*_{\mathbf{O}}(U'_i, \theta)$ for either $i = 1$ or 2. So, either $U_1 \in \mathcal{U}_{O_\circ}$ or $U_2 \in \mathcal{U}_{O_\circ}$. If $U_2 \in \mathcal{U}_{O_\circ}$, then the result follows immediately from Lemma 6.37. If $U_1 \in \mathcal{U}_{O_\circ}$, then we have $U'_1 \in \mathcal{U}_{O_\circ}$ by induction hypothesis, from which the result follows by Lemma 6.37.

□

Lemma 6.39: If $U \in \mathcal{U}_{O_\circ}$, then $mayob^*_{\mathbf{O}}(U)$.

Proof: The proof proceeds by induction on the structure of U.

- Case for $U = \alpha$: This case cannot happen, since $FV(U) = \emptyset$.

- Cases for $U = 0$ or $I_a.U_1$: This case cannot happen either. Suppose $U \in \mathcal{U}_{O_\circ}$. Then, it must be the case that U can be decomposed into $\theta U'$ such that $ob^*_{\mathbf{O}}(U', \theta)$ and U' is not a variable. This cannot be the case, however, by the definition of $ob^*_{\mathbf{O}}$.

- Case for $U = O_a.U_1$: Suppose $U \in \mathcal{U}_{O_\mathbf{o}}$. Then, there must exist θ, U' such that

$$\begin{array}{l}\theta U' = O_a.U_1 \\ ob^*_{\mathbf{O}}(U', \theta) \\ \forall \alpha \in FV(U').(\theta\alpha \in \mathcal{U}_{O_\mathbf{o}})\end{array}$$

  Without loss of generality, we can assume that $U' = O_a.U'_1$ and $\theta U'_1 = U_1$. From $ob^*_{\mathbf{O}}(U', \theta)$, we get $\mathbf{o} \subseteq a$. So, $mayob^*_{\mathbf{O}}(U)$ holds.

- Case for $U = U_1||U_2$: Suppose $U \in \mathcal{U}_{O_\mathbf{o}}$. Then, there must exist θ, U' such that

$$\begin{array}{l}\theta U' = U_1||U_2 \\ ob^*_{\mathbf{O}}(U', \theta) \\ \forall \alpha \in FV(U').(\theta\alpha \in \mathcal{U}_{O_\mathbf{o}})\end{array}$$

  Without loss of generality, we can assume that $U' = U'_1||U'_2$ and $\theta U'_i = U_i$ for $i = 1, 2$. From $ob^*_{\mathbf{O}}(U', \theta)$, we get $ob^*_{\mathbf{O}}(U'_1, \theta)$ or $ob^*_{\mathbf{O}}(U'_2, \theta)$, which implies $U_1 \in \mathcal{U}_{O_\mathbf{o}}$ or $U_2 \in \mathcal{U}_{O_\mathbf{o}}$. By induction hypothesis, we have $mayob^*_{\mathbf{O}}(U_1)$ or $mayob^*_{\mathbf{O}}(U_2)$, which implies $mayob^*_{\mathbf{O}}(U)$ as required.

- Case for $U = U_1 \sqcap U_2$ or $U = *U_1$: Similar to the above case.

- Case for $U = \mathbf{rec}\,\alpha.U_1$: Suppose $U \in \mathcal{U}_{O_\mathbf{o}}$. Then, there must exist θ, U' such that

$$\begin{array}{l}\theta U' = \mathbf{rec}\,\alpha.U_1 \\ ob^*_{\mathbf{O}}(U', \theta) \\ \forall \alpha \in FV(U').(\theta\alpha \in \mathcal{U}_{O_\mathbf{o}})\end{array}$$

  Without loss of generality, we can assume that $U' = \mathbf{rec}\,\alpha.U'_1$ and $U_1 = \theta U'_1$. From $ob^*_{\mathbf{O}}(U', \theta)$, we get $mayob^*_{\mathbf{O}}(\theta U'_1)$, which implies $mayob^*_{\mathbf{O}}(\mathbf{rec}\,\alpha.\theta U'_1)$. The result follows, since $U = \theta U' = \mathbf{rec}\,\alpha.\theta U'_1$.

□

Proof of Theorem 6.31: We show only the case for $ob_{\mathbf{O}}$. The case for $ob_{\mathbf{I}}$ is similar.

⇒: A special case of Lemma 6.35.

⇐: Suppose $ob^*_{\mathbf{O}}(U, id)$ and $U \succeq' U'$. Then, by the definition of $\mathcal{U}_{O_\mathbf{o}}$, $U \in \mathcal{U}_{O_\mathbf{o}}$. By $U \succeq' U'$ and Lemma 6.38, $U' \in \mathcal{U}_{O_\mathbf{o}}$. Lemma 6.39 implies $mayob^*_{\mathbf{O}}(U')$. Therefore, there exists a, U_1, U_2 such that $U' \succeq O_a.U_1||U_2$ and $\mathbf{o} \subseteq a$ as required.

□

6.2.6 Checking the satisfiability of the remaining constraints

Now, we can assume that we have only constraints of the form c_a below:

$$\begin{array}{l}c_a ::= c_L \Rightarrow c_R \\ c_L ::= c_{atom} \mid c_L \wedge c_L \mid c_L \vee c_L \mid \mathbf{true} \mid \mathbf{false} \\ c_R ::= c_t \mid c_{atom} \mid c_R \wedge c_R \mid c_R \vee c_R \mid \mathbf{true} \mid \mathbf{false} \\ c_t ::= t_1 \mathcal{T} t_2 \\ c_{atom} ::= \mathbf{c} \subseteq a \mid \mathbf{o} \subseteq a \mid ob_{\mathbf{O}}(\alpha) \mid ob_{\mathbf{I}}(\alpha)\end{array}$$

Note that $\mathbf{c} \subseteq a$ is represented in the form $\mathbf{true} \Rightarrow \mathbf{c} \subseteq a$. The satisfiability of a set of constraints of the form c_a is clearly decidable. Note that we have only a finite set of attribute variables and each ranges over a finite set $\{\emptyset, \mathbf{c}, \mathbf{o}, \mathbf{co}\}$. Note also that since all the constraints on usage variables are of the form $ob_{\mathbf{O}}(\alpha)$ or $ob_{\mathbf{I}}(\alpha)$, we need to assign to each usage variable only a usage in $\{0, I_{\mathbf{o}}.0, O_{\mathbf{o}}.0, I_{\mathbf{o}}.0||O_{\mathbf{o}}.0\}$ and check the satisfiability. So, we can try all the possible assignments to attribute variables and usage variables. We can decide the satisfiability more efficiently as follows.

1. Replace $ob_{\mathbf{O}}(\alpha)$ and $ob_{\mathbf{I}}(\alpha)$ with variables ob_α and ob'_α respectively.
2. Assign $\emptyset$ to every attribute variable and assign **false** to ob_α and ob'_α.
3. If there is a constraint $c_L \Rightarrow c_R$ such that c_L is true, replace $c_L \Rightarrow c_R$ with $\mathbf{true} \Rightarrow c_R$ and increase the assignment to variables so that obligation constraints and attribute constraints in c_R are satisfied. If such increase is impossible, fail. Repeat this step.
4. If there is no more constraint $c_L \Rightarrow c_R$ such that c_L is true but some attribute constraints in c_R are not satisfied, then extract all the constraints of the form $t_1 \mathcal{T} t_2$ from each constraint $\mathbf{true} \Rightarrow c_R$, and check whether there is a strict partial order $\mathcal{T}$ satisfying them (by using, for example, a graph algorithm for cycle detection).

An important fact used in the above algorithm is that both c_L and c_R are monotonic on the variables in c_L and c_R. So, once c_L and c_R are satisfied, they remain true afterwards. The above algorithm actually may not be efficient because the third step may involve backtracking: if c_R is $\mathbf{o} \subseteq a_1 \vee \mathbf{o} \subseteq a_2$ (which is generated by the reduction of a reliability constraint), then we can either increase a_1 or a_2. If the above algorithm turns out to be inefficient in practice, we can strengthen the reliability constraint by adding the following extra condition to Definition 4.6:

3 If $U \succeq O_{a_1}.U_1||I_{a_2}.U_2||U_3$ for some U_1, U_2 and U_3, then (i)if $\mathbf{c} \subseteq a_1$ then $\mathbf{o} \subseteq a_2$ and (ii)if $\mathbf{c} \subseteq a_2$ then $\mathbf{o} \subseteq a_1$.

Then, we can reduce usage constraints so that $\vee$ does not appear in c_R. (Details are omitted.)

6.2.7 How to compute the set $Deriv(U, C)$

Given a usage U and $C = \{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$, we construct a Petri net that has a place p_V for each element V of $SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n)$. Then, each usage consisting of elements of $SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n)$ can be mapped to a state of the Petri net as follows.

$$
\begin{aligned}
&UtoP(0) = \emptyset \\
&UtoP(\alpha) = \{p_\alpha \mapsto 1\} \\
&UtoP(I_a.V) = \{p_{I_a.V} \mapsto 1\} \\
&UtoP(O_a.V) = \{p_{O_a.V} \mapsto 1\} \\
&UtoP(V_1||V_2) = UtoP(V_1) + UtoP(V_2) \\
&UtoP(V_1 \sqcap V_2) = \{p_{V_1 \sqcap V_2} \mapsto 1\} \\
&UtoP(\mathbf{rec}\,\alpha.V) = \{p_{\mathbf{rec}\,\alpha.V} \mapsto 1\} \\
&UtoP(*V) = \{p_{*V} \mapsto 1\}
\end{aligned}
$$

We introduce the following transitions into the above Petri net.

- A transition:
$$\{p_{I_a.V_1} \mapsto 1, p_{O_a.V_2} \mapsto 1\} \longrightarrow UtoP(V_1||V_2)$$
for each pair of $p_{I_a.V_1}$ and $p_{O_{a'}.V_2}$.

- Transitions:
$$\{p_{V_1 \sqcap V_2} \mapsto 1\} \longrightarrow UtoP(V_1)$$
$$\{p_{V_1 \sqcap V_2} \mapsto 1\} \longrightarrow UtoP(V_2)$$
for each $p_{V_1 \sqcap V_2}$.

- A transition
$$\{p_{\mathbf{rec}\,\alpha.V} \mapsto 1\} \longrightarrow UtoP([\alpha \mapsto \mathbf{rec}\,\alpha.V]V)$$
for each $p_{\mathbf{rec}\,\alpha.V}$.

- A transition
$$\{p_{*V} \mapsto 1\} \longrightarrow UtoP(*V||V)$$
for each p_{*V}.

- A transition
$$\{p_{\alpha_i} \mapsto 1\} \longrightarrow UtoP(U_i)$$
for each $\alpha_i \leq U_i \in C$.

Then, it is trivial that $U \succeq_C (\longrightarrow \succeq_C)^* U'$ if and only if $UtoP(U')$ is reachable from the initial state $UtoP(U)$ by the above Petri net.

Let $\{V_1, \ldots, V_n\}$ be a subset of $SubUExp(U) \cup SubUExp(U_1) \cup \cdots \cup SubUExp(U_n)$. Then, $\{V_1, \ldots, V_n\} \in Deriv(U, C)$ if and only if $UtoP(V_1|| \cdots ||V_n)$ is reachable from $UtoP(U)$ by the Petri net obtained by adding the following transitions to the above Petri net.

$$\{p_{V_i} \mapsto 1\} \longrightarrow \emptyset \text{ (for each } i \in \{1, \ldots, n\})$$

Because the reachability problem of Petri nets is decidable [4], we can compute the set $Deriv(U, C)$.

In the current prototype implementation of our type-inference algorithm, however, we approximate the above reachability problem by a integer linear programming problem.

7 Extensions

This section discusses extensions of our type system.

7.1 Extension of time tags and tag ordering

Although the treatment of time tags and tag orderings greatly affects the expressive power of the deadlock-free process calculus, it is very naive in this paper. For example, all channels passed through the same channel are forced to have exactly the same time tag. As a result, the calculus presented in this paper does not have enough expressive power. (For processes without annotations on capabilities/obligations, it has almost the same expressive power as the ordinary simply-typed π-calculus [5] except that ours have no recursive types, but many useful processes annotated with capabilities/obligation cannot be typed.) Fortunately, however, it is easy to replace it with the more sophisticated treatment in the previous deadlock-free typed

calculi [10, 23]. Because an algorithm for automatically inferring time tags and tag orderings has already been developed, there would be no problem in integrating it with the type reconstruction algorithm developed in this paper. The resulting deadlock-free process calculus would become more expressive than the previous calculi [10, 23], which were already shown to be expressive enough to encode the simply-typed λ-calculus with various evaluation strategies, semaphores, and typical concurrent objects.

Because the treatment of time tags and tag orderings in the previous works [10, 23] was complex and ad hoc, we plan to generalize and clarify it before integrating it with the implicitly-typed calculus of this paper.

7.2 Partial type/usage annotations

We have dealt with an implicitly-typed calculus in this paper, and instead, we provided a way for annotating each input/output process with capabilities/obligations. However, programmers may still want to explicitly annotate some parts of their programs with types and/or usages. For example, one may write a functional process in the following way in order to ensure that the process really behaves like a function:

$$*f?^{\mathbf{o}}[x, r : O_{\mathbf{o}}.0]. \cdots$$

Partial type/usage annotations like above do not require much modification to the entire structure of our type reconstruction algorithm, but we need to develop an algorithm that inputs usage constants (i.e., usages not containing free usage variables) U_1 and U_2 and decides whether $U_1 \leq U_2$ holds or not. We have not yet checked whether $U_1 \leq U_2$ is decidable or not and if so, whether there is an efficient decision algorithm. However, even if there doesn't exist an efficient decision algorithm that works for an arbitrary input, we will be able to restrict the syntax of usage/type annotations appropriately, so that the decision of $U_1 \leq U_2$ can be made efficiently in practice.

7.3 Structural subtyping

It is also possible to introduce structural subtyping, following Pierce and Sangiorgi's subtyping based on input-only/output-only channel types [16]. For example, $[int]^t/I_a.0$ can be considered a subtype of $[real]^t/I_a.0$, and $[real]^t/O_a.0$ can be considered a subtype of $[int]^t/O_a.0$, provided that *int* is a subtype of *real*. For another example, the type $[[]^{t_1}/(I_{\mathbf{c}}.I_{\mathbf{c}}.0)]^{t_2}/O_a.0$ can be considered a subtype of $[[]^{t_1}/(I_{\mathbf{c}}.0||I_{\mathbf{c}}.0)]^{t_2}/O_a.0$. In general, we can introduce the following rule for subtyping:

$$\frac{\begin{array}{c}(U' \text{ contains } O) \Rightarrow (\tau_1' \leq \tau_1 \wedge \cdots \wedge \tau_n' \leq \tau_n) \\ (U' \text{ contains } I) \Rightarrow (\tau_1 \leq \tau_1' \wedge \cdots \wedge \tau_n \leq \tau_n') \\ U \leq U'\end{array}}{[\tau_1, \ldots, \tau_n]^t/U \leq [\tau_1', \ldots, \tau_n']^t/U'}$$

We have not checked details, but we think that a type reconstruction algorithm can be developed in a manner similar to Igarashi and Kobayashi's development of a type reconstruction algorithm for the linear π-calculus with I/O subtyping [8].

7.4 Unreliable channels

In Kobayashi's original deadlock-free process calculus [10], deadlock-freedom was not guaranteed for channels called *unreliable channels*; so, even a well-typed process may deadlock. However, if a deadlock occurs, some sub-process must be trying to communicate on an unreliable channel, causing the deadlock. (That is why

the original calculus was called "partially" deadlock-free). We can introduce unreliable channels into the type system of this paper, by not requiring the condition $rel(\tau)$ in the rule (T-New) for the creation of unreliable channels.

7.5 Other process constructors

It is possible to extend our type system and type reconstruction algorithm to deal with other constructors of the π-calculus, such as choice $P_1 + P_2$ and matching [14]. For example, a simple way to deal with the choice is to introduce the following typing rule, which is similar to the rule (T-If) for conditionals:

$$\frac{\Gamma; \mathcal{T} \vdash P_1 \qquad \Gamma; \mathcal{T} \vdash P_2}{\Gamma; \mathcal{T} \vdash P_1 + P_2} \qquad \text{(T-Choice-Naive)}$$

However, it may be useful to treat the choice more carefully. For example, consider the process $P = P_1 + P_2$, where $P_1 = x?[\,].\,\mathbf{0}$ and $P_2 = x![\,].\,\mathbf{0}$. P_1 and P_2 fulfill the obligations to input and output on x respectively. Indeed, the following judgment is valid in our type system.

$$x : [\,]^t / I_{\mathbf{o}}.0; \emptyset \vdash P_1$$
$$x : [\,]^t / O_{\mathbf{o}}.0; \emptyset \vdash P_2$$

However, with the above rule (T-Choice-Naive), we can obtain only the following judgment for P:

$$x : [\,]^t / (I_{\mathbf{o}}.0 \sqcap O_{\mathbf{o}}.0); \emptyset \vdash P$$

The type environment means that neither input nor output is guaranteed to be performed by P, because a usage $U_1 \sqcap U_2$ means that the channel may be used according to either U_1 or U_2 indeterministically. Actually, however, P fulfills both obligations on input and output. In fact, both $P \mid x?^{\mathbf{c}}[\,].\,\mathbf{0}$ and $P \mid x!^{\mathbf{c}}[\,]\mathbf{0}$ can be reduced.

We can make the process $P \mid x?^{\mathbf{c}}[\,].\,\mathbf{0}$ well-typed by introducing another usage constructor $\sqcup$. $U_1 \sqcup U_2$ means that the channel can be used according to either U_1 or U_2 as in the case for $U_1 \sqcap U_2$, but the choice can be made by external processes. Readers who are familiar with linear logic [6] may notice that $\sqcap$ corresponds to the additive disjunction $\oplus$, while $\sqcup$ corresponds to the additive conjunction &. The distinction between these two kinds of choice have already been made by Takeuchi, Honda, and Kubo [24] (although their type system does not guarantee the deadlock-freedom in the sense of ours). This distinction will be particularly useful when our type system is extended to deal with object types as described below in Section 7.7.

7.6 Extending channel types

Channel types in this paper force each channel to be used for passing values of the same type throughout its lifetime. It may sometimes be too restrictive. For example, if a channel is used according to $I_{a_1}.I_{a_2}.\mathbf{0} || O_{a_3}.O_{a_4}.\mathbf{0}$, then the first input never interacts with the second output, and therefore it does not cause an error even if an integer is sent first and then a string is sent along the channel. We can allow such uses of a channel by changing the syntax of types as follows:

$$\tau ::= [\tau_1, \ldots, \tau_n]_{I_a}.\tau \mid [\tau_1, \ldots, \tau_n]_{O_a}.\tau \mid (\tau_1 || \tau_2) \mid \cdots$$

Here, $[\tau_1, \ldots, \tau_n]_{I_a}.\tau$ ($[\tilde{\tau}]_{O_a}.\tau$, resp.) expresses the type of a channel that is first used once for receiving (sending, resp.) a tuple of values of types $\tau_1, \ldots, \tau_n$, and then used as a channel of type τ. $\tau_1 || \tau_2$ expresses

the type of a channel that is used as a channel of type τ_1 and, in parallel to that, used as a channel of type τ_2. So, usages and types are now integrated, and the operations on usages become those on types. For example, the type of a channel mentioned above can be expressed as $[int]_{I_{a_1}}.[string]_{I_{a_2}}|||[int]_{O_{a_1}}.[string]_{O_{a_2}}$. This kind of channel type (which allows different types of values to be passed through the same channel) has already appeared in other types systems such as the above-mentioned Takeuchi, Honda, and Kubo's type system [24] and Yoshida's graph types [26] (although combinations with capability/obligation attributes are new).

Some type expressions like $[int]_{I_a}|||[bool]_{O_a}$ (which means that a channel is used for sending a boolean value and for receiving an integer) are of course invalid. They can be excluded out by imposing appropriate consistency conditions, such as "For each subexpression $[\tau_1, \ldots, \tau_n]_{I_a}$, $\tau'_1 \leq \tau_1, \ldots, \tau'_n \leq \tau_n$ must hold for every subexpression $[\tau'_1, \ldots, \tau'_n]_{O_a}$ that may be used in parallel."

7.7 Object types

We can encode typical concurrent objects into our deadlock-free process calculus or its extensions mentioned above, as demonstrated in [10]. However, it may be useful to extend the type system to handle concurrent objects and their types directly.

One way to encode concurrent objects in the π-calculus is to express a concurrent object as a pair of (1) a channel to store the current state and (2) a set of processes each of which executes each method [10, 17]. An object identity is expressed as a record of channels for receiving requests. For example, a counter object with a method **inc** to increment the counter and a method **read** to read the counter is expressed as the following process:

$$\begin{array}{l}(\nu state)\,(\\ \quad state![1]\\ \quad \mid *inc?[reply].\, state?^{\mathbf{c}}[n].\, (reply![]\mid state![n+1])\\ \quad \mid *read?[reply].\, state?^{\mathbf{c}}[n].\, (reply![n]\mid state![n])\end{array}$$

The channel $state$ holds the counter value, and the processes $*inc?[reply].\,\cdots$ and $*read?[reply].\,\cdots$ respectively executes the methods **inc** and **read**. The identity of a counter object is expressed as a record $\{\mathbf{inc} = inc, \mathbf{reply} = reply\}$ ($\{$**inc** and **reply** are field names, and inc and $reply$ are the corresponding field values). The type of a counter object is therefore represented by a record type $\{\mathbf{inc} : [[\,]^{t_1}/O_{\mathbf{o}}.0]^{t_2}/*O_{\mathbf{c}}.0, \mathbf{read} : [[int]^{t_3}/O_{\mathbf{o}}.0]^{t_4}/*O_{\mathbf{c}}.0\}$ (see [10] for more details). The type means that method requests are eventually accepted (which follows from the usage $*O_{\mathbf{c}}.0$) and that a reply to each request is also eventually sent (which follows from the usage $O_{\mathbf{o}}.0$).

In the above encoding of concurrent objects, the order in which method requests are accepted can be captured only indirectly through time tags. This is problematic for certain objects like one-place buffer objects, which can execute a put method and a get method only alternately. We can express the behavior of such objects by introducing object types as primitives, instead of encoding them in terms of record types and channel types. For example, object types can be defined as the following extension of the channel types in Section 7.6:

$$\tau ::= M[\tau_1, \ldots, \tau_n]_{I_a}.\tau \mid M[\tau_1, \ldots, \tau_n]_{O_a}.\tau \mid (\tau_1||\tau_2) \mid \cdots$$

Here, M denotes a method name. $M[\tau_1, \ldots, \tau_n]_{I_a}.\tau$ expresses the type of a concurrent object which accepts a request for the method M and then behaves as specified by type τ. $M[\tau_1, \ldots, \tau_n]_{O_a}.\tau$ expresses the type of a concurrent object for which some process sends a request for the method M and then behaves as specified by type τ. The behavior of a one-place buffer object for storing an integer can be expressed by a recursive type τ such that $\tau = Put[int]_{I_{\mathbf{o}}}.Get[int]_{I_{\mathbf{o}}}.\tau$. From a user of the object, the object can be viewed as having

type τ' such that $\tau' = Put[int]_{O_c}.Get[int]_{O_c}.\tau'$. Concrete definitions of a deadlock-free concurrent object calculus and its type system based on the above idea are left for future work.

7.8 Detecting Livelock

Our type system cannot guarantee freedom from livelocks, i.e., situations in which a process diverges without fulfilling obligations or with keeping a subprocess that is trying to use a capability waiting forever. Guaranteeing freedom from livelocks corresponds to ensuring termination in functional languages. It is therefore unrealistic to develop a type system guaranteeing livelock-freedom for a general purpose concurrent programming language: It would restrict the expressive power too much or impose too heavy a burden on the programmer by requiring explicit annotations of complex types. However, for some special purposes for which less expressive power is necessary or livelock-freedom is extremely important, it may be useful to guarantee freedom from livelock by restricting recursive processes to primitive recursive ones (analogues of primitive recursive functions), or by requiring complex type annotations.

One compromise applicable to general purpose languages would be to guarantee livelock-freedom only for certain channels. For that purpose, for example, we can introduce a constructor $\bigcirc$ for usages, which expresses the passage of time. For example, $\bigcirc O_{\mathbf{o}}.\mathbf{0}$ means that an output must be performed only after the passage of some time. Then, we can infer that the output obligation in $\mathbf{rec}\,\alpha.(\alpha \sqcap O_{\mathbf{o}}.\mathbf{0})$ is eventually fulfilled even in a divergent process (under some fair scheduling), but that the output obligation in $\mathbf{rec}\,\alpha.(\bigcirc\alpha \sqcap O_{\mathbf{o}}.\mathbf{0})$ may not.

8 Related Work

A number of advanced type systems have recently been proposed for process calculi or concurrent object calculi. However, we are not aware of any other advanced typed process calculi that are equipped with type reconstruction algorithms, except for Igarashi and Kobayashi's type reconstruction algorithm [8] for the linear π-calculus [11]. We discuss Igarashi and Kobayashi's work in Section 8.1.

In addition, we compare our deadlock-free process calculus with other typed concurrent process/object calculi in Section 8.2. We have already discussed some of them in the previous papers [10, 23], but thanks to the generalization of our type system in this paper, it became easier to compare them with our type system.

8.1 Type reconstruction for the linear pi-calculus

Our type reconstruction algorithm can be considered an extension of Igarashi and Kobayashi's type reconstruction algorithm [8] for the linear π-calculus.

In Igarashi and Kobayashi's type system [8], channel types are of the form $[\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)}$. κ_1 and κ_2 range over the set of elements $0, 1, \omega$ called *uses* and indicate how many times (zero times, once, or an arbitrary number of times) channels may be used for input and output respectively. As in this paper, the operations $+, \sqcap$ and a relation $\leq$ on types and uses are introduced. For example, $+$ is defined by:

$$[\tau_1, \ldots, \tau_n]^{(\kappa_{11},\kappa_{12})} + [\tau_1, \ldots, \tau_n]^{(\kappa_{21},\kappa_{22})} = [\tau_1, \ldots, \tau_n]^{(\kappa_{11}+\kappa_{21},\kappa_{12}+\kappa_{22})}$$

$$\kappa_1 + \kappa_2 = \begin{cases} \kappa_2 & \text{if } \kappa_1 = 0 \\ \kappa_1 & \text{if } \kappa_2 = 0 \\ \omega & \text{otherwise} \end{cases}$$

So, a pair (κ_1, κ_2) corresponds to a usage in this paper. (Formally, we might be able to show the correspondence through the existence of a homomorphism from the algebra of usages to the algebra of pairs of

uses.) Type reconstruction proceeds in a manner similar to our type reconstruction: a principal typing is first obtained as a pair of a type environment containing variables and a set of constraints on the variables, and then the constraint set is reduced step by step.

Igarashi and Kobayashi's algorithm can also deal with Pierce and Sangiorgi's structural subtyping based on input-only/output-only channel types [16]. So, as mentioned in Section 7, we expect that we can extend our type reconstruction algorithm also to handle structural subtyping.

8.2 Other related type systems

We compare below other related type systems for process or concurrent object calculi. Type reconstruction algorithms for them have not been developed to our knowledge, and therefore, they are not directly related to the aim of this paper. However, it would be useful to clarify relationships between them and our type system in the terminology of usages, capabilities, obligations, etc., because most of them have been independently developed and have not been compared enough in a meaningful way. Clear connections would help development of type reconstruction algorithms for those type systems and also integration of them.

Yoshida's graph types [26] Yoshida [26] proposed a type system that can guarantee deadlock-freedom in a sense slightly different from ours. In her type system, a channel type is a directed graph, whose vertices express information on each use of the channel (whether it is used for input or output and what kind of value is passed) and whose arrows express the order of the uses. For example, the graph $\downarrow [x, y] \rightarrow x \downarrow [] \rightarrow y \uparrow []$ ($\rightarrow$ represents an arrow of the graph; $\downarrow$ and $\uparrow$ are parts of node labels) expresses the type of a channel that is used for receiving a pair of channels, receiving a null tuple through the first channel and then sending a null tuple through the second channel. There are some similarities between her type system and ours. A vertex corresponds to the part $[\tau_1, \ldots, \tau_n]$ and the part I/O of a channel type in our type system, and an edge corresponds to a tag ordering and the sequencing operator "." on usages in our type system. Two processes accessing the same channel can be composed only if their uses of the channel are represented by mutually complementary graphs (in the sense that one is obtained from the other by exchanging $\uparrow$ of each vertex with $\downarrow$). This corresponds to the reliability and the relation $\sim$ in our type system.

In our terminology, differences can be explained as follows. Her type system has only two classes of channels: one is the class of channels of the usage $*I_\emptyset.0 \,|\, *O_\emptyset.0$, which can be used in an arbitrary manner but for which neither capabilities nor obligations are guaranteed, and the other is the class of channels whose usages are always annotated with the attribute **co**. No subusage relation like ours is allowed for the usages of the latter class of channels. Also, unlike our tag ordering, the arrow of her graph types expresses the exact order of channel uses: if there is an arrow from τ_1 to τ_2, then a channel can *and must* be used according to types τ_1 and τ_2 in this order. (This can both be an advantage and a disadvantage: see discussions in [10].) Another difference is that her type system allows a channel to be used for passing values of different types, as discussed in Section 7.6.

To summarize, Yoshida's type system can be roughly seen as our type system, plus the extended channel types in Section 7.6, minus the capability-only/obligation-only attributes and the subusage relation.

Takeuchi, Honda, and Kubo's type system [24] Some similar ideas are also found in an earlier work by Takeuchi, Honda and Kubo [24] on a type system for a concurrent object-oriented language. Similarly to our usages and Yoshida's graph types [26], channel types are composed by a sequencing operator and choice operators. Each channel can be shared by two processes, and it is guaranteed by the type system that the two processes use the channel in a complementary manner. For example, if one uses the channel for sending an integer and then for receiving a string, then the other uses it for receiving an integer and

then for sending a string. Deadlock-freedom is not guaranteed, however, because the type system does not take into account in which order multiple channels are used.

In short, their type system can be roughly seen as our type system, *plus* unreliable channel types in Section 7.4, the extended channel types in Section 7.6, and the object types in Section 7.7, *minus* the empty/capability-only/obligation-only attributes, the tag orderings, and the subusage relation.

Sangiorgi's receptive names [22] Sangiorgi [22] also introduced a type system of the π-calculus which can guarantee deadlock-freedom in a certain sense. It guarantees that a process is always waiting to input on certain channels (called *receptive names*), so that outputs on those channels always succeed. Roughly speaking, receptive names correspond to channels of the usage $I_{\mathbf{o}}.0||O_{\mathbf{c}}.0$ or $*I_{\mathbf{o}}.0||*O_{\mathbf{c}}.0$. (Actually, he introduced two versions of receptive names: linear receptive names ω-receptive names. The former corresponds to the usage $I_{\mathbf{o}}.0||O_{\mathbf{c}}.0$ and the latter to $*I_{\mathbf{o}}.0||*O_{\mathbf{c}}.0$.)

There is something that cannot be captured by our type system. Consider the process $(\nu x)\, a![x].*x?[y].\, P$. In his type system, x is judged to be a (ω-)receptive name even if a is not a receptive name. This is because if the output on a succeeds, $*x?[y].\, P$ is executed, and therefore, every process that has received x through a can successfully performs an output on x, while if the output on a does not succeed, x cannot be used for output at all. However, in our type system, a must also be a receptive name: According to the rule (T-Out) in our type system, the output on a must also be guaranteed to succeed. In order to solve this problem, we can add another rule for output:

$$\frac{\begin{array}{c}\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U; \mathcal{T} \vdash P \\ a \subseteq a' \qquad v_i \in \{\mathit{true}, \mathit{false}\} \Rightarrow \tau_i = \mathit{bool} \text{ for each } i \in \{1, \ldots, n\}\end{array}}{x : [\tau_1, \ldots, \tau_n]^t/O_{a'}.U + ((v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma) \sqcap \emptyset); \mathcal{T} \vdash x!^a[v_1, \ldots, v_n].\, P} \qquad \text{(T-Out2)}$$

In the above rule, the condition $ob(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma) \Rightarrow \mathbf{c} \subseteq a'$ of (T-Out) is removed; instead, the type environment $v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma$ is weakened to $(v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma) \sqcap \emptyset$, reflecting the fact that $v_1, \ldots, v_n$ are not used at all and the body P is not executed unless the output on x succeeds. With this kind of modification, we think that our type system can subsume Sangiorgi's type system for receptive names [22].

To summarize, Sangiorgi's type system [22] can be roughly seen as our type system plus the above extension, minus usages except for $*I_\emptyset.0||!O_\emptyset.0$, $I_{\mathbf{o}}.0||O_{\mathbf{c}}.0$, and $*I_{\mathbf{o}}.0||*O_{\mathbf{c}}.0$, and the subusage relation.

Nierstrasz's regular types [15] An idea similar to our usages is also found in an earlier work by Nierstrasz on regular types [15]. In order to express changing behaviors of a concurrent object, he introduced types (called regular types) composed of sequencing, choice (like $\sqcup$ in Section 7.5), and recursion operators, and formalized a subtyping relation on those types. Nierstrasz [15] has not presented a type system, but his work can be basically viewed as our type system plus object types in Section 7.7, minus usage attributes, usage constructors $O_a.$, $|$, and $\sqcap$, and the tag orderings.

Puntigam's type system for concurrent objects [19] Puntigam [19] extends Nierstrasz's work [15] above by giving a type system that can ensure that a concurrent object really provides services specified by types (similar to regular types) and that users of a concurrent object do not send requests for unavailable services. In our terminology, every output operation always has a capability attribute, while every input operation has an obligation attribute. The syntax of object types is rather different from that of Nierstrasz's regular types, but we think that Puntigam's object types essentially express information almost equivalent to the regular types plus a parallel composition operator (like $||$ of ours). The underlying calculus is different

from ours in that the message arrival order is FIFO (messages are accepted by an object in the order they are sent) as in the actor model [7]. We think, however, that this is not an essential difference of the type systems. In fact, we can introduce the typing rule for an asynchronous, FIFO output operation "$x![v_1, \ldots, v_n]; P$" as a mixture of the rule for an asynchronous output $x![v_1, \ldots, v_n] \,|\, P$ and that for a synchronous output $x![v_1, \ldots, v_n].\, P$:

$$\frac{\begin{array}{c}\Gamma, x : [\tau_1, \ldots, \tau_n]^t/U; \mathcal{T} \vdash P \\ a \subseteq a' \qquad t\mathcal{T}(v_1 : \tau_1 + \cdots + v_n : \tau_n) \qquad ob(v_1 : \tau_1 + \cdots + v_n : \tau_n) \Rightarrow \mathbf{c} \subseteq a' \\ v_i \in \{true, false\} \Rightarrow \tau_i = bool \text{ for each } i \in \{1, \ldots, n\}\end{array}}{x : [\tau_1, \ldots, \tau_n]^t/O_{a'}.U + v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma; \mathcal{T} \vdash x!^a[v_1, \ldots, v_n]; P} \quad \text{(T-Out-FIFO)}$$

The difference from the case for a synchronous output is that the body P can be executed irrespectively of whether or not the output succeeds, and therefore we need not require that the output has a capability attribute even if $ob(\Gamma)$ holds.

To summarize, Puntigam's type system [19] can be regarded as our type system plus the object types, minus the usage attributes except for the output capability and the input obligation and the tag orderings. In Puntigam's recent work [20], a method reply channel corresponding to a channel of the usage $O_{\mathbf{o}}.0||I_{\mathbf{c}}.0$ and an ordering similar to the tag orderings have been introduced.

Boudol's type system [3] Boudol [3] proposed a kind of deadlock-free type system for the blue calculus [2]. It is conjectured that the type system guarantees deadlock-freedom in the sense that output processes are not blocked forever. For that purpose, the order of communications is expressed by using sequencing, parallel composition, recursion operators on types, which are somewhat similar to the corresponding operators on the usages of our type system. We still have some difficulties in making clear connections, but our current understanding is that his type system corresponds to our type system plus the extended channel types in Section 7.6, minus the input capability and the output obligation, the usage constructor $\sqcap$, and the subusage relation.

9 Conclusion

We have extended our previous type systems for deadlock-freedom [10, 23] and developed its type reconstruction algorithm. A prototype type inference system is available at `http://www.yl.is.s.u-tokyo.ac.jp/~shin/pub/`.

There remain a number of issues in applying our type system and algorithm to real concurrent programming languages [18, 21], such as whether the type system is expressive enough, how to make the algorithm efficient, and how to present the result of type reconstruction to programmers. We plan to perform experiments using existing CML or Pict programs to answer these questions. Future work also includes extensions discussed in Section 7, in particular, the generalization and clarification of the time tags and tag orderings mentioned in Section 7.1 and the object types discussed in Section 7.7.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Gérard Boudol. The pi-calculus in direct syle. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1997.

[3] Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Proceedings of ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, 1997.

[4] J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994.

[5] Simon 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.

[6] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[8] Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation*. To appear. A preliminary summary appeared in Proceedings of SAS'97, LNCS 1302, pp.187-201.

[9] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, available at `http://www.cs.purdue.edu/homes/palsberg/publications.html`, 1999.

[10] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary summary appeared in Proceedings of LICS'97, pages 128–139.

[11] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999. Preliminary summary appeared in Proceedings of POPL'96, pp.358-371.

[12] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

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

[15] Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995. A preliminary version appeared in Proceedings of OOPSLA'93, pp.1-15.

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

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

[18] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[19] Franz Puntigam. Coordination requirements expressed in types for active objects. In *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388, 1997.

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

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

[22] Davide Sangiorgi. The name discipline of uniform receptiveness (extended abstract). In *Proceedings of ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 303–313, 1997.

[23] Eijiro Sumii and Naoki 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.

[24] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer-Verlag, 1994.

[25] Vasco T. Vasconcelos and Kohei 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.

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

A Proof of Theorem 4.47

Lemma A.1: If $\Gamma, x:\tau; \mathcal{T} \vdash P$ and x is not free in P, then $noob(\tau)$ and $\Gamma; \mathcal{T} \vdash P$ hold.

Proof: This follows by straightforward induction on derivation of $\Gamma, x:\tau; \mathcal{T} \vdash P$ (notice that $x:\tau$ can be introduced only by (T-Weak)). □

We write $\tau_1 \leq_S \tau_2$ when $\tau_1 \leq \tau_2$ and $ob(\tau_1)$ implies $ob(\tau_2)$. We extend $\leq_S$ to a relation between type environments by: $\Gamma_1 \leq_S \Gamma_2$ if and only if (i)$dom(\Gamma_1) \supseteq dom(\Gamma_2)$, (ii)$\forall x \in dom(\Gamma_2).(\Gamma_1(x) \leq_S \Gamma_2(x))$, and (iii)$\forall x \in dom(\Gamma_1) \backslash dom(\Gamma_2).(noob(\Gamma_1(x)))$. We write $\Gamma; \mathcal{T} \vdash'' P$ if $\Delta; \mathcal{T} \vdash P$ and $\Gamma \leq_S \Delta$ for some Δ. We also write $\Gamma; \mathcal{T} \vdash' P$ when $\Delta; \mathcal{T} \vdash P$ and $\Gamma \leq \Delta$ for some Δ.

Lemma A.2: Suppose $P \equiv Q$. If $\Gamma; \mathcal{T} \vdash'' P$, then if and only if $\Gamma; \mathcal{T} \vdash'' Q$.

Proof: The proof proceeds by induction on derivation of $P \equiv Q$, with case analysis on the last rule used. Since the induction steps (cases where the last rule is a rule for congruence, symmetry, and transitivity) are trivial, we show only the base cases (except for the cases for the rule of reflexivity and (SCong-Zero), which are also trivial) and the induction step for the congruence on (νx).

- Case for (SCong-Commut): Suppose $\Gamma; \mathcal{T} \vdash'' P \,|\, Q$. By Theorem 5.1, we have $\Gamma_1; \mathcal{T} \vdash P$, $\Gamma_2; \mathcal{T} \vdash Q$, and $\Gamma \leq_S \Gamma_1 + \Gamma_2$. By using (T-Par), we obtain $\Gamma_2 + \Gamma_1; \mathcal{T} \vdash Q \,|\, P$. By the fact $\Gamma_1 + \Gamma_2 \leq_S \Gamma_2 + \Gamma_1$, we obtain $\Gamma; \mathcal{T} \vdash'' Q \,|\, P$.

- Case for (SCong-Assoc): Similar to the above case (notice that $\Gamma_1 + (\Gamma_2 + \Gamma_3) \leq_S (\Gamma_1 + \Gamma_2) + \Gamma_3$ holds).

- Case for (SCong-New): Suppose $\Gamma; \mathcal{T} \vdash'' (\nu x)\,(P \,|\, Q)$ and x is not free in Q. By Theorem 5.1, it must be the case that

$$\begin{array}{l} \Gamma_1; \mathcal{T} \vdash P \\ \Gamma_2; \mathcal{T} \vdash Q \\ \mathit{rel}((\Gamma_1 + \Gamma_2)(x)) \\ \Gamma \leq_S (\Gamma_1 + \Gamma_2)\backslash\{x\} \end{array}$$

  for some Γ_1 and Γ_2. By Lemma A.1, we have $\Gamma_2\backslash\{x\}; \mathcal{T} \vdash Q$ and $\mathit{noob}(\Gamma_2(x))$. So, we have $\mathit{rel}(\Gamma_1(x))$ by Lemma 4.27 and the fact $\mathit{noob}(U_2) \Rightarrow U_1||U_2 \leq U_1$. So, by using (T-New), and (T-Par), we obtain $\Gamma; \mathcal{T} \vdash'' (\nu x)\, P \,|\, Q$.

  On the other hand, suppose $\Gamma; \mathcal{T} \vdash'' (\nu x)\, P \,|\, Q$ and x is not free in Q. By Theorem 5.1, it must be the case that

$$\begin{array}{l} \Gamma_1, x : \tau; \mathcal{T} \vdash P \\ \Gamma_2; \mathcal{T} \vdash Q \\ \mathit{rel}(\tau) \\ \Gamma \leq_S \Gamma_1 + \Gamma_2 \end{array}$$

  By Lemma A.1, $\Gamma_2\backslash\{x\}; \mathcal{T} \vdash Q$ and $\Gamma_2 \leq \Gamma_2\backslash\{x\}$. So, by using (T-Par) and (T-New), we obtain $\Gamma; \mathcal{T} \vdash'' (\nu x)\,(P \,|\, Q)$.

- Case for the congruence rule on (νx): Suppose $\Gamma; \mathcal{T} \vdash'' (\nu x)\, P'$ and $P' \equiv Q'$. By the former assumption, $\Gamma, x : \tau; \mathcal{T} \vdash'' P'$ and $\mathit{rel}(\tau)$ for some τ. By induction hypothesis, we have $\Gamma, x : \tau; \mathcal{T} \vdash'' Q'$. So, there exists Γ', τ' such that $\Gamma', x : \tau'; \mathcal{T} \vdash Q'$, $\Gamma \leq_S \Gamma'$, and $\tau \leq_S \tau'$. By Lemma 4.27, we have $\mathit{rel}(\tau')$. Therefore, we have $\Gamma'; \mathcal{T} \vdash (\nu x)\, Q'$ and $\Gamma \leq_S \Gamma'$ as required.

□

Lemma A.3: If $\Gamma; \mathcal{T} \vdash *P$, then $\Gamma; \mathcal{T} \vdash' P \,|\, *P$.

Proof: Suppose $\Gamma; \mathcal{T} \vdash *P$. By Theorem 5.1, there exist Γ_1 such that $\Gamma_1; \mathcal{T} \vdash P$ and $\Gamma \leq *\Gamma_1$. By using (T-Par), we obtain $\Gamma_1 + *\Gamma_1; \mathcal{T} \vdash P \,|\, *P$. By Lemma 4.19, we have $*\Gamma_1 \leq \Gamma_1 + *\Gamma_1$. Therefore, we can obtain $\Gamma; \mathcal{T} \vdash' P \,|\, *P$. □

Lemma A.4: If $\Gamma; \mathcal{T} \vdash P$ and $x \notin \mathit{dom}(\Gamma)$, then x is not free in P.

Proof: Straightforward induction on derivation of $\Gamma; \mathcal{T} \vdash P$. □

Lemma A.5 [Substitution Lemma]: If $\Gamma, x : \tau; \mathcal{T} \vdash P$ and $\Gamma + v : \tau$ is well defined, then $\Gamma + v : \tau; \mathcal{T} \vdash' [x \mapsto v]P$ holds.

Proof of Lemma A.5: We show the following stronger property by induction on type derivation, with case analysis on the last rule used.

"If $\Gamma, x:\tau; \mathcal{T} \vdash P$ holds and $\Gamma + v:\tau$ is well defined, then $\Gamma + v:\tau; \mathcal{T} \vdash'' [x \mapsto v]P$."

- Case (T-Zero), case (T-Par), (T-New), (T-If), and (T-Weak): Trivial by induction hypothesis.
- Case (T-Out): In this case, it must be the case that:

$$\begin{array}{l} P = z!^a[w_1, \ldots, w_n].\, Q \\ \Gamma_1, z:[\tau_1, \ldots, \tau_n]^t/U; \mathcal{T} \vdash Q \\ a \subseteq a' \\ t\mathcal{T}(w_1:\tau_1 + \cdots + w_n:\tau_n + \Gamma_1) \\ ob(w_1:\tau_1 + \cdots + w_n:\tau_n + \Gamma_1) \Rightarrow \mathbf{c} \subseteq a' \\ \Gamma, x:\tau \leq_S z:[\tau_1, \ldots, \tau_n]^t/O_{a'}.U + w_1:\tau_1 + \cdots + w_n:\tau_n + \Gamma_1 \end{array}$$

  The only nontrivial cases are the case where $z = x$ and $v \in dom(\Gamma_1)$ and the case where $z = v$ and $x \in dom(\Gamma_1)$: the other cases are trivial by induction hypothesis. Suppose $z = x$ and $v \in dom(\Gamma_1)$, i.e., $\Gamma_1 = \Gamma_2, v:[\tau_1, \ldots, \tau_n]^t/U_1$. By the types of $x, w_1, \ldots, w_n$, none of $w_1, \ldots, w_n$ can be x or v (note that we do not have recursive types). By induction hypothesis, we have $\Gamma_2, v:[\tau_1, \ldots, \tau_n]^t/(U||U_1); \mathcal{T} \vdash'' [x \mapsto v]Q$. So, there exists Δ and U_2 such that $\Delta, v:[\tau_1, \ldots, \tau_n]^t/U_2; \mathcal{T} \vdash [x \mapsto v]Q$, $\Gamma_2 \leq_S \Delta$, and $U||U_1 \leq_S U_2$. From the facts $t\mathcal{T}(w_1:\tau_1 + \cdots + w_n:\tau_n + \Gamma_1)$ and $ob(w_1:\tau_1 + \cdots + w_n:\tau_n + \Gamma_1) \Rightarrow \mathbf{c} \subseteq a'$, we also get $t\mathcal{T}(w_1:\tau_1 + \cdots + w_n:\tau_n + \Delta)$ and $ob(w_1:\tau_1 + \cdots + w_n:\tau_n + \Delta) \Rightarrow \mathbf{c} \subseteq a'$. So, by using (T-Out), we obtain $v:[\tau_1, \ldots, \tau_n]^t/O_{a'}.U_2 + w_1:\tau_1 + \cdots + w_n:\tau_n + \Delta; \mathcal{T} \vdash [x \mapsto v]P$. By the condition $t\mathcal{T}(w_1:\tau_1 + \cdots + w_n:\tau_n + \Gamma_1)$, it must be the case that $t\mathcal{T}(v:[\tau_1, \ldots, \tau_n]^t/U_1)$. Since $\mathcal{T}$ is a strict partial order, it must be the case that $U_1 \leq 0$. So, we have $(O_{a'}.U)||U_1 \leq O_{a'}.(U||U_1) \leq O_{a'}.U_2$ by Lemma 4.24. We also have $ob(O_{a'}.U||U_1) \iff \mathbf{o} \subseteq a' \iff O_{a'}.U_2$. Therefore, we have $\Gamma + v:\tau \leq_S v:[\tau_1, \ldots, \tau_n]^t/O_{a'}.U_2 + w_1:\tau_1 + \cdots + w_n:\tau_n + \Delta$, from which $\Gamma + v:\tau; \mathcal{T} \vdash'' [x \mapsto v]P$ follows.

  The case where $z = v$ and $x \in dom(\Gamma_1)$ is similar.

- Case (T-In): Similar to the above case.
- Case (T-Rep): It must be the case that $P = *Q$, $(\Gamma, x:\tau) \leq_S *\Gamma_1$, and $\Gamma_1; \mathcal{T} \vdash Q$. If $x \notin dom(\Gamma_1)$, then it must be the case that $[x \mapsto v]Q = Q$ (by Lemma A.4), $noob(\tau)$, and $\Gamma \leq_S *\Gamma_1$. From Lemma 4.34 and the second and third facts, we get $\Gamma + v:\tau \leq_S \Gamma + \emptyset \leq_S *\Gamma_1$. So, we obtain $\Gamma + v:\tau; \mathcal{T} \vdash'' [x \mapsto v]P$ by using (T-Rep).

  Suppose $x \in dom(\Gamma_1)$ and $\Gamma_1 = \Gamma_1', x:\tau'$. Then, we have $\Gamma \leq_S *\Gamma_1'$ and $\tau \leq_S *\tau'$. Since $\Gamma + v:\tau$ is well defined, so is $\Gamma_1' + v:\tau'$. (Notice that if $\tau_1 \leq \tau_2$ holds, then τ_1 and τ_2 can differ only in their usage.) So, by induction hypothesis, we have Δ such that $\Delta; \mathcal{T} \vdash [x \mapsto v]Q$ and $\Gamma_1' + v:\tau' \leq_S \Delta$. By using (T-Rep), we obtain $*\Delta; \mathcal{T} \vdash [x \mapsto v]P$. The required result follows, because $\Gamma + v:\tau \leq_S *\Gamma_1' + v:*\tau' \leq_S *(\Gamma_1' + v:\tau') \leq_S *\Delta$ (the second relation follows from Lemma 4.24).

□

Proof of Theorem 4.47: The proof proceeds by induction on derivation of $P \xrightarrow{l} Q$, with case analysis on the last rule used.

- Case for (R-Com): It must be the case that

$$\begin{array}{l} P = x!^a[\tilde{v}].\, P_1 \mid x?^{a'}[\tilde{z}].\, P_2 \\ Q = P_1 \mid [\tilde{z} \mapsto \tilde{v}]P_2 \\ l = x \end{array}$$

Assuming $\Gamma, x : [\tilde{\tau}]^t/U; \mathcal{T} \vdash P$, we need to show $\Gamma, x : [\tilde{\tau}]^t/U'; \mathcal{T} \vdash' Q$ for some U' such that $U \longrightarrow U'$. By Theorem 5.1, we have

$$\begin{array}{l}\Gamma_1, x : [\tilde{\tau}]^t/U_1; \mathcal{T} \vdash P_1\\ \Gamma_2, x : [\tilde{\tau}]^t/U_2, \tilde{z} : \tilde{\tau'}; \mathcal{T} \vdash P_2\\ \Gamma, x : [\tilde{\tau}]^t/U \leq \Gamma_1 + v_1 : \tau_1 + \cdots + v_n : \tau_n + \Gamma_2, x : [\tilde{\tau}]^t/(I_{a_1}.U_1||O_{a_2}.U_2)\\ \tau_1 \leq \tau'_1, \ldots, \tau_n \leq \tau'_n\end{array}$$

By the substitution lemma (Lemma A.5), we have $\Gamma_2 + v_1 : \tau'_1 + \cdots + v_n : \tau'_n, x : [\tilde{\tau}]^t/U_2; \mathcal{T} \vdash'' [\tilde{z} \mapsto \tilde{v}]P_2$. By using (T-Par), we obtain $\Gamma, x : [\tilde{\tau}]^t/(U_1||U_2); \mathcal{T} \vdash' Q$. By the fact $U \leq I_{a_1}.U_1||O_{a_2}.U_2 \longrightarrow U_1||U_2$, there exists U' such that $U \longrightarrow U'$ and $U' \leq U_1||U_2$. So, we get $\Gamma, x : [\tilde{\tau}]^t/U'; \mathcal{T} \vdash' Q$ and $U \longrightarrow U'$ as required.

- Case for (R-Par): In this case, it must be the case that $P = P_1 \mid R$, $Q = Q_1 \mid R$, and $P_1 \xrightarrow{l} Q_1$. Suppose $\Gamma, x : [\tilde{\tau}]^t/U; \mathcal{T} \vdash P$ and $l = x$. By Theorem 5.1 and Lemma A.1, we can assume without loss of generality that

$$\begin{array}{l}\Gamma_1, x : [\tilde{\tau}]^t/U_1; \mathcal{T} \vdash P_1\\ \Gamma_2, x : [\tilde{\tau}]^t/U_2; \mathcal{T} \vdash R\\ \Gamma \leq \Gamma_1 + \Gamma_2\\ U \leq U_1||U_2\end{array}$$

  for some U_1, U_2, Γ_1, and Γ_2. By induction hypothesis, $U_1 \longrightarrow U'_1$ and $\Gamma_1, x : [\tilde{\tau}]^t/U'_1; \mathcal{T} \vdash' Q_1$ for some U'_1. By using (T-Par), we obtain $\Gamma, x : [\tilde{\tau}]^t/(U'_1||U_2); \mathcal{T} \vdash' Q_1 \mid R$. By the fact $U \leq U_1||U_2 \longrightarrow U'_1||U_2$, there exists U' such that $U \longrightarrow U'$ and $U' \leq U'_1||U_2$. So, we obtain $\Gamma, x : [\tilde{\tau}]^t/U'; \mathcal{T} \vdash' Q$ and $U \longrightarrow U'$ as required. Case for $l = \epsilon$ is similar.

- Case for (R-New1): In this case, it must be the case that $P = (\nu x)\, P_1$, $Q = (\nu x)\, Q_1$, and $P_1 \xrightarrow{x} Q_1$. Assuming $\Gamma; \mathcal{T} \vdash P$, we shall show $\Gamma; \mathcal{T} \vdash' Q$. By Theorem 5.1, it must be the case that

$$\begin{array}{l}\Gamma_1, x : [\tilde{\tau}]^t/U; \mathcal{T} \vdash P_1\\ \Gamma \leq \Gamma_1\\ \mathit{rel}(U)\end{array}$$

  By induction hypothesis, there exists U' such that $U \longrightarrow U'$ and $\Gamma_1, x : [\tilde{\tau}]^t/U'; \mathcal{T} \vdash' Q_1$. So, there exists Δ, U'' such that $\Delta, x : [\tilde{\tau}]^t/U''; \mathcal{T} \vdash Q_1$, $\Gamma \leq \Delta$, and $U' \leq U''$. By the definition of *rel* and Lemma 4.27, $\mathit{rel}(U'')$ also holds. So, we can obtain $\Delta; \mathcal{T} \vdash Q$ by using (T-New), from which $\Gamma; \mathcal{T} \vdash' Q$ follows.

- Case for (R-New2): Trivial by induction hypothesis.

- Case for (R-IfT): In this case, $P =$ **if** *true* **then** Q **else** Q' . Assuming $\Gamma; \mathcal{T} \vdash P$. we must show $\Gamma; \mathcal{T} \vdash' Q$. By Theorem 5.1, it must be the case that $\Gamma \leq (\Gamma_1 \sqcap \Gamma_2) + \mathit{true} : \mathit{bool}$, $\Gamma_1; \mathcal{T} \vdash Q$ and $\Gamma_2; \mathcal{T} \vdash Q'$. Since $(\Gamma_1 \sqcap \Gamma_2) + \mathit{true} : \mathit{bool} \leq \Gamma_1$, we obtain $\Gamma; \mathcal{T} \vdash' Q$ as required.

- Case for (R-IfF): Similar to the above case.

- Case for (R-Rep): In this case, it must be the case that $P = *P_1 \mid R$ and $*P_1 \mid P_1 \mid R \xrightarrow{l} Q$. Suppose $\Gamma; \mathcal{T} \vdash P$. It suffices to show that $\Gamma; \mathcal{T} \vdash' *P_1 \mid P_1 \mid R$ holds (because the result follows by induction

hypothesis). By Theorem 5.1, we have

$$\Gamma_1; \mathcal{T} \vdash P_1$$
$$\Gamma_2; \mathcal{T} \vdash R$$
$$\Gamma \leq *\Gamma_1 + \Gamma_2$$

From these, we can obtain $*\Gamma_1 + \Gamma_1 + \Gamma_2; \mathcal{T} \vdash *P_1 \mid P_1 \mid R$. By using the fact $\Gamma \leq *\Gamma_1 + \Gamma_2 \leq *\Gamma_1 + \Gamma_1 + \Gamma_2$, we get $\Gamma; \mathcal{T} \vdash' *P_1 \mid P_1 \mid R$, as required.

- Case for (R-Cong): Trivial by induction hypothesis and Lemma A.2

□

B Proof of Theorem 4.49

Definition B.1: A process is *guarded* if it is an inaction, an input, an output, or a conditional expression.

Lemma B.2: For every process P, there exists a set $\{(P_1, \ldots, P_n)\}$ of guarded processes such that P is obtained by combining $P_1, \ldots, P_n$ with process constructors $\mid$, (νx), and $*$.

Proof: Trivial by the syntax of processes. □

Proof of Theorem 4.49: Suppose $\emptyset; \mathcal{T} \vdash P$ and $\mathit{Waiting}(P)$. Then, there must exist a set of guarded processes $\{Q_1, \ldots, Q_n\}$ such that P is obtained by combining $Q_1, \ldots, Q_n$ with process constructors $\mid$, (νy), and $*$. Moreover, by the assumption $\mathit{Waiting}(P)$, Q_i is of the form $x!^a[\tilde{v}].\, Q_i'$ or $x?^a[\tilde{y}].\, Q_i'$ for some $i \in \{1, \ldots, n\}$ and $a(\supseteq \mathbf{c})$. Without loss of generaility we can assume $i = 1$. Suppose $Q_1 = x!^a[\tilde{v}].\, Q_1'$ (the other case is similar). By Theorem 5.1 and the assumption $\emptyset; \mathcal{T} \vdash P$, there exist $\Gamma_1, \ldots, \Gamma_n$ such that (i) $\Gamma_i; \mathcal{T} \vdash_{\mathcal{STR}} Q_i$ for each $i \in \{1, \ldots, n\}$ and (ii) the empty type environment $\emptyset$ can be obtained from $\Gamma_1, \ldots, \Gamma_n$ by only applying the operations $+, *$ and removing a binding $w : \tau$ such that $\mathit{rel}(\tau)$.

If some Q_i is a conditional expression **if** b **then** R_1 **else** R_2 , then b must be *true* or *false* by the typing rules. So, Q_i can be reduced, which implies that the whole process $(\nu \tilde{z})\,(x!^{\mathbf{c}}[\tilde{v}].\, P \mid Q)$ can also be reduced.

Otherwise, each Q_i must be an inaction, an input, or an output process. Let S be the following subset of $\{1, \ldots, n\} \times \mathbf{T}$:

$$\{(i, t_i) \mid Q_i \text{ is of the form } y!^a[\tilde{v'}].\, Q_i',\ \Gamma_i(y) \text{ is of the form } [\tilde{\tau}]^{t_i}/O_{a'}.U, \text{ and } \mathbf{c} \subseteq a'\}$$
$$\cup \{(i, t_i) \mid Q_i \text{ is of the form } y?^a[\tilde{w}].\, Q_i',\ \Gamma_i(y) \text{ is of the form } [\tilde{\tau}]^{t_i}/I_{a'}.U, \text{ and } \mathbf{c} \subseteq a'\}$$

Intuitively, S is the set of pairs of the index of a process trying to use a capability on a channel and the channel's time tag. S at least contains the pair $(1, t_1)$, since $Q_1 = x!^a[\tilde{v}].\, Q_1'$ and $\mathbf{c} \subseteq a$.

Let (i_0, t_{i_0}) be an element of S such that t_{i_0} is minimal (i.e., there is no time tag that is less than t_{i_0}) with respect to $\mathcal{T}$ among the time tags in S. (Such t_{i_0} always exists since $\mathcal{T}$ is a strict partial order.) Suppose Q_{i_0} is an output expression $y!^a[\tilde{v'}].\, Q_{i_0}'$ (the case for input is similar). By the condition (ii) of the type environments $\Gamma_1, \ldots, \Gamma_n$ and the definition of the reliability predicate rel, there must exist $j \in \{1, \ldots, n\}$ such that $\Gamma_j(y)$ is of the form $[\tilde{\tau}]^{t_{i_0}}/U$ and $ob_{\mathbf{I}}(U)$. Because t_{i_0} is a minimal time tag among those in S, by $\Gamma_j; \mathcal{T} \vdash Q_j$ and the typing rules, it must be the case that Q_j is of the form $y?^{a''}[\tilde{w}].\, Q_j'$. Therefore, Q_{i_0} and Q_j can be reduced together, which implies that the whole process P can also be reduced.

□