Type Reconstruction for Linear Pi-Calculus with I/O Subtyping*

Atsushi Igarashi Naoki Kobayashi

Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033 Japan
{igarashi, koba}@is.s.u-tokyo.ac.jp

February 16, 2001

Abstract

Powerful concurrency primitives in recent concurrent languages and thread libraries provide great flexibility about implementation of high-level features like concurrent objects. However, they are so low-level that they often make it difficult to check global correctness of programs or to perform non-trivial code optimization, such as elimination of redundant communication. In order to overcome those problems, advanced type systems for input-only/output-only channels and linear (use-once) channels have been recently studied, but the type reconstruction problem for those type systems remained open, and therefore, their applications to concurrent programming languages have been limited. In this paper, we develop type reconstruction algorithms for variants of Kobayashi, Pierce, and Turner's linear channel type system with Pierce and Sangiorgi's subtyping based on input-only/output-only channel types, and prove correctness of the algorithms. To our knowledge, no complete type reconstruction algorithm has been previously known for those type systems. We have implemented one of the algorithms and incorporated it into the compiler of the concurrent language *HACL*. This paper also shows some experimental results on the algorithm and its application to compile-time optimizations.

keywords: linear type, linear channel, static analysis, concurrent language

1 Introduction

1.1 Background

Advantages and Disadvantages of Low-level Concurrency Primitives. Many recent concurrent languages and thread libraries provide programmers with powerful but rather low-level concurrency primitives: dynamic creation of processes and first-class communication channels. The major advantages of providing those primitives are: (1) complex communication mechanisms can be easily implemented and modified; (2) their semantics can be obtained uniformly in terms of the semantics of those primitives; and (3) implementation of concurrent languages can be substantially simplified. However, such advantages are not free: when low-level primitives are used for implementing high-level features like concurrent objects, useful

*In *Information and Computation*, 161(1):1–44, August 2000. Copyright © 2000 by Academic Press. An earlier version of the paper was presented at the Fourth International Static Analysis Symposium (SAS'97) under the title "Type-Based Analysis of Communication for Concurrent Progaramming Languages", appearing in Lecture Notes on Computer Science, volume 1302, pp 187–201, Springer-Varlag, Berlin.

information about their behavior may be lost, and as a result, it is difficult to check global correctness of programs or to perform non-trivial code optimization, such as elimination of redundant communication.

Type Systems for Process Calculi. In order to overcome the above problems, a number of type systems [Gay93, VH93, PS93, KPT96, Kob98, PS97] have been studied through process calculi. Among them, Pierce and Sangiorgi's input-only/output-only channel type system with subtyping [PS93] and Kobayashi, Pierce, and Turner's linear type system [KPT96] come up with refined process equivalence theories, thus making it easier to reason about program behavior and enabling non-trivial code optimizations.

In order to illustrate the ideas, we consider the following asynchronous process calculus:

$$
\begin{array}{rcll}
P & ::= & P_1 \,|\, P_2 & \text{(parallel execution of } P_1 \text{ and } P_2\text{)} \\
 & | & (\nu x)P & \text{(creates a channel } x \text{ and executes } P\text{)} \\
 & | & x![y_1,\ldots,y_n] & \text{(sends } y_1,\ldots,y_n \text{ along the channel } x\text{)} \\
 & | & x?[y_1,\ldots,y_n].P & \text{(receives values } v_1,\ldots,v_n \text{ along } x \\
 & & & \ \text{and executes } [v_1/y_1,\ldots,v_n/y_n]P\text{)} \\
 & | & x?^*[y_1,\ldots,y_n].P & \text{(replication of } x?[y_1,\ldots,y_n].P\text{)} \\
 & | & \mathbf{0} & \text{(inaction)}
\end{array}
$$

For example, the process $x?[z].\,y![z]$ receives a value along the channel x and then forwards it to the channel y. The process $x?^*[z].y![z]$ represents an infinite number of parallel copies of $x?[z].\,y![z]$, so that it repeatedly forwards values received along x to y. Earlier type systems [Gay93, VH93] for concurrent languages (including CML [Rep91]) were only concerned with the types of values transmitted along channels; so both $f?[x].\,x![1]$ and $f?[x].\,x?[n].\,\mathbf{0}$ are well typed under the type environment $f : [[Int]]$, where $[\tau]$ denotes the type of channels used for transmitting values of type τ.

The idea of the input-only/output-only channel type system is to annotate channel types with information about operations (input and/or output) allowed for channels. Let us write $[\tau]^{(\omega,\,0)}$ for the type of channels used for *receiving* (input of) values of type τ, $[\tau]^{(0,\,\omega)}$ for the type of channels used for *sending* (output of) values of type τ, and $[\tau]^{(\omega,\,\omega)}$ for the type of channels used for both receiving and sending values of type τ. Then, $f : [[Int]^{(0,\,\omega)}]^{(\omega,\,0)} \vdash f?[x].\,x![1]$ is a valid type judgment but $f : [[Int]^{(0,\,\omega)}]^{(\omega,\,0)} \vdash f?[x].\,x?[n].\,\mathbf{0}$ is not. Furthermore, the distinction between input/output capabilities naturally leads to a subtyping relation: an input-only channel type $[\tau]^{(\omega,\,0)}$ is covariant in τ, while an output-only channel type $[\tau]^{(0,\,\omega)}$ is contravariant in τ and an input-output channel type $[\tau]^{(\omega,\,\omega)}$ is invariant. This refined type system admits a coarser process equivalence: for example, if f has type $[Int, [Int]^{(0,\,\omega)}]^{(\omega,\,\omega)}$, the process $(\nu r)\,(f![n,r] \,|\, r?^*[m].r'![m])$ is equivalent to the more efficient process $f![n,r']$ with respect to an appropriate typed barbed congruence [PS93, KPT96], which is not the case in the usual untyped process equivalence [Mil93].

The linear type system further refines channel types by adding information on *how often* channels can be used for input or output. Throughout the paper, a phrase like 'a channel being used for input or output' means that a process *tries to* receive or send a value along the channel, rather than that a process succeeds in receiving or sending a value by finding its communication partner. For example, the type environment $f : [[Int]^{(0,\,1)}]^{(\omega,\,0)}$ means that f can be used many times for receiving a channel and the received channel can then be used for sending an integer *at most once*. So, $f?[x].\,x![1]$ is well typed under that environment but $f?[x].\,(x![1] \,|\, x![2])$ is not. By using such type information, we can safely replace the process $(\nu r)\,(f![n,r] \,|\, r?[m].\,r'![m])$ with the more efficient process $f![n,r']$. As we have argued elsewhere [KNY95, KPT96], this optimization corresponds to tail-call optimization of functions.

The previous papers on the input/output channel type system [PS93] and the linear type system [KPT96] have been only concerned with type checking, and therefore, their applications to concurrent programming languages have been limited. (Pict [PT97] has an input/output channel type system, but does not have a linear channel type system.) The major difficulty in type reconstruction is that those type systems have no principal typing property. For example, there are many possible typings for the process $f![1,r] \mid r?[\,].\, r'![\,]$:

$$\begin{array}{l} f\colon [\mathit{Int}, [\,]^{(0,\,1)}]^{(0,\,1)}, r\colon [\,]^{(1,\,1)}, r'\colon [\,]^{(0,\,1)} \vdash f![1,r] \mid r?[\,].\, r'![\,] \\ f\colon [\mathit{Int}, [\,]^{(1,\,0)}]^{(0,\,1)}, r\colon [\,]^{(\omega,\,0)}, r'\colon [\,]^{(0,\,1)} \vdash f![1,r] \mid r?[\,].\, r'![\,] \\ \vdots \end{array}$$

There is no general typing which represents all the candidates.

1.2 Our Goal and Approach

The main goal of this paper is to develop type reconstruction algorithms both for a variant of the above linear type system [KPT96] and for its extension with subtyping, so that those type systems can be applied to concurrent programming languages without putting an additional burden on programmers.

Our Approach. The key idea of our reconstruction algorithm is to introduce *use variables*, and constraints on them, so that partial information on channel usage can be expressed. In our new type system, the above process is typed as follows:

$$\begin{array}{l} f : [\mathit{Int}, [\,]^{(j_1,\,k_1)}]^{(j_3,\,k_3)}, r : [\,]^{(j_2,\,k_2)}, r' : [\,]^{(j_4,\,k_4)}; \{j_2 \geq j_1 + 1, k_2 \geq k_1, k_3 \geq 1, k_4 \geq 1\} \\ \qquad \vdash f![1,r] \mid r?[\,].\, r'![\,] \end{array}$$

Here, r may be used for input once by $r?[\,].\, r'![\,]$ and j_1 times by a receiver of $f![1,r]$; so, the total number j_2 of allowed inputs on r should be at least $j_1 + 1$. It is expressed by the constraint $j_2 \geq j_1 + 1$. By instantiating use variables $j_1, k_1, \ldots, j_4, k_4$ to elements in $\{0, 1, \omega\}$ so that the constraint is satisfied, we can obtain all the possible typings. Similarly, the process $f![r] \mid r?[x].\, x![\,]$ is typed as (we allow structural subtyping here; without it, the constraint is much simpler):

$$\begin{array}{l} f : [[[\,]^{(j_1,\,k_1)}]^{(j_2,\,k_2)}]^{(j_3,\,k_3)}, r : [[\,]^{(j_4,\,k_4)}]^{(j_5,\,k_5)}; \\ \quad \{k_3 \geq 1, j_5 \geq j_2 + 1, k_5 \geq k_2, k_4 \geq 1, \\ \quad\ j_2 \geq 1 \Rightarrow (j_4 \geq j_1 \wedge k_4 \geq k_1), k_2 \geq 1 \Rightarrow (j_1 \geq j_4 \wedge k_1 \geq k_4)\} \\ \qquad \vdash f![r] \mid r?[x].\, x![\,] \end{array}$$

The constraint $k_2 \geq 1 \Rightarrow (j_1 \geq j_4 \wedge k_1 \geq k_4)$ above expresses the condition that if a receiver on f uses r for sending a channel (i.e., if $k_2 \geq 1$), then the type of the sent channel should be what is expected by a receiver on r (i.e., $[\,]^{(j_1,\,k_1)}$ is a subtype of $[\,]^{(j_4,\,k_4)}$). In general, the constraint on use variables is fairly simple, so that it can be solved by a simple method. Especially, without structural subtyping, our type reconstruction algorithm runs in time polynomial in the size of a process expression.

Applications. Our algorithm can fully recover type information from an unannotated program. So, a programmer needs to put type information only in those places where he or she wants a compiler to check channel usage. Therefore, the algorithm is applicable to ML-style, implicitly-typed concurrent languages such as CML [Rep91] and *HACL* [KY95].

In addition to the check of correct channel usage, type information recovered by our algorithm (especially, information about use-once channels, which we call *linear channels*) is useful for the following optimization of concurrent programs.

(1) Elimination of redundant communication and channel creation: as mentioned above, usage information can be used for tail-call optimization of functions and methods of concurrent objects.

(2) Reduction of the cost of communication: we can optimize run-time representation of a linear channel and also reduce run-time check of its state (and sometimes we can allocate it on a register).

(3) Improvement of memory utilization: the memory space for a linear channel can be reclaimed immediately after it is used for communication.

The amount of performance improvement of course depends on how often linear channels are used in actual concurrent programs. (Informal) profiling of programs written in CML [Rep91], Pict [PT97], and *HACL* [KY95] indicates that linear channels are very frequently used: it is because at least one of the two channels used in a typical function or method call is linear.

Contributions. The main contributions of the present work are: (1) formalization of the type system mentioned above and a proof of the existence of principal typing, (2) development of algorithms to infer usage information, which consists of an algorithm to compute a principal typing and algorithms to solve the derived constraint, and (3) evaluation of performance improvement gained by our type system via simple benchmarks. For clarity and brevity, we use here a pure process calculus as the target language; however, we believe that our technique is applicable to many other concurrent languages [Rep91, YT87, Yon90, PT97, KY95]: in fact, we have already incorporated our type reconstruction algorithm into the compiler of *HACL* (which has functions, records, and a polymorphic type system) and given a formal proof of the correctness of our analysis for *HACL* (please refer to Igarashi's master thesis [Iga97]).

1.3 Structure of the Paper

The rest of this paper is organized as follows. Section 2 introduces a basic linear channel type system with subtyping, and Section 3 shows its correctness. Section 4 and Section 5 are the main part of this paper: in Section 4, we first modify the basic type system by introducing use and type variables and subtyping constraints. The refined type system is shown to have a principal typing property, and an algorithm to compute a principal typing is presented. In Section 5, we show how to reduce the subtyping constraint in the principal typing to a constraint on use variables and solve them. Section 6 reports experimental results of applying our analysis to compile-time optimizations of concurrent programs. After discussing related work in Section 7, we conclude in Section 8.

2 Linear Pi-Calculus with Subtyping

In this section, we introduce the syntax and type system of a process calculus with linear channels, for which we will present a type reconstruction algorithm later in Section 4 and 5. The calculus can be considered an asynchronous fragment of the polyadic π-calculus [Mil93], and it is close to the core languages of *HACL* [KY95] and Pict [PT97]. It is similar to the original linear π-calculus [KPT96], except that it allows subtyping as in [PS93] (but recursive types are not treated here) and that unlimited channels (channels that can be used many times) can be coerced into linear channels.

2.1 Types with Uses

As sketched in Section 1, each instance of the channel type constructor is annotated with usage information, called *uses* given below.

2.1.1 Definition: A *use* is 0, 1, or ω.

The use 0 means that channels can never be used, 1 means that channels can be used at most once, and ω means that channels can be used an arbitrary number of times. We use metavariables $\kappa, \kappa_1, \kappa_2, \ldots$ for uses.

The syntax of types is defined as follows.

2.1.2 Definition: The set of *bare types*, ranged over by ρ, and the set of *types*, ranged over by τ, are given by the following syntax:

$$\begin{array}{lcl} \rho & ::= & [\tau_1, \ldots, \tau_n] \\ \tau & ::= & \rho^{(\kappa_1,\, \kappa_2)} \end{array}$$

A bare type $[\tau_1, \ldots, \tau_n]$ (n may be 0), often abbreviated to $[\tilde{\tau}]$, denotes the type of channels via which a tuple of values of types $\tau_1, \ldots, \tau_n$ is transmitted. The superscripted use κ_1 (κ_2, resp.), often called *input use* (*output use*, resp.), denotes how often the channel is used for input (output, resp.).

2.1.3 Example: A channel type $[\tau]^{(0,\, 1)}$ denotes the type of channels used for sending a value of type τ *at most once* and not used for receiving at all.

We introduce a countably infinite set of *variables*, ranged over by metavariables $x, y, z, \ldots$ to define type environments.

2.1.4 Definition: A *type environment* Γ is a mapping from a finite set of variables to the set of types.

We write $dom(\Gamma)$ for the domain of Γ. We write $x_1\colon \tau_1, \ldots, x_n\colon \tau_n$, abbreviated to $\tilde{x}\colon \tilde{\tau}$, for the type environment Γ such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$. When $x \notin dom(\Gamma)$, we write $\Gamma, x\colon \tau$ for the type environment Γ' such that $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ if $x \neq y$.

Several operations on uses, types, and type environments are defined below.

2.1.5 Definition: The binary relation $\geq$ between uses is the total order defined by $\omega \geq 1 \geq 0$.

2.1.6 Definition: The *summation* of two uses, written $\kappa_1 + \kappa_2$, is the commutative and associative operation that satisfies $0+0 = 0$, $1+0 = 1$, and $1+1 = \omega+0 = \omega+1 = \omega+\omega = \omega$. The summation of two types, written $\tau_1 + \tau_2$, is defined only when their bare types are identical: $\rho^{(\kappa_1,\, \kappa_2)} + \rho^{(\kappa_3,\, \kappa_4)} = \rho^{(\kappa_1+\kappa_3,\, \kappa_2+\kappa_4)}$.

2.1.7 Example: $[[\,]^{(0,\, 1)}]^{(1,\, 0)} + [[\,]^{(0,\, 1)}]^{(1,\, \omega)} = [[\,]^{(0,\, 1)}]^{(\omega,\, \omega)}$.

The operation '+' on types is pointwise extended to type environments.

2.1.8 Definition: The summation $\Gamma_1 + \Gamma_2$ of two type environments is defined as follows:

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

2.1.9 Example: $(x\colon[\,]^{(0,\,1)}, y\colon[\,]^{(1,\,1)}) + (x\colon[\,]^{(1,\,1)}, y\colon[\,]^{(0,\,0)}) = x\colon[\,]^{(1,\,\omega)}, y\colon[\,]^{(1,\,1)}$

2.1.10 Definition: The *product* of two uses, written $\kappa_1 \cdot \kappa_2$, is the commutative and associative operation that satisfies $0 \cdot 0 = 0 \cdot 1 = 0 \cdot \omega = 0$, $1 \cdot 1 = 1$, and $1 \cdot \omega = \omega \cdot \omega = \omega$. The product is extended to an operation on uses and types by $\kappa \cdot \rho^{(\kappa_1,\,\kappa_2)} = \rho^{(\kappa\cdot\kappa_1,\,\kappa\cdot\kappa_2)}$. It is further extended to an operation on uses and type environments by $\kappa\cdot(x_1\colon\tau_1, \ldots, x_n\colon\tau_n) = x_1\colon\kappa\cdot\tau_1, \ldots, x_n\colon\kappa\cdot\tau_n$.

2.1.11 Example: $\omega \cdot [[\,]^{(0,\,1)}]^{(0,\,1)} = [[\,]^{(0,\,1)}]^{(0,\,\omega)}$.

2.1.12 Example: $\omega \cdot (x\colon[\,]^{(0,\,1)}, y\colon[[\,]^{(1,\,1)}]^{(1,\,1)}) = x\colon[\,]^{(0,\,\omega)}, y\colon[[\,]^{(1,\,1)}]^{(\omega,\,\omega)}$.

2.1.13 Example: $0 \cdot (x\colon[[\,]^{(1,\,\omega)}]^{(\omega,\,1)}) = x\colon[[\,]^{(1,\,\omega)}]^{(0,\,0)}$.

2.2 Process Expressions

The syntax of process expressions differs slightly from the one used in Section 1: we introduce new syntax for choice and process definitions (**def** $x[y_1, \ldots, y_n]\colon\tau = P_1$ **in** P_2 **end**, which corresponds to $(\nu x\colon\tau)\,(x?^*[y_1, \ldots, y_n].P_1 \mid P_2)$), and attach type annotations (which will be recovered by our type reconstruction algorithm) to variables in ν-prefix and process definitions.

2.2.1 Definition: The set of *process expressions*, ranged over by P and Q, is defined by:

$$
\begin{array}{rcll}
P & ::= & P_1 \mid P_2 & (\textit{parallel composition}) \\
 & | & (\nu x\colon\tau)P & (\textit{channel creation}) \\
 & | & x![y_1, \ldots, y_n] & (\textit{output atom}) \\
 & | & x_1?[y_1, \ldots, y_{n_1}].P_1 + \cdots & \\
 & & \quad \cdots + x_m?[y_1, \ldots, y_{n_m}].P_m & (\textit{guarded choice of input prefixes}) \\
 & | & \textbf{def } x[y_1, \ldots, y_n]\colon\rho^{(\omega,\,\kappa)} = P_1 \textbf{ in } P_2 \textbf{ end} & (\textit{local process definition})
\end{array}
$$

The bound variables of a process expression are defined in a customary fashion, i.e., (1) a variable x is bound in P of $(\nu x\colon\tau)P$ and in both P_1 and P_2 of **def** $x[y_1, \ldots, y_n]\colon\tau = P_1$ **in** P_2 **end** and, (2) variables $y_1, \ldots, y_n$ are bound in P of $x?[y_1, \ldots, y_n].P$ and **def** $x[y_1, \ldots, y_n]\colon\tau = P$ **in** P_2 **end**. A variable that is not bound will be called a free variable. We define α-conversions of bound variables in a customary manner and assume that implicit α-conversions make all the bound variables in a process expression different from the other bound variables and free variables.

2.2.2 Notation: A sequence of variables $y_1, \ldots, y_n$ (n may be 0) is often written as $\tilde{y}$. We write $[y_1/x_1, \ldots, y_n/x_n]P$, abbreviated to $[\tilde{y}/\tilde{x}]P$, for a process expression obtained from P by replacing the free variables $x_1, \ldots, x_n$ with $y_1, \ldots, y_n$. We often write $\mathbf{0}$ for $(\nu x\colon[\,]^{(0,\,1)})\,x![\,]$. We give $(\nu x\colon\tau)$ and $x?[\tilde{y}]$. a higher precedence than $\mid$; for example, $x?[z].y![z] \mid (\nu w\colon\tau)P_1 \mid P_2$ means $(x?[z].y![z]) \mid ((\nu w\colon\tau)P_1) \mid P_2$. We omit type annotations when they are not important.

The intuitive meanings of the expressions which were not introduced in Section 1 are as follows. A guarded choice of input prefixes $x_1?[y_1,\ldots,y_{n_1}].P_1+\cdots+x_m?[y_1,\ldots,y_{n_m}].P_m$ waits for a value to arrive on one of the channels $x_1,\ldots,x_m$: when it receives $z_1,\ldots,z_{n_i}$ from the channel x_i, it behaves like $[z_1/y_1,\ldots,z_{n_i}/y_{n_i}]P_i$. The local process definition $\mathbf{def}\ x[\tilde{y}]\colon\rho^{(\omega,\,\kappa)}= P_1\ \mathbf{in}\ P_2\ \mathbf{end}$ first creates a fresh channel x and spawns a process that repeatedly receives values $\tilde{z}$ from the channel x and spawns $[\tilde{z}/\tilde{y}]P_1$; it then executes the process P_2. Thus, $x[\tilde{y}]=P_1$ can be regarded as a process definition in the sense that $x![\tilde{z}]$ is always reduced to $[\tilde{z}/\tilde{y}]P_1$. The output use κ represents how often x can be used for output, i.e., how often the process definition can be expanded. Since a process definition is used exactly κ times, we do not have to count the input and output uses separately. Therefore, the input use is set to ω just for technical convenience for presenting a type reconstruction algorithm.

We give several examples of process expressions. (A formal definition of the reduction relation is given in Section 3.)

2.2.3 Example: A process $x![\,]\,|\,y![\,]\,|\,(x?[\,].\,P_1+y?[\,].\,P_2)$ is reduced either to $y![\,]\,|\,P_1$ by communication on x, or to $x![\,]\,|\,P_2$ by communication on y.

2.2.4 Example: A process $\mathbf{def}\ x[y]\colon[[\,]^{(0,\,1)}]^{(\omega,\,\omega)}=y![\,]\ \mathbf{in}\ x![z]\,|\,x![w]\ \mathbf{end}$ is reduced to $\mathbf{def}\ x[y]\colon[[\,]^{(0,\,1)}]^{(\omega,\,\omega)}=y![\,]\ \mathbf{in}\ z![\,]\,|\,w![\,]\ \mathbf{end}$.

2.2.5 Example: Let P be a process

$$\begin{array}{l}\mathbf{def}\ \mathit{fact}[n,r]=\\ \quad\mathbf{if}\ n=0\ \mathbf{then}\ r![1]\ \mathbf{else}\ (\nu r')(\mathit{fact}![n-1,r']\,|\,r'?[k].\,r![k\times n])\\ \mathbf{in}\ \mathit{fact}![2,x]\ \mathbf{end}.\end{array}$$

It computes the factorial of 2 and outputs the result to x. (For simplicity, we assume that we have integers, booleans and several primitives such as $\times,=$, and **if**.) P is reduced as follows:

$$\begin{array}{rl}P & \rightarrow\ \mathbf{def}\ \mathit{fact}[n,r]=\cdots\ \mathbf{in}\ (\nu r')(\mathit{fact}![1,r']\,|\,r'?[k].\,x![k\times 2])\ \mathbf{end}\\ & \rightarrow\ (\nu r')\mathbf{def}\ \mathit{fact}[n,r]=\cdots\ \mathbf{in}\ (\nu r'')(\mathit{fact}![0,r'']\,|\,r''?[k'].\,r'![k'\times 1]\,|\,r'?[k].\,x![k\times 2])\ \mathbf{end}\\ & \rightarrow\ (\nu r')(\nu r'')\mathbf{def}\ \mathit{fact}[n,r]=\cdots\ \mathbf{in} r''![1]\,|\,r''?[k'].\,r'![k'\times 1]\,|\,r'?[k].\,x![k\times 2]\ \mathbf{end}\\ & \rightarrow\ (\nu r')(\nu r'')\mathbf{def}\ \mathit{fact}[n,r]=\cdots\ \mathbf{in}\ r'![1]\,|\,r'?[k].\,x![k\times 2]\ \mathbf{end}\\ & \rightarrow\ (\nu r')(\nu r'')\mathbf{def}\ \mathit{fact}[n,r]=\cdots\ \mathbf{in}\ x![2]\ \mathbf{end}\end{array}$$

2.3 Typing

A type judgment is of the form $\Gamma\vdash P$, read as "P is well-typed under the type environment Γ." It means not only that P is well-typed in the ordinary sense, but also that each channel in P is used according to the uses of its type in Γ or P; for example, the type judgment $\Gamma,x\colon[\tau]^{(1,\,0)}\vdash P$ means that P uses x *for receiving a value of the type τ at most once, and it never uses x for sending a value.* The rules for deriving a type judgment are given below.

Since type environments are concerned with uses of variables, we need to take special care in merging type environments. For example, if $\Gamma_1,x\colon[\tau]^{(0,\,1)}\vdash P_1$ and $\Gamma_2,x\colon[\tau]^{(1,\,0)}\vdash P_2$, then x is totally used for output once and for input once in $P_1\,|\,P_2$. Therefore, the total use of a variable in $P_1\,|\,P_2$ should be obtained by adding the uses in two type environments. Thus, the rule for parallel composition is:

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

On the other hand, in a choice $P_1 + \cdots + P_n$ (where each P_i is an input prefix), only one of $P_1, \dots, P_n$ is executed. So, each expression should be typed under the same type environment. Thus, the rule for choice is:

$$\frac{\Gamma \vdash P_1 \qquad \cdots \qquad \Gamma \vdash P_n}{\Gamma \vdash P_1 + \cdots + P_n} \tag{T-Choice}$$

The rule for an output atom is given as follows.

$$x\colon [\tau_1, \dots, \tau_n]^{(0,\,1)} + y_1\colon \tau_1 + \cdots + y_n\colon \tau_n \vdash x![\tilde{y}] \tag{T-Out}$$

Here, $x\colon [\tau_1, \dots, \tau_n]^{(0,\,1)}$ expresses the fact that x must be a channel that can be used at least for output, and $y_1\colon \tau_1 + \cdots + y_n\colon \tau_n$ takes into account the usage of $y_1, \dots, y_n$ by a receiver.

Similarly, the rule for an input prefix is given as follows.

$$\frac{\Gamma, \tilde{y}\colon \tilde{\tau} \vdash P}{\Gamma + x\colon [\tilde{\tau}]^{(1,\,0)} \vdash x?[\tilde{y}].P} \tag{T-In}$$

The typing rule for a local process definition is:

$$\frac{\begin{array}{c}\Gamma_1, x\colon [\tau_1, \dots, \tau_n]^{(0,\,\kappa_1)}, y_1\colon \tau_1, \dots, y_n\colon \tau_n \vdash P_1 \\ \Gamma_2, x\colon [\tau_1, \dots, \tau_n]^{(0,\,\kappa_2)} \vdash P_2\end{array}}{(\kappa_2 \cdot (\kappa_1 + 1) \cdot \Gamma_1) + \Gamma_2 \vdash \mathbf{def}\ x[y_1, \dots, y_n]\colon [\tau_1, \dots, \tau_n]^{(\omega,\,\kappa_2 \cdot (\kappa_1 + 1))} = P_1\ \mathbf{in}\ P_2\ \mathbf{end}} \tag{T-Def}$$

Note that unlike the case for an input prefix, P_1 of $\mathbf{def}\ x[\tilde{y}] = P_1\ \mathbf{in}\ P_2\ \mathbf{end}$ may be executed more than once: since the first premise $\Gamma_1, x\colon [\tau_1, \dots, \tau_n]^{(0,\,\kappa_1)}, y_1\colon \tau_1, \dots, y_n\colon \tau_n \vdash P_1$ means that type environment Γ_1 is necessary for *one copy* of P_1, Γ_1 should be multiplied by an upper bound $\kappa_2 \cdot (\kappa_1 + 1)$ of the number of spawned P_1s. The reason why the upper bound is given as $\kappa_2 \cdot (\kappa_1 + 1)$ is as follows. The first premise of the rule indicates that x may be used at most κ_1 times for output in the process P_1. Therefore, each time x is used for output in P_2, P_1 is invoked and x may be used κ_1 times for output; moreover, each use of x in P_1 again spawns P_1 and causes x to be used κ_1 times; thus, each use of x for output in P_2 may cause P_1 to be spawned $1 + \kappa_1 + \kappa_1^2 + \cdots$ times. Since the second premise of the rule indicates that x may be used at most κ_2 times for output in P_2, the total number of spawned P_1s is bound by $\kappa_2 \cdot (1 + \kappa_1 + \kappa_1^2 + \cdots) = \kappa_2 \cdot (1 + \kappa_1)$. For example, in the expression $\mathbf{def}\ x[\,] = (x![\,] \,|\, x![\,])\ \mathbf{in}\ x![\,]\ \mathbf{end}$, $x![\,]$ produces two more copies of $x![\,]$, each of which again produces two copies; thus, the total number of messages sent to x is $1 + 2 + 2^2 + \cdots = \omega$.

The rule for channel creation moves the corresponding binding from the type environment to the ν-prefix.

$$\frac{\Gamma, x\colon \tau \vdash P}{\Gamma \vdash (\nu x\colon \tau)P} \tag{T-New}$$

The last two rules T-Sub and T-Weak below are standard rules for subsumption and weakening. $\preceq$ denotes a subtyping relation introduced in the next subsection.

$$\frac{\Gamma, x\colon \tau' \vdash P \qquad \tau \preceq \tau'}{\Gamma, x\colon \tau \vdash P} \tag{T-Sub}$$

$$\frac{\Gamma \vdash P}{\Gamma, x\colon \tau \vdash P} \tag{T-Weak}$$

2.4 Subtyping Relation

A channel of some type may be used as that of another type. For example, since a channel of type $[\tau]^{(\omega,\,\omega)}$ can be used an arbitrary number of times for both input and output, it may be used as a channel of type $[\tau]^{(\kappa_1,\,\kappa_2)}$ for any κ_1 and κ_2. In order to formalize such a type coercion, we give two kinds of subtyping relations: a non-structural subtyping relation $\preceq_{NonStr}$ and a structural subtyping relation $\preceq_{Str}$. The former relation allows only the outermost uses to be changed: for example, $[\tau]^{(0,\,\omega)} \preceq_{NonStr} [\tau']^{(0,\,1)}$ holds only if $\tau = \tau'$. (Thus, the resulting type system is the same as the one presented in [IK97].) The latter relation, based on input-only/output-only channel types [PS93], allows more general type coercion: for example, $[\tau]^{(0,\,\omega)} \preceq_{Str} [\tau']^{(0,\,1)}$ holds if $\tau' \preceq \tau$ (since a value of type τ' can be coerced into that of type τ, it is safe to send a value of type τ' along a channel on which a value of type τ is expected). While the former relation enables a faster analysis of how often channels are used, the latter relation enables a more precise analysis (see Examples 2.4.6 and 2.4.7 given later).

2.4.1 Definition: The binary relation $\preceq_{NonStr}$ on types is defined by: $\rho_1^{(\kappa_{11},\,\kappa_{12})} \preceq_{NonStr} \rho_2^{(\kappa_{21},\,\kappa_{22})}$ iff $\rho_1 = \rho_2$ and $\kappa_{11} \geq \kappa_{21}$ and $\kappa_{12} \geq \kappa_{22}$.

2.4.2 Example: $[[\,]^{(0,\,1)}]^{(\omega,\,\omega)} \preceq_{NonStr} [[\,]^{(0,\,1)}]^{(1,\,1)}$.

2.4.3 Definition: The binary relation $\preceq_{Str}$ between types is defined as the least relation closed under the following rule:[1]

$$\frac{\begin{array}{c} \mathit{Match}(\tau_i, \tau_i') \text{ for } i \in \{1, \ldots, n\} \\ \kappa_1 \geq \kappa_1' \qquad \kappa_2 \geq \kappa_2' \\ \kappa_1' \geq 1 \Rightarrow (\tau_1 \preceq_{Str} \tau_1' \wedge \cdots \wedge \tau_n \preceq_{Str} \tau_n') \qquad \kappa_2' \geq 1 \Rightarrow (\tau_1' \preceq_{Str} \tau_1 \wedge \cdots \wedge \tau_n' \preceq_{Str} \tau_n) \end{array}}{[\tau_1, \ldots, \tau_n]^{(\kappa_1,\,\kappa_2)} \preceq_{Str} [\tau_1', \ldots, \tau_n']^{(\kappa_1',\,\kappa_2')}} \tag{S-Chan2}$$

where the relation $\mathit{Match}(\tau_1, \tau_2)$ between types is defined by

> $\mathit{Match}([\tau_1, \ldots, \tau_n]^{(\kappa_{11},\,\kappa_{12})}, [\tau_1', \ldots, \tau_m']^{(\kappa_{21},\,\kappa_{22})})$ iff $n = m$ and $\mathit{Match}(\tau_i, \tau_i')$ for $i \in \{1, \ldots, n\}$.

2.4.4 Example: $[[\,]^{(0,\,1)}]^{(1,\,1)} \preceq_{Str} [[\,]^{(0,\,\omega)}]^{(0,\,1)}$.

By definition, $\preceq_{Str}$ and $\preceq_{NonStr}$ are partial orders.

With respect to $\preceq_{Str}$, an output-only channel type $[\tilde{\tau}]^{(0,\,\kappa)}$ is contravariant in the argument types $\tau_1, \ldots, \tau_n$, an input-only channel type is covariant, and an input-output channel type is invariant. It means that a sender can put a value of any subtype of τ into a channel of bare type $[\tau]$, while a receiver can use a value extracted from x as a value of any supertype τ'. A type of the form $[\tilde{\tau}]^{(0,\,0)}$ is a supertype of any type τ' as far as $[\tilde{\tau}]^{(0,\,0)}$ and τ' have the same shape (i.e., if they differ only in uses), because a value of type $[\tilde{\tau}]^{(0,\,0)}$ cannot be used at all.

When we write $\Gamma \vdash P$ below, we always assume that it has been derived by using either $\preceq_{NonStr}$ or $\preceq_{Str}$ for $\preceq$ in T-Sub. When we want to make it explicit, we write $\Gamma \vdash_{NonStr} P$ if $\preceq_{NonStr}$ is used, and write $\Gamma \vdash_{Str} P$ if $\preceq_{Str}$ is used.

In the rest of this section, we give several examples of typing of processes.

[1] The first premise $\mathit{Match}(\tau_i, \tau_i')$ could be removed for a sound type system. We include it just in order to simplify discussions on type reconstruction.

2.4.5 Example: $x\colon [[\,]^{(0,\,1)}]^{(0,\,1)} \vdash (\nu y\colon [\,]^{(1,\,1)})(x![y] \mid y?[\,].\,\mathbf{0})$ holds, but $x\colon [[\,]^{(1,\,0)}]^{(0,\,1)} \vdash (\nu y\colon [\,]^{(1,\,1)})(x![y] \mid y?[\,].\,\mathbf{0})$ does not.

As is shown by the following examples, the relation $\preceq_{Str}$ gives a more accurate judgment about the usage of channels (but instead, type reconstruction is more complicated).

2.4.6 Example: Let P be a process $x?[z].\,z![\,] \mid y?[z].\,z?[\,].\,\mathbf{0} \mid x![w]$. Then, P uses w only for output: indeed, P is typed as

$$x\colon [[\,]^{(0,\,\omega)}]^{(\omega,\,\omega)}, y\colon [[\,]^{(\omega,\,0)}]^{(\omega,\,0)}, w\colon [\,]^{(0,\,\omega)} \vdash P$$

by using either subtyping relation. Suppose P is executed in parallel to $Q = u![x] \mid u![y]$. Unless a receiver on u uses x for input, w is still used only for output: in fact, $P \mid Q$ is typed as

$$x\colon [[\,]^{(0,\,\omega)}]^{(\omega,\,\omega)}, y\colon [[\,]^{(\omega,\,0)}]^{(\omega,\,\omega)}, w\colon [\,]^{(0,\,\omega)}, u\colon [[[\,]^{(\omega,\,\omega)}]^{(0,\,\omega)}]^{(0,\,\omega)} \vdash_{Str} P \mid Q$$

since $[[\,]^{(0,\,\omega)}]^{(\omega,\,\omega)} \preceq_{Str} [[\,]^{(\omega,\,\omega)}]^{(0,\,\omega)}$ and $[[\,]^{(\omega,\,0)}]^{(\omega,\,\omega)} \preceq_{Str} [[\,]^{(\omega,\,\omega)}]^{(0,\,\omega)}$. However, the same judgment cannot be derived by using $\preceq_{NonStr}$: we must weaken the type environment to

$$x\colon [[\,]^{(\omega,\,\omega)}]^{(\omega,\,\omega)}, y\colon [[\,]^{(\omega,\,\omega)}]^{(\omega,\,\omega)}, w\colon [\,]^{(\omega,\,\omega)}, u\colon [[[\,]^{(\omega,\,\omega)}]^{(0,\,\omega)}]^{(0,\,\omega)},$$

where information of w being used only for output is lost.

2.4.7 Example: Let P be a process $x![z] \mid z?[\,].\,\mathbf{0} \mid x?[y].\,y![\,]$. Since z is used both for input and for output only once, $(\nu z)P$ is typed as

$$x\colon [[\,]^{(0,\,1)}]^{(\omega,\,\omega)} \vdash (\nu z\colon [\,]^{(1,\,1)})P$$

by using either subtyping relation. Let $Q = u![x] \mid u![v] \mid v?[w].\,(w![\,] \mid w![\,])$. If a receiver on u uses x only for output, z is still used for input and for output once in $(\nu z)P \mid Q$. In fact,

$$x\colon [[\,]^{(0,\,1)}]^{(\omega,\,\omega)}, v\colon [[\,]^{(0,\,\omega)}]^{(\omega,\,\omega)}, u\colon [[[\,]^{(0,\,\omega)}]^{(0,\,\omega)}]^{(0,\,\omega)} \vdash_{Str} (\nu z\colon [\,]^{(1,\,1)})P \mid Q$$

since $[[\,]^{(0,\,1)}]^{(\omega,\,\omega)} \preceq_{Str} [[\,]^{(0,\,\omega)}]^{(0,\,\omega)}$, but it cannot by using $\preceq_{NonStr}$: we must weaken the type of x to $[[\,]^{(0,\,\omega)}]^{(\omega,\,\omega)}$ and attach $[\,]^{(1,\,\omega)}$ to (νz). Thus, information of z being used only once is lost.

3 Correctness of the Type System

We prove soundness of the type system presented in the previous section with respect to operational semantics. By soundness, we mean that the type system correctly estimates how often each channel is used: for example, it must be guaranteed that if $\Gamma, x\colon [\tilde{\tau}]^{(0,\,1)} \vdash P$, then x can be used only for output and at most once during evaluation of P. We first define the operational semantics of the linear π-calculus (in Subsection 3.1), and then show the correctness of the type system with respect to this reduction semantics (in Subsection 3.2), following the corresponding proof for the original linear π-calculus [KPT96].

3.1 Reduction Semantics of Linear Pi-Calculus

Following the standard presentation for process calculi [Mil93], the reduction semantics for process expressions is defined via two relations: a *structural congruence* $P_1 \cong P_2$ and a *reduction relation.*

3.1.1 Definition: Let $FV(y[\tilde{z}] = P)$ be $FV(P) \cup \{y\} \setminus \{\tilde{z}\}$ where $FV(P)$ is the set of free variables in P. *Structural congruence* $P_1 \cong P_2$ is the least congruence on process expressions closed under the following rules.

$$
\begin{array}{rcll}
P_1 \,|\, P_2 & \cong & P_2 \,|\, P_1 & \\
(P_1 \,|\, P_2) \,|\, P_3 & \cong & P_1 \,|\, (P_2 \,|\, P_3) & \\
(\nu x)(P_1 \,|\, P_2) & \cong & P_1 \,|\, (\nu x)P_2 & (\text{if } x \notin FV(P_1)) \\
\mathbf{def}\ x[\tilde{y}] = P_1\ \mathbf{in}\ P_2 \,|\, P_3\ \mathbf{end} & \cong & P_2 \,|\, \mathbf{def}\ x[\tilde{y}] = P_1\ \mathbf{in}\ P_3\ \mathbf{end} & (\text{if } x \notin FV(P_2)) \\
(\nu x)\mathbf{def}\ y[\tilde{z}] = P_1\ \mathbf{in}\ P_2\ \mathbf{end} & \cong & \mathbf{def}\ y[\tilde{z}] = P_1\ \mathbf{in}\ (\nu x)P_2\ \mathbf{end} & (\text{if } x \notin FV(y[\tilde{z}] = P_1))
\end{array}
$$

3.1.2 Example: $u![\,] \,|\, \mathbf{def}\ y[z] = z![\,]\ \mathbf{in}\ (\nu x)x![w]\ \mathbf{end} \cong (\nu x)\mathbf{def}\ y[z] = z![\,]\ \mathbf{in}\ u![\,] \,|\, x![w]\ \mathbf{end}$.

Next, we shall define the reduction relation. Usually, the reduction relation is written as $P \to P'$, which means "P is reduced to P' in one step." In this paper, this relation is annotated with a label l, and written as $P \xrightarrow{l} P'$. The label l is a special symbol ε, a variable, or of the form $x[\tilde{y}] = Q$: $P \xrightarrow{\varepsilon} P'$ means that P is reduced by communication on a bound channel, $P \xrightarrow{x} P'$ means that P is reduced by communication on a free channel x, and $P \xrightarrow{x[\tilde{y}]=Q} P'$ means P is reduced to P' by replacing a single occurrence of $x![\tilde{z}]$ with $[\tilde{z}/\tilde{y}]Q$. Note that on well-typed processes, our reduction relation coincides with the usual (untyped) reduction relation [Mil93].

3.1.3 Definition: The relation $P \xrightarrow{l} P'$ is the least relation closed under the following rules:

$$(\cdots + x?[y_1, \ldots, y_n].\, P + \cdots) \,|\, x![z_1, \ldots, z_n] \xrightarrow{x} [z_1/y_1, \ldots, z_n/y_n]P \qquad (\text{R-Comm})$$

$$x![z_1, \ldots, z_n] \xrightarrow{x[y_1, \ldots, y_n]=P} [z_1/y_1, \ldots, z_n/y_n]P \qquad (\text{R-Call})$$

$$\frac{P_1 \cong P_2 \qquad P_1 \xrightarrow{l} P_1' \qquad P_1' \cong P_2'}{P_2 \xrightarrow{l} P_2'} \qquad (\text{R-Cong})$$

$$\frac{P_1 \xrightarrow{l} P_1'}{P_1 \,|\, P_2 \xrightarrow{l} P_1' \,|\, P_2} \qquad (\text{R-Par})$$

$$\frac{P \xrightarrow{x} P'}{(\nu x\colon \rho^{(\kappa_1,\, \kappa_2)})P \xrightarrow{\varepsilon} (\nu x\colon \rho^{(\kappa_1^-,\, \kappa_2^-)})P'} \qquad (\text{R-New1})$$

$$\frac{P \xrightarrow{l} P' \qquad x \notin FV(l)}{(\nu x\colon \rho^{(\kappa_1,\, \kappa_2)})P \xrightarrow{l} (\nu x\colon \rho^{(\kappa_1,\, \kappa_2)})P'} \qquad (\text{R-New2})$$

$$\frac{P_2 \xrightarrow{x[\tilde{y}]=P_1} P_2'}{\mathbf{def}\ x[\tilde{y}]\colon \rho^{(\omega,\, \kappa)} = P_1\ \mathbf{in}\ P_2\ \mathbf{end} \xrightarrow{\varepsilon} \mathbf{def}\ x[\tilde{y}]\colon \rho^{(\omega,\, \kappa^-)} = P_1\ \mathbf{in}\ P_2'\ \mathbf{end}} \qquad (\text{R-Def1})$$

$$\frac{P_2 \xrightarrow{l} P_2' \qquad x \notin FV(l)}{\mathbf{def}\ x[\tilde{y}]\colon \rho^{(\omega,\, \kappa)} = P_1\ \mathbf{in}\ P_2\ \mathbf{end} \xrightarrow{l} \mathbf{def}\ x[\tilde{y}]\colon \rho^{(\omega,\, \kappa)} = P_1\ \mathbf{in}\ P_2'\ \mathbf{end}} \qquad (\text{R-Def2})$$

where $FV(\varepsilon) = \emptyset$ and κ^- is defined by: $1^- = 0$ and $\omega^- = \omega$ (0^- is undefined).

The main difference from the ordinary reduction semantics lies in rules R-NEW1 and R-DEF1, in which the used capabilities for communication are removed from the binding on the channel. For example, $(\nu x\colon [\,]^{(1,\,1)})\,(x![\,] \mid x?[\,].\,P)$ is reduced to $(\nu x\colon [\,]^{(0,\,0)})\,P$ by communication on x, thus P can no longer use the channel x.

3.1.4 Example: The process expression $\mathbf{def}\ x[w]\colon[\tau]^{(\omega,\,1)} = v![w]\ \mathbf{in}\ y![u] \mid y?[z].x![z]\ \mathbf{end}$ is reduced by communication on the free channel y:

$$\mathbf{def}\ x[w]\colon[\tau]^{(\omega,\,1)} = v![w]\ \mathbf{in}\ y![u] \mid y?[z].x![z]\ \mathbf{end}$$
$$\xrightarrow{y} \mathbf{def}\ x[w]\colon[\tau]^{(\omega,\,1)} = v![w]\ \mathbf{in}\ x![u]\ \mathbf{end}$$

It is further reduced by communication on the channel x:

$$\mathbf{def}\ x[w]\colon[\tau]^{(\omega,\,1)} = v![w]\ \mathbf{in}\ x![u]\ \mathbf{end} \xrightarrow{\varepsilon} \mathbf{def}\ x[w]\colon[\tau]^{(\omega,\,0)} = v![w]\ \mathbf{in}\ v![u]\ \mathbf{end}$$

which is derived by applying rule R-DEF1 to

$$x![u] \xrightarrow{x[w]=v![w]} v![u].$$

3.2 Correctness about Uses of Channels

As in [KPT96], correctness of the type system is shown by the subject reduction theorem (Theorem 3.2.1), which implies that well-typedness of a process is preserved during reduction, together with Theorem 3.2.2, which implies the lack of immediate misuse of channels by any well-typed process expressions. They are valid for both subtyping relations: $\preceq_{NonStr}$ and $\preceq_{Str}$.

The subject reduction theorem is stated below. Note that if the reduction comes from communication on a free channel (the second case below), the reduced process should be well typed under the type environment obtained by removing the consumed capabilities.

3.2.1 Theorem [Subject Reduction]:

1. If $\Gamma \vdash P$ and $P \xrightarrow{\varepsilon} P'$, then $\Gamma \vdash P'$.
2. If $\Gamma, x\colon \rho^{(\kappa_1,\,\kappa_2)} \vdash P$ and $P \xrightarrow{x} P'$, then $\Gamma, x\colon \rho^{(\kappa_1^-,\,\kappa_2^-)} \vdash P'$.

Proof: See Appendix A. □

By the lack of immediate misuse of channels, we mean, for example, that there is no case where $z\colon [\,]^{(0,\,1)} \vdash P$ but $P \cong z![\,] \mid z![\,]$ (i.e., although the type system judges that z is a channel used at most once, two messages are currently sent to z). It is stated as follows.

3.2.2 Theorem [Run-time Safety]: Suppose $\Gamma \vdash P$ and $P \cong (\nu w_1\colon\tau_1)\cdots(\nu w_n\colon\tau_n)$ $\mathbf{def}$ $y_1[\tilde{z}_1]\colon\tau = P_1'$ $\mathbf{in}$ $\mathbf{def}$ $\ldots$ $\mathbf{in}$ $P_1\{\mid P_2\}$ $\mathbf{end}$ $\ldots$ $\mathbf{end}$ (where $\{\mid P\}$ stands for either nothing or a parallel composition with P.)

1. If P_1 is $x![y_1,\ldots,y_n] \mid (\cdots + x?[z_1,\ldots,z_m].P' + \cdots)$, then $n = m$ and for the use pair (κ_1,κ_2) of the binding of x in either Γ or ν-prefix, $\kappa_1 \geq 1$ and $\kappa_2 \geq 1$. Furthermore, x is not bound by **def**.
2. If P_1 is $x![y_1,\ldots,y_n]$, then the output use of the binding of x (in either Γ, ν or **def**) is greater than 0. Moreover, if x is bound by $\mathbf{def}\ x[z_1,\ldots,z_m] = P'\ \mathbf{in} \ldots \mathbf{end}$, then $n = m$.
3. If P_1 is $(\cdots + x?[\tilde{y}].P' + \cdots)$, then the input use of the binding of x is greater than 0. Furthermore, x is not bound by **def**.

4. If P_1 is $x![\tilde{y}] \,|\, x![\tilde{z}]$, then the output use of the binding of x is ω.

5. If P_1 is $(\cdots + x?[\tilde{y}].P' + \cdots) \,|\, (\cdots + x?[\tilde{z}].P'' + \cdots)$, then the input use of the binding of x is ω.

Proof: Trivial from the typing rules and the fact that $\rho_1^{(\kappa_{11},\kappa_{12})} \preceq \rho_2^{(\kappa_{21},\kappa_{22})}$ implies $\kappa_{11} \geq \kappa_{21}$ and $\kappa_{12} \geq \kappa_{22}$. $\square$

4 Type Reconstruction

We now focus on the main goal of this paper: type reconstruction. It consists of two phases: in the first phase, constraints on types and uses are extracted from a given process expression, and in the second phase, the constraints are solved. This section discusses the first phase. The second phase is deferred until Section 5.

The typing rules presented in Section 2 are not suitable for type reconstruction. For example, consider how to infer a typing for the process expression $x![z] \,|\, y![z]$. Rule T-Par tells us to first compute the most general typings for $x![z]$ and $y![z]$, and then add the obtained type environments. However, since we do not know how z will be used by receivers on x and y, the reconstruction step stops there.

In order to avoid it, we introduce type variables and *use variables* to represent undetermined types and uses, and keep information on such variables as a *subtyping constraint*. Thus, the most general typing for a process is represented as a pair consisting of a type environment and a subtyping constraint. For example, the most general typings for $x![z]$ and $y![z]$ can be represented in the forms: $((x\colon \tau_x, z\colon \alpha^{(j_1,\, j_2)}), C_1)$ and $((y\colon \tau_y, z\colon \alpha^{(k_1,\, k_2)}), C_2)$ where α is a type variable, j_1, j_2, k_1, and k_2 are use variables, and C_1 and C_2 are subtyping constraints. From these typings, the typing for $x![z] \,|\, y![z]$ is obtained as a pair $((x\colon \tau_x, y\colon \tau_y, z\colon \beta^{(l_1,\, l_2)}), C_1 \cup C_2 \cup \{\beta^{(l_1,\, l_2)} \preceq \alpha^{(j_1+k_1,\, j_2+k_2)}\})$. This reconstruction step is expressed by the rule:

$$\frac{\Gamma_1; C_1 \vdash P_1 \qquad \Gamma_2; C_2 \vdash P_2 \qquad C \models \Gamma \preceq \Gamma_1 + \Gamma_2 \qquad C \models C_1 \cup C_2}{\Gamma; C \vdash P_1 \,|\, P_2} \quad \text{(ST-Par)}$$

where $C \models \Gamma \preceq \Gamma_1 + \Gamma_2$ means that for each variable x bound in $\Gamma_1 + \Gamma_2$, $\Gamma(x)$ is a subtype of $(\Gamma_1 + \Gamma_2)(x)$ under the assumption C, and $C \models C_1 \cup C_2$ means that C is a stronger constraint than $C_1 \cup C_2$. (The formal definition of $\models$ is found in Definition 4.1.4.) The intended meaning of the new type judgment $\Gamma; C \vdash P$ is that $S\Gamma \vdash SP$ holds for any substitution S of types/uses for type/use variables if S satisfies subtyping relations in C. With these modifications, we can obtain, given a process expression P, the most general pair (Γ, C) such that $\Gamma; C \vdash P$.

The modified type system and the definition of principal typings are given in Subsection 4.1. Then, we describe an algorithm to compute principal typings in Subsection 4.2. Discussions in the rest of this section are independent of whether we use $\preceq_{Str}$ or $\preceq_{NonStr}$ for a subtyping relation $\preceq$.

4.1 Type System for Reconstruction and Principal Typing

As mentioned above, we introduce a countably infinite set of *use variables* (ranged over by $j, k, \ldots$), and replace a use with a *use expression*.

4.1.1 Definition: The set of use expressions is given by the following syntax.

$$\kappa ::= 0 \mid 1 \mid \omega \mid j \mid \kappa_1 + \kappa_2 \mid \kappa_1 \cdot \kappa_2$$

We often call a use expression just a use, and call 0, 1, or ω a *use constant*. We do not distinguish between an expression without use variables and its corresponding use constant. For example, we identify $1+0$ with 1. A countably infinite set of *type variables* (ranged over by $\alpha, \beta, \ldots$) is added to the set of bare types. The operations $+$ and $\cdot$ on types are extended in an obvious way. A substitution, ranged over by S, is a finite mapping from type variables to bare types and from use variables to uses. We write $[\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n, \kappa_1/j_1, \ldots, \kappa_m/j_m]$ for a substitution which maps each α_i to ρ_i and each j_i to κ_i. We write S_1S_2 for the composition of S_1 and S_2. We say S is *ground* if S maps use variables to use constants and type variables to bare types without type variables.

Then, we define a subtyping constraint and several related notions.

4.1.2 Definition: A *subtyping constraint* C is a set of expressions of the form $\tau_1 \preceq \tau_2$.

We often call an element in C a *constraint expression*.

4.1.3 Definition: A ground substitution S is a *solution* of C if and only if $S\tau_1 \preceq S\tau_2$ holds for every expression $\tau_1 \preceq \tau_2$ in C.

4.1.4 Definition: $C_1 \models C_2$ if and only if every solution of C_1 is also a solution of C_2.

By its definition, $\models$ is a preorder (on subtyping constraints) closed under substitutions. When $dom(\Gamma_1) \supseteq dom(\Gamma_2)$, we write $\Gamma_1 \preceq \Gamma_2$ for the subtyping constraint $\{\Gamma_1(x) \preceq \Gamma_2(x) \mid x \in dom(\Gamma_2)\}$.

4.1.5 Example: $\{\alpha^{(j_1,\, j_2)} \preceq \beta^{(k_1,\, k_2)}, \beta^{(k_1,\, k_2)} \preceq \gamma^{(l_1+1,\, l_2)}\} \models \{\alpha^{(j_1,\, j_2)} \preceq \gamma^{(1,\, 0)}\}$.

The new typing rules for reconstruction are shown in Figure 1. Rules T-Sub and T-Weak are combined with the other rules, so that the new rules are syntax-directed.

4.1.6 Example: $x\colon \alpha^{(j,\, k)}, y\colon \beta^{(l,\, m)}; \{\alpha^{(j,\, k)} \preceq [\beta^{(l,\, m)}]^{(0,\, 1)}\} \vdash x![y]$ is derivable in the new type system.

To avoid confusion, we often write $\Gamma \vdash_{\mathcal{TR}} P$ for the type judgment derived from the previous typing rules and $\Gamma; C \vdash_{\mathcal{STR}} P$ for the one from the new typing rules.

The new type system is essentially equivalent to the previous one in the following sense.

4.1.7 Theorem [Equivalence of the Two Type Systems]:

1. Suppose $\Gamma; C \vdash_{\mathcal{STR}} P$. If S is a solution of C and its domain includes all type/use variables in Γ and P, then $S\Gamma \vdash_{\mathcal{TR}} SP$.

2. If $\Gamma \vdash_{\mathcal{TR}} P$, then $\Gamma; \emptyset \vdash_{\mathcal{STR}} P$, where $\emptyset$ is the empty subtyping constraint.

Proof: The first part is proved by straightforward induction on the derivation of $\Gamma; C \vdash_{\mathcal{STR}} P$ because each syntax-directed rule corresponds to a combination of the old rules. The second part follows from the fact that the derivation of $\Gamma; \emptyset \vdash_{\mathcal{STR}} P$ can be constructed by combining derivation steps which use rule T-Weak or T-Sub with the other derivation steps. $\Box$

A principal typing of a process expression P is the most general pair (Γ, C) such that $\Gamma; C \vdash_{\mathcal{STR}} P$. Its formal definition is given as follows.

4.1.8 Definition: (Γ, C) is a *principal typing* of P if and only if the following two conditions are satisfied: (1) $\Gamma; C \vdash_{\mathcal{STR}} P$, and (2) if $\Gamma'; C' \vdash_{\mathcal{STR}} S'P$ for some S', Γ' and C', then there exists a substitution S such that $C' \models SC$ and $\Gamma' \supseteq S\Gamma$ where $S'P = SP$.

$$\frac{\Gamma_1; C_1 \vdash P_1 \qquad \Gamma_2; C_2 \vdash P_2 \qquad C \models \Gamma \preceq \Gamma_1 + \Gamma_2 \qquad C \models C_1 \cup C_2}{\Gamma; C \vdash P_1 \,|\, P_2} \quad (\text{ST-Par})$$

$$\frac{\begin{array}{c}\Gamma_1; C_1 \vdash P_1 \qquad \cdots \qquad \Gamma_n; C_n \vdash P_n \\ C \models \bigcup_i (\Gamma \preceq \Gamma_i) \qquad C \models C_1 \cup \cdots \cup C_n\end{array}}{\Gamma; C \vdash P_1 + \cdots + P_n} \quad (\text{ST-Choice})$$

$$\frac{C \models \Gamma \preceq (x\colon [\tau_1, \ldots, \tau_n]^{(0,\,1)} + y_1\colon \tau_1 + \cdots + y_n\colon \tau_n)}{\Gamma; C \vdash x![y_1, \ldots, y_n]} \quad (\text{ST-Out})$$

$$\frac{\Gamma, \tilde{y}\colon \tilde{\tau}; C \vdash P \qquad C' \models C \qquad C' \models \Gamma' \preceq (\Gamma + x\colon [\tilde{\tau}]^{(1,\,0)})}{\Gamma'; C' \vdash x?[\tilde{y}].P} \quad (\text{ST-In})$$

$$\frac{\begin{array}{c}\Gamma_1, x\colon \tau_{x1}, y_1\colon \tau_1, \ldots, y_n\colon \tau_n; C_1 \vdash P_1 \qquad \Gamma_2, x\colon \tau_{x2}; C_2 \vdash P_2 \\ C \models C_1 \cup C_2 \cup \left\{ \begin{array}{rcl} [\tau_1, \ldots, \tau_n]^{(0,\,\kappa_1)} & \preceq & \tau_{x1}, \\ [\tau_1, \ldots, \tau_n]^{(0,\,\kappa_2)} & \preceq & \tau_{x2}, \\ \tau & \preceq & [\tau_1, \ldots, \tau_n]^{(\omega,\,\kappa_2 \cdot (\kappa_1 + 1))} \end{array} \right\} \\ C \models \Gamma \preceq ((\kappa_2 \cdot (\kappa_1 + 1) \cdot \Gamma_1) + \Gamma_2)\end{array}}{\Gamma; C \vdash \textbf{def } x[y_1, \ldots, y_n]\colon \tau = P_1 \textbf{ in } P_2 \textbf{ end}} \quad (\text{ST-Def})$$

$$\frac{\Gamma, x\colon \tau'; C \vdash P \qquad C \models \tau \preceq \tau'}{\Gamma; C \vdash (\nu x\colon \tau) P} \quad (\text{ST-New})$$

Figure 1: Syntax-directed Typing Rules

4.1.9 Example: Let $P = f![r] \,|\, r?[\,].\, s![\,]$. A principal typing of P is (Γ, C) where:

$$\Gamma = f\colon \alpha_f^{(j_f,\,k_f)}, r\colon \alpha_r^{(j_r,\,k_r)}, s\colon \alpha_s^{(j_s,\,k_s)}$$

$$C = \left\{ \begin{array}{rcl} \alpha_f^{(j_f,\,k_f)} & \preceq & [\beta_r^{(j_{r1},\,k_{r1})}]^{(0,\,1)}, \\ \alpha_r^{(j_r,\,k_r)} & \preceq & \beta_r^{(j_{r1}+j_{r2},\,k_{r1}+k_{r2})}, \\ \beta_r^{(j_{r2},\,k_{r2})} & \preceq & [\,]^{(1,\,0)}, \\ \alpha_s^{(j_s,\,k_s)} & \preceq & [\,]^{(0,\,1)} \end{array} \right\}$$

Since the channels f and s are used for output, C must contain $\alpha_f^{(j_f,\,k_f)} \preceq [\beta_r^{(j_{r1},\,k_{r1})}]^{(0,\,1)}$ and $\alpha_s^{(j_s,\,k_s)} \preceq [\,]^{(0,\,1)}$. Similarly, since the channel r is used for input, C must also contain $\beta_r^{(j_{r2},\,k_{r2})} \preceq [\,]^{(1,\,0)}$. Moreover, because r is also sent to f as a value of type $\beta_r^{(j_{r1},\,k_{r1})}$, the type of r in Γ must be a subtype of the summation of $\beta_r^{(j_1,\,k_1)}$ and $\beta_r^{(j_2,\,k_2)}$: it is expressed by the constraint $\alpha_r^{(j_r,\,k_r)} \preceq \beta_r^{(j_{r1}+j_{r2},\,k_{r1}+k_{r2})}$.

4.2 Type Reconstruction Algorithm *PTU*

A type reconstruction algorithm is obtained by reading the new rules in a bottom-up manner. Before describing our algorithm, we introduce several auxiliary functions $\bigoplus$, $\bigsqcup$, and $\bigodot$ used in the algorithm. $\Gamma_1 \bigoplus \Gamma_2$ computes the most general pair (Γ, C) such that $C \models \Gamma \preceq \Gamma_1 + \Gamma_2$. The functions $\bigsqcup$ and $\bigodot$ are used for solving $C \models \bigcup_i (\Gamma \preceq \Gamma_i)$ and $C \models \Gamma \preceq \kappa \cdot \Gamma'$, respectively.

4.2.1 Definition: $\Gamma_1 \bigoplus \Gamma_2$ is defined as the following procedure:

$\Gamma_1 \bigoplus \Gamma_2 =$
    let
        Γ_1' and Γ_2' be type environments such that
          $dom(\Gamma_1') = dom(\Gamma_2') = dom(\Gamma_1) \cap dom(\Gamma_2)$,
          $\Gamma_1'(x) = \alpha_x^{(j_{x1},\,k_{x1})}$, and $\Gamma_2'(x) = \alpha_x^{(j_{x2},\,k_{x2})}$
              where $\alpha_x, j_{x1}, j_{x2}, k_{x1}$ and k_{x2} are fresh for each x
        $C_i = \{\Gamma_i'(x) \preceq \Gamma_i(x) \mid x \in dom(\Gamma_i')\}$ for $i = 1, 2$
        Γ be a type environment such that
          $dom(\Gamma) = dom(\Gamma_1) \cup dom(\Gamma_2)$ and $\Gamma(x) = \beta_x^{(j_x,\,k_x)}$
              where β_x, j_x, and k_x are fresh
        $C = C_1 \cup C_2 \cup \{\Gamma(x) \preceq \Gamma_1'(x) + \Gamma_2'(x) \mid x \in dom(\Gamma_1) \cap dom(\Gamma_2)\}$
                  $\cup \{\Gamma(x) \preceq \Gamma_1(x) \mid x \in dom(\Gamma_1) \setminus dom(\Gamma_2)\}$
                  $\cup \{\Gamma(x) \preceq \Gamma_2(x) \mid x \in dom(\Gamma_2) \setminus dom(\Gamma_1)\}$
    in (Γ, C)

4.2.2 Definition: $\Gamma_1 \bigsqcup \Gamma_2$ is defined as the following procedure:

$\Gamma_1 \bigsqcup \Gamma_2 =$
    let
        Γ be a type environment such that
          $dom(\Gamma) = dom(\Gamma_1) \cup dom(\Gamma_2)$,
          $\Gamma(x) = \alpha_x^{(j_x,\,k_x)}$ where α_x, j_x and k_x are fresh for each x
        $C = \{\Gamma(x) \preceq \Gamma_1(x) \mid x \in dom(\Gamma_1)\} \cup \{\Gamma(x) \preceq \Gamma_2(x) \mid x \in dom(\Gamma_2)\}$
    in (Γ, C)

$\Gamma_1 \bigoplus \cdots \bigoplus \Gamma_n$ ($n \geq 3$) is defined by $(\Gamma, C' \cup C)$ where $(\Gamma, C) = \Gamma' \bigoplus \Gamma_n$ and $(\Gamma', C') = \Gamma_1 \bigoplus \cdots \bigoplus \Gamma_{n-1}$. We define $\Gamma_1 \bigsqcup \cdots \bigsqcup \Gamma_n$ similarly.

4.2.3 Definition: $\kappa \bigodot \Gamma$ is defined as the following procedure:

$$
\begin{array}{l}
\kappa \bigodot \Gamma = \\
\quad \text{let } \Gamma' \text{ be a type environment such that} \\
\qquad \mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \\
\qquad \Gamma'(x) = \alpha_x^{(j_x,\, k_x)} \text{ where } \alpha_x, j_x \text{ and } k_x \text{ are fresh for each } x \\
\quad\; C = \{\Gamma'(x) \preceq \kappa \cdot \Gamma(x) \mid x \in \mathit{dom}(\Gamma)\} \\
\quad \text{in } (\Gamma', C)
\end{array}
$$

4.2.4 Example: Let $\Gamma_1 = x\colon[\,]^{(0,\,1)}, y\colon[[\,]^{(1,\,1)}]^{(1,\,0)}$ and $\Gamma_2 = x\colon[\,]^{(1,\,0)}$. Then, $\Gamma_1 \bigoplus \Gamma_2 = ((x\colon\beta_x^{(j_x,\,k_x)}, y\colon\beta_y^{(j_y,\,k_y)}), C)$ where:

$$
C = \left\{
\begin{array}{rcl}
\beta_x^{(j_x,\,k_x)} & \preceq & \alpha_x^{(j_{x1}+j_{x2},\,k_{x1}+k_{x2})}, \\
\alpha_x^{(j_{x1},\,k_{x1})} & \preceq & [\,]^{(0,\,1)}, \\
\alpha_x^{(j_{x2},\,k_{x2})} & \preceq & [\,]^{(1,\,0)}, \\
\beta_y^{(j_y,\,k_y)} & \preceq & [[\,]^{(1,\,1)}]^{(1,\,0)}
\end{array}
\right\}
$$

.

The type reconstruction algorithm PTU, shown in Figure 2, takes a process expression P as an input, and returns a principal typing of P. As is already mentioned, a programmer need not put type annotations into a process expression: before passing a process expression to PTU, a system can automatically put $\alpha^{(j,\,k)}$ (α, j, and k are fresh) into a place where type annotations are omitted.

Note that $PTU(P)$ may succeed even when there is no Γ and C such that $\Gamma; C \vdash P$. For example, consider the process **def** $x[\,]\colon[\,]^{(\omega,\,j)} = x![\,]$ **in** $x?[\,].\,x![\,]$ **end**: although it violates the rule that a channel x bound by **def** cannot be used for input, PTU succeeds and outputs a subtyping constraint including $[\,]^{(0,\,k)} \preceq [\,]^{(1,\,1)}$; this kind of process will be rejected when the satisfiability is checked during the phase for constraint solving (described in the next section).

We can prove that $PTU(P)$ computes a principal typing of P.

4.2.5 Theorem [Correctness of PTU]: Given P, if $\Gamma; C \vdash SP$, for some S, Γ, and C, then $PTU(P)$ outputs a principal typing of P.

Proof: See Appendix B. □

5 Recovering Type Annotation by Constraint Solving

In the previous section, we showed how to compute a principal typing (Γ, C) for a given process expression P. By solving the subtyping constraint C, we can find a type annotation for P and detect linear channels. Among many possible type annotations for P, we are interested in an *optimal* one in the sense that the uses of channels created in P are estimated to be as small as possible (because we want to find as many linear channels as possible). We first define the optimality of type annotations and discuss which solution for C gives an optimal annotation in Subsection 5.1. We then show how to compute such a solution for C. It depends of course on particular subtyping relations: we describe the case for $\preceq_{\mathit{NonStr}}$ in Subsection 5.2 and the case for $\preceq_{\mathit{Str}}$ in Subsection 5.3.

Every type/use variable is assumed to be fresh below. $\Gamma \setminus \{x_1, \ldots, x_n\}$ denotes the type environment whose domain is restricted to $dom(\Gamma) \setminus \{x_1, \ldots, x_n\}$.

$$
\begin{array}{l}
PTU(P_1 \mid P_2) = \\
\qquad \text{let } (\Gamma_1, C_1) = PTU(P_1) \\
\qquad\qquad (\Gamma_2, C_2) = PTU(P_2) \\
\qquad\qquad (\Gamma, C) = \Gamma_1 \bigoplus \Gamma_2 \\
\qquad \text{in } (\Gamma, C \cup C_1 \cup C_2)
\end{array}
$$

$$
\begin{array}{l}
PTU(P_1 + \cdots + P_n) = \\
\qquad \text{let } (\Gamma_1, C_1) = PTU(P_1) \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad (\Gamma_n, C_n) = PTU(P_n) \\
\qquad\qquad (\Gamma, C) = \Gamma_1 \bigsqcup \cdots \bigsqcup \Gamma_n \\
\qquad \text{in } (\Gamma, C \cup C_1 \cup \cdots \cup C_n)
\end{array}
$$

$$
\begin{array}{l}
PTU((\nu x\colon \tau)P) = \\
\qquad \text{let } (\Gamma, C) = PTU(P) \\
\qquad \text{in if } x \in dom(\Gamma) \text{ then } (\Gamma \setminus \{x\}, C \cup \{\tau \preceq \Gamma(x)\}) \\
\qquad\qquad\qquad\qquad\quad \text{else } \ (\Gamma, C)
\end{array}
$$

$$
PTU(x![y_1, \ldots, y_n]) = x\colon [\alpha_1^{(j_1,\, k_1)}, \ldots, \alpha_n^{(j_n,\, k_n)}]^{(0,\, 1)} \bigoplus y_1\colon \alpha_1^{(j_1,\, k_1)} \bigoplus \cdots \bigoplus y_n\colon \alpha_n^{(j_n,\, k_n)}
$$

$$
\begin{array}{l}
PTU(x?[y_1, \ldots, y_n].\, P) = \\
\qquad \text{let } (\Gamma', C') = PTU(P) \\
\qquad\qquad (\Gamma, C) = x\colon [\tau_1, \ldots, \tau_n]^{(1,\, 0)} \bigoplus (\Gamma' \setminus \{y_1, \ldots, y_n\}) \\
\qquad\qquad\quad \text{where } \tau_i \ (i \in \{1, \ldots, n\}) \text{ is } \Gamma'(y_i) \text{ if } y_i \in dom(\Gamma'), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{or } \alpha_i^{(j_i,\, k_i)} \text{ otherwise} \\
\qquad \text{in } (\Gamma, C \cup C')
\end{array}
$$

$$
\begin{array}{l}
PTU(\mathbf{def}\ x[y_1, \ldots, y_n]\colon \tau = P_1 \ \mathbf{in}\ P_2 \ \mathbf{end}) = \\
\qquad \text{let } (\Gamma_1, C_1) = PTU(P_1) \\
\qquad\qquad (\Gamma_2, C_2) = PTU(P_2) \\
\qquad\qquad (\Gamma_3, C_3) = (l_2 \cdot (l_1 + 1)) \bigodot (\Gamma_1 \setminus \{x, y_1, \ldots, y_n\}) \\
\qquad\qquad (\Gamma, C_4) = \Gamma_3 \bigoplus (\Gamma_2 \setminus \{x\}) \\
\qquad \text{in } (\Gamma, C_1 \cup C_2 \cup C_3 \cup C_4 \cup \{[\tau_1, \ldots, \tau_n]^{(0,\, l_i)} \preceq \tau_{xi} \mid i \in \{1, 2\}\} \\
\qquad\qquad\qquad\qquad\qquad\quad \cup \{\tau \preceq [\tau_1, \ldots, \tau_n]^{(\omega,\, l_2 \cdot (l_1+1))}\}) \\
\qquad\qquad \text{where } \tau_{xi} \text{ (for } i \in \{1, 2\}) \text{ is } \Gamma_i(x) \text{ if } x \in dom(\Gamma_i), \text{ or } \alpha_i^{(j_i,\, k_i)} \text{ otherwise} \\
\qquad\qquad\qquad \tau_i \text{ (for } i \in \{1, \ldots, n\}) \text{ is } \Gamma_1(y_i) \text{ if } y_i \in dom(\Gamma_1), \text{ or } \alpha_{y_i}^{(j_{y_i},\, k_{y_i})} \text{ otherwise}
\end{array}
$$

Figure 2: Type Reconstruction Algorithm PTU

For simplicity, we consider only closed process expressions. If a process containing free variables should be analyzed (for the purpose of separate compilation, etc.), we must let a programmer declare some type information on the free variables. (Uses need not necessarily be declared: ω can be assigned to unknown uses. In the case for $\preceq_{NonStr}$, even type declaration is unnecessary.) Instead of such type declarations, we can delay instantiations of certain type/use variables and keep constraints on them. An algorithm for such incremental constraint solving can be easily obtained from our algorithms described in Subsection 5.2 and 5.3 since they transform constraints step by step.

5.1 Optimality of Type Annotation

To state the optimality of a type annotation, we introduce an ordering $\geq$ on process expressions. $P \geq P'$ means that two process expressions P and P' are identical except in their type annotations, and that the pair of the (outermost) uses of every type appearing in P is equal to or greater than that of the type in the corresponding position in P'.

5.1.1 Definition: The relation $\geq$ between process expressions is the least relation closed under the following rules:

$$x![y_1, \ldots, y_n] \geq x![y_1, \ldots, y_n]$$

$$\frac{P_1' \geq P_1 \qquad P_2' \geq P_2}{P_1' \,|\, P_2' \geq P_1 \,|\, P_2}$$

$$\frac{P' \geq P \qquad \kappa_{11} \geq \kappa_{21} \qquad \kappa_{12} \geq \kappa_{22}}{(\nu x\colon \rho_1^{(\kappa_{11},\,\kappa_{12})})P' \geq (\nu x\colon \rho_2^{(\kappa_{21},\,\kappa_{22})})P}$$

$$\frac{P_1' \geq P_1 \qquad \cdots \qquad P_m' \geq P_m}{x_1?[y_1, \ldots, y_{n_1}].\, P_1' + \cdots + x_m?[y_1, \ldots, y_{n_m}].\, P_m' \geq x_1?[y_1, \ldots, y_{n_1}].\, P_1 + \cdots + x_m?[y_1, \ldots, y_{n_m}].\, P_m}$$

$$\frac{P_1' \geq P_1 \qquad P_2' \geq P_2 \qquad \kappa_1 \geq \kappa_2}{\textbf{def } x[y_1, \ldots, y_n]\colon \rho_1^{(\omega,\,\kappa_1)} = P_1' \textbf{ in } P_2' \textbf{ end} \geq \textbf{def } x[y_1, \ldots, y_n]\colon \rho_2^{(\omega,\,\kappa_2)} = P_1 \textbf{ in } P_2 \textbf{ end}}$$

Now the optimality of type annotation is defined as follows.

5.1.2 Definition: A substitution S of types/uses for type/use variables in a process expression P is *optimal* (with respect to P) if (1) $\emptyset \vdash_{\mathcal{TR}} SP$ and (2) $S'P \geq SP$ for any S' such that $\emptyset \vdash_{\mathcal{TR}} S'P$.

5.1.3 Example: A substitution $[0/j_1, 1/j_2]$ is optimal with respect to $(\nu x\colon [\,]^{(j_1,\,j_2)})\, x![\,]$ but $[0/j_1, \omega/j_2]$ is not.

Because $\emptyset; C \vdash_{\mathcal{STR}} P$ implies $\emptyset \vdash_{\mathcal{TR}} SP$ for any solution S of C, we can obtain an optimal type annotation by computing a solution that assigns to use variables as small uses as possible.

5.1.4 Definition: A substitution S is a *minimal* solution of C if S is a solution of C and if, for any solution S' of C, $S'j \geq Sj$ for each use variable j appearing in C.

5.1.5 Theorem: Let P be a closed process expression and $PTU(P)$ be $(\emptyset, C)$. S is optimal with respect to P if S is a minimal solution of C and if S assigns 0 to all the use variables appearing not in C but in P.

Proof: Trivial from the definition of a minimal solution. □

By the above theorem, we can focus our attention on finding a minimal solution of a subtyping constraint, which is the subject of the following two subsections.

5.2 Constraint Solving without Structural Subtyping

This subsection discusses how to solve a subtyping constraint in the case where $\preceq$ is the relation $\preceq_{NonStr}$. By the definition of $\preceq_{NonStr}$, each constraint expression $\rho_1^{(\kappa_{11},\,\kappa_{12})} \preceq \rho_2^{(\kappa_{21},\,\kappa_{22})}$ in C is reduced to an equality $\rho_1 = \rho_2$ on bare types and inequalities $\kappa_{11} \geq \kappa_{21}$ and $\kappa_{12} \geq \kappa_{22}$ on uses. Equalities on bare types can be solved by the ordinary first-order unification, since we can assume, by the definition of PTU (and the auxiliary functions), that every use in bare types is a use constant or variable. Inequalities on uses can be solved by a simple iterative method.

Before describing more details, we show a simple example.

5.2.6 Example: Let $C = \{\alpha^{(j,\,k)} \preceq \beta^{(l,\,m)}, \beta^{(l,\,m)} \preceq [\,]^{(1,\,0)}\}$. It is reduced to equality constraints $\alpha = \beta$ and $\beta = [\,]$ and inequalities $j \geq l, k \geq m, l \geq 1$, and $m \geq 0$ on uses. By solving the equality constraints, we obtain $\alpha = \beta = [\,]$, while we obtain $j = 1, k = 0, l = 1$ and $m = 0$ by solving the inequalities. Thus, we obtain a substitution $[[\,]/\alpha, [\,]/\beta, 1/j, 0/k, 1/l, 0/m]$ as a (minimal) solution of C.

Now we describe more details on how to solve equalities on bare types and inequalities on uses. As mentioned above, solving equalities on bare types is a first-order unification problem.

5.2.7 Definition: A substitution S is a *unifier* of a set of pairs of bare types $\{(\rho_1, \rho'_1), \ldots, (\rho_n, \rho'_n)\}$ if $S\rho_i = S\rho'_i$ for $i \in \{1, \ldots, n\}$. A unifier S of $E = \{(\rho_1, \rho'_1), \ldots, (\rho_n, \rho'_n)\}$ is *most general* if for any other unifier S' of E, there exists a substitution S'' such that $S' = S''S$.

5.2.8 Theorem [Unification [Rob65]**]:** Suppose every use expression appearing in $\rho_1, \ldots, \rho_n, \rho'_1, \ldots, \rho'_n$ is either a use constant or variable. Then there exists an algorithm U that computes the most general unifier of the set $\{(\rho_1, \rho'_1), \ldots, (\rho_n, \rho'_n)\}$ if a unifier exists, or reports failure if a unifier does not exist.

The final task is to solve inequalities on uses. Because we are interested in a minimal solution of C, we need to find the least solution, i.e., a solution that assigns to use variables as small uses as possible. By the definition of PTU, we can assume that κ_1 of each inequality $\kappa_1 \geq \kappa_2$ is either a use variable or constant. So, the inequalities on uses are divided into two sets $\{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$ and $\{c_{n+1} \geq \kappa_{n+1}, \ldots, c_m \geq \kappa_m\}$ where each c_i is a use constant. Without loss of generality, we can assume that $j_1, \ldots, j_n$ comprise all of the use variables which occur in the $\kappa_1, \ldots, \kappa_m$ and that they are distinct: we can add constraints $j \geq 0$ when j appear in κ_i but not in $j_1, \ldots, j_n$, and replace inequalities $j \geq \kappa_1, \ldots, j \geq \kappa_l$ with a single inequality $j \geq \kappa_1 \sqcup \cdots \sqcup \kappa_l$, where $\kappa_1 \sqcup \kappa_2$ represents the least upper-bound of κ_1 and κ_2, i.e., $\kappa_1 \sqcup \kappa_2 = \kappa_1$ if $\kappa_1 \geq \kappa_2$ or κ_2 otherwise. The least solution of the first set of inequalities is computed and then it is checked whether it is also a solution of the second part. (If the check fails, the original subtyping constraint has no solutions.) Because every operation on uses is monotonic and use variables $j_1, \ldots, j_n$ can range over finite space $\{0, 1, \omega\}$, the least solution of the first part is calculated by the following simple iterative method:

5.2.9 Lemma [Least Solution of Inequalities between Uses]: Let $\Theta_1 = \{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$ and $\Theta_2 = \{c_{n+1} \geq \kappa_{n+1}, \ldots, c_m \geq \kappa_m\}$ where $j_1, \ldots, j_n$ are distinct and each c_i is a use constant. Define $\kappa_i^{(m)}$ $(m \geq 0, 1 \leq i \leq n)$ by:

$$\begin{aligned}
\kappa_i^{(0)} &= 0 \\
\kappa_i^{(m+1)} &= [\kappa_1^{(m)}/j_1, \ldots, \kappa_n^{(m)}/j_n]\kappa_i
\end{aligned}$$

Then, for some $M \geq 0$, $S = [\kappa_1^{(M)}/j_1, \ldots, \kappa_n^{(M)}/j_n]$ is the least solution of the set Θ_1. If $\Theta_1 \cup \Theta_2$ has a solution, then S is its least solution. Otherwise, S is not a solution of Θ_2.

Proof: Since each use variable can range over the finite space and the operations on uses are monotonic, there exists M such that $\kappa_i^{(M)} = [\kappa_1^{(M)}/j_1, \ldots, \kappa_n^{(M)}/j_n]\kappa_i$ holds for all $i \in \{1, \ldots, n\}$. We show $S = [\kappa_1^{(M)}/j_1, \ldots, \kappa_n^{(M)}/j_n]$ is the least solution of Θ_1. Suppose that $\kappa'_i \geq [\kappa'_1/j_1, \ldots, \kappa'_n/j_n]\kappa_i$ holds for some $\kappa'_1, \ldots, \kappa'_n$. It suffices to show $\kappa'_i \geq \kappa_i^{(n)}$ for any n by mathematical induction on n. The base case $n = 0$ is trivial. Suppose $\kappa'_i \geq \kappa_i^{(m)}$ for $i \in \{1, \ldots, n\}$. Then, for $i \in \{1, \ldots, n\}$,

$$\begin{aligned}
\kappa'_i &\geq [\kappa'_1/j_1, \ldots, \kappa'_n/j_n]\kappa_i \\
&\geq [\kappa_1^{(m)}/j_1, \ldots, \kappa_n^{(m)}/j_n]\kappa_i \\
&= \kappa_i^{(m+1)}
\end{aligned}$$

Next, we show the second part of the lemma. Suppose there is a solution $[\kappa'_1/j_1, \ldots, \kappa'_n/j_n]$ of $\Theta_1 \cup \Theta_2$. Since $[\kappa_1^{(M)}/j_1, \ldots, \kappa_n^{(M)}/j_n]$ is the least solution of Θ_1, from the first part, it must be that $\kappa'_i \geq \kappa_i^{(M)}$ for $i \in \{1, \ldots, n\}$. So, for all $i \in \{n+1, \ldots, m\}$, we have $c_i \geq [\kappa'_1/j_1, \ldots, \kappa'_n/j_n]\kappa_i \geq [\kappa_1^{(M)}/j_1, \ldots, \kappa_n^{(M)}/j_n]\kappa_i$, which implies $[\kappa_1^{(M)}/j_1, \ldots, \kappa_n^{(M)}/j_n]$ is also a solution of $\Theta_1 \cup \Theta_2$. The case where $\Theta_1 \cup \Theta_2$ has no solution is easy since S is a solution of Θ_1, which implies S must not be a solution of Θ_2. $\square$

In the above two steps, we obtain a minimal solution of C:

5.2.10 Theorem [Minimal Solution (I)]: Let C be a subtyping constraint such that for any constraint expression $\tau \preceq \rho^{(\kappa_1, \kappa_2)} \in C$, every use in τ and ρ is either a use variable or constant. Let $E = \{(\rho_1, \rho_2) \mid \rho_1^{(\kappa_{11}, \kappa_{12})} \preceq \rho_2^{(\kappa_{21}, \kappa_{22})} \in C\}$ and $\Theta = \{\kappa_{11} \geq \kappa_{21}, \kappa_{12} \geq \kappa_{22} \mid \rho_1^{(\kappa_{11}, \kappa_{12})} \preceq \rho_2^{(\kappa_{21}, \kappa_{22})} \in C\}$. If C has a solution, then $S_1 = U(E)$ and the least solution S_2 of the set of inequalities $S_1\Theta$ are successfully obtained and S_2S_1 is a minimal solution of C. If C has no solution, then $U(E)$ reports failure, or it is reported that there is no solution for $(U(E))\Theta$.

Proof: Follows from Theorem 5.2.8 and Lemma 5.2.9. $\square$

5.2.1 Computational Complexity

We informally discuss the computational cost of computing the least solution of inequalities on uses. First, we show the cost of computing the least solution of a given set $\{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$ is polynomial in the size n of the set. In the discussion below, $\kappa_i^{(j)}$ refers to the one defined in Lemma 5.2.9 and the size $\mathit{size}(\kappa)$ is defined as 1 if κ is either a constant or a variable, or $1 + \mathit{size}(\kappa_1) + \mathit{size}(\kappa_2)$ if κ is $\kappa_1 + \kappa_2$, $\kappa_1 \sqcup \kappa_2$, or $\kappa_1 \cdot \kappa_2$. Since $\kappa_i^{(j)}$ increases monotonically as j increases, the number of steps M of iteration is at most $2 \cdot n$. In each step, the total cost of computing $\kappa_1^{(j)}, \ldots, \kappa_n^{(j)}$ is $O(\sum_i \mathit{size}(\kappa_i)) (\leq O(n \cdot \max_i(\mathit{size}(\kappa_i))))$.

The number n of the inequalities, which is twice the size of a subtyping constraint set, is $O(p^2)$ where p is the size of a process expression because the auxiliary functions like $\bigoplus$ add at most six times as many constraint expressions as the number of variables in a process expression.

As a result, the computational complexity of computing the least solution is polynomial in the size of an expression. Note that this estimation of the order is very rough and there may be a better upper-bound.

5.3 Constraint Solving with Structural Subtyping

This subsection discusses the case where $\preceq$ is the relation $\preceq_{Str}$. By the definition of rule S-CHAN2, a constraint expression $\tau \preceq \tau'$ implies the condition $Match(\tau, \tau')$. Since $Match(\tau, \tau')$ means that τ and τ' only differ in their uses, it is essentially solved by the first-order unification, with uses ignored. Once $Match(\tau, \tau')$ is satisfied for each constraint expression $\tau \preceq_{Str} \tau'$, it can be reduced to constraints of the form $\kappa_1 \geq 1 \Rightarrow \cdots \Rightarrow \kappa_n \geq 1 \Rightarrow \kappa \geq \kappa'$. They can be solved by an iterative method. We can begin with the assignment of use 0 to all the use variables, and then increase the assigned use step by step until all the constraints get satisfied: if $\kappa_1 \geq 1, \ldots, \kappa_n \geq 1$ become true at some step, then $\kappa_1 \geq 1 \Rightarrow \cdots \Rightarrow \kappa_n \geq 1 \Rightarrow \kappa \geq \kappa'$ is simplified to $\kappa \geq \kappa'$. Before a formal description of the algorithm, let us consider a simple example.

5.3.11 Example: Let $C = \{[[\,]^{(k_1,\,k_2)}]^{(\omega,\,\omega)} \preceq \alpha^{(k_1,\,k_2)}, [\,]^{(k_1,\,k_2)} \preceq [\,]^{(0,\,1)}\}$. By the condition $Match([[\,]^{(k_1,\,k_2)}]^{(\omega,\,\omega)}, \alpha^{(k_1,\,k_2)})$, α is instantiated to a bare type $[[\,]^{(j_1,\,j_2)}]$ for some fresh use variables j_1 and j_2. Then, C is reduced to the following constraints on uses:

$$\begin{array}{rl}
\omega \geq k_1, & \omega \geq k_2 \\
k_1 \geq 1 \Rightarrow k_1 \geq j_1, & k_1 \geq 1 \Rightarrow k_2 \geq j_2 \\
k_2 \geq 1 \Rightarrow j_1 \geq k_1, & k_2 \geq 1 \Rightarrow j_2 \geq k_2 \\
k_1 \geq 0, & k_2 \geq 1,
\end{array}$$

where the first six conditions come from $[[\,]^{(k_1,\,k_2)}]^{(\omega,\,\omega)} \preceq [[\,]^{(j_1,\,j_2)}]^{(k_1,\,k_2)}$, and the last two come from $[\,]^{(k_1,\,k_2)} \preceq [\,]^{(0,\,1)}$. First, let us assign 0 to all the use variables above. Since $k_2 \geq 1$ is not satisfied, the assignment to k_2 is incremented to 1. Now, because $k_2 \geq 1$ is true, the conditions $k_2 \geq 1 \Rightarrow j_1 \geq k_1$ and $k_2 \geq 1 \Rightarrow j_2 \geq k_2$ are simplified to $j_1 \geq k_1$ and $j_2 \geq k_2$. Then, the assignment to j_2 is incremented to 1 so that $j_2 \geq k_2$ is satisfied. Because all the conditions are satisfied now, we obtain $[[\,]^{(0,\,1)}/\alpha, 0/k_1, 1/k_2]$ as a minimal solution for C.

Now we describe the algorithm more formally. First, we extend the definition $Match(\tau, \tau')$ so that $Match(\rho_1^{(\kappa_{11},\kappa_{12})}, \rho_2^{(\kappa_{21},\kappa_{22})})$ holds if both ρ_1 and ρ_2 are type variables. Then, we introduce a procedure $\mathcal{M}$ for the condition $Match(\tau, \tau')$. It takes a subtyping constraint C as an input and outputs the most general substitution S such that $Match(\tau, \tau')$ holds for any constraint expression $\tau \preceq \tau' \in SC$, or outputs *fail* if there is no such substitution. We call a subtyping constraint C a *matching* constraint [FM90] if $Match(\tau, \tau')$ holds for any $\tau \preceq \tau' \in C$. Procedure $\mathcal{M}$ is almost the same as *MATCH* described in [FM90].

5.3.12 Definition: $\mathcal{M}(C)$ is defined as the following procedure where $\uplus$ stands for a union of disjoint sets:

$\mathcal{M}(C) = \hat{\mathcal{M}}(\{(\tau_1, \tau_2) \mid \tau_1 \preceq \tau_2 \in C\}, [\,])$ where $[\,]$ is the empty substitution.
  where
  $\hat{\mathcal{M}}(T \uplus \{([\tau_1, \ldots, \tau_n]^{(\kappa_{11},\,\kappa_{12})}, [\tau'_1, \ldots, \tau'_m]^{(\kappa_{21},\,\kappa_{22})})\}, S)$
    $=$ if $n = m$
      then $\hat{\mathcal{M}}(T \cup \{(\tau_i, \tau'_i) \mid 1 \leq i \leq n\}, S)$
      else *fail*

  $\hat{\mathcal{M}}(T \uplus \{(\alpha^{(\kappa_{11},\,\kappa_{12})}, [\tau_1, \ldots, \tau_n]^{(\kappa_{21},\,\kappa_{22})}\}), S)$

$= \text{if } \alpha \text{ does not occur in } [\tau_1, \ldots, \tau_n]$
$\quad \text{then } \hat{\mathcal{M}}(S_\alpha T \cup \{(\alpha_i^{(j_i,\, k_i)}, \tau_i) \mid 1 \leq i \leq n\}, S_\alpha S)$
$\quad\quad \text{where } S_\alpha = [[\alpha_1^{(j_1,\, k_1)}, \ldots, \alpha_n^{(j_n,\, k_n)}]/\alpha] \text{ and every } \alpha_i^{(j_i,\, k_i)} \text{ is fresh.}$
$\quad \text{else } \mathit{fail}$

$\hat{\mathcal{M}}(T \uplus \{([\tau_1, \ldots, \tau_n]^{(\kappa_{11},\, \kappa_{12})}, \beta^{(\kappa_{21},\, \kappa_{22})})\}, S)$
$= \text{if } \beta \text{ does not occur in } [\tau_1, \ldots, \tau_n]$
$\quad \text{then } \hat{\mathcal{M}}(S_\beta T \cup \{(\beta_i^{(j_i,\, k_i)}, \tau_i) \mid 1 \leq i \leq n\}, S_\beta S)$
$\quad\quad \text{where } S_\beta = [[\beta_1^{(j_1,\, k_1)}, \ldots, \beta_n^{(j_n,\, k_n)}]/\beta] \text{ and every } \beta_i^{(j_i,\, k_i)} \text{ is fresh.}$
$\quad \text{else } \mathit{fail}$

$$\hat{\mathcal{M}}(\{(\alpha_1^{(\kappa_{11},\, \kappa_{12})}, \beta_1^{(\kappa'_{11},\, \kappa'_{12})}), \ldots, (\alpha_n^{(\kappa_{n1},\, \kappa_{n2})}, \beta_n^{(\kappa'_{n1},\, \kappa'_{n2})})\}, S) = S$$

5.3.13 Lemma: Let C be a subtyping constraint. If there exists S' such that $S'C$ is a matching constraint, then $\mathcal{M}(C)$ succeeds and outputs S such that: (1) SC is a matching constraint, and (2) there exists a substitution S'' such that $S' = S''S$. Otherwise, $\mathcal{M}(C)$ outputs *fail*.

Proof: Similar to the proof of the correctness of *MATCH* in [FM90]. □

5.3.14 Example: $\mathcal{M}(\{[\alpha^{(j_1,\, k_1)}, \beta^{(j_2,\, k_2)}]^{(0,\, \omega)} \preceq [\gamma^{(j_3,\, k_3)}, [\,]^{(1,\, 1)}]^{(j_4,\, k_4)}\}) = [[\,]/\beta]$.

Because we are interested only in one minimal solution for C, we can assign the bare type $[\,]$ to all the type variables that have not been instantiated by $\mathcal{M}$. After that, the subtyping constraint is transformed to simpler constraints on uses step by step. We describe the algorithm as rewriting of a quadruple $(C, \Theta_1, \Theta_2, S)$. C represents a subtyping constraint which has not yet been reduced to constraints on uses. Θ_1 and Θ_2 are sets of inequalities on uses: Θ_1 keeps inequalities whose satisfiability needs to be checked, and Θ_2 keeps already checked inequalities. We begin with $(C, \emptyset, \emptyset, [0/j_1, \ldots, 0/j_n])$ where C is a matching constraint and $j_1, \ldots, j_n$ are the use variables in C. During rewriting steps, $S(= [c_1/j_1, \ldots, c_n/j_n])$ is always a solution of Θ_2 and the satisfiability of $C \cup \Theta_1 \cup \Theta_2 \cup \{j_1 \geq c_1, \ldots, j_n \geq c_n\}$ is preserved. If C has a solution, rewriting always terminates with a quadruple $(C', \emptyset, \Theta'_2, S)$ where C' contains only expressions of the form $\kappa \geq 1 \Rightarrow \tau_1 \preceq \tau_2$ for κ such that $S\kappa \not\geq 1$. Then, S is a minimal solution of C.

We need several preliminary definitions. In order to transform a subtyping constraint $\tau \preceq \tau'$ into conditions on uses step by step, we extend the syntax of a constraint expression and the definition of a solution of a subtyping constraint.

5.3.15 Definition: An *extended subtyping constraint* C is a set of expressions of the form $\tau_1 \preceq \tau_2$ or $\kappa \Rightarrow \tau_1 \preceq \tau_2$. An extended subtyping constraint C is said to be matching if $\mathit{Match}(\tau_1, \tau_2)$ for any $\tau_1 \preceq \tau_2$ and $\kappa \Rightarrow \tau_1 \preceq \tau_2 \in C$.

5.3.16 Definition: A substitution S is a *solution* of an extended subtyping constraint C if and only if $S\tau_1 \preceq S\tau_2$ for each $\tau_1 \preceq \tau_2$ in C, and $S\kappa \geq 1$ implies $S\tau_1 \preceq S\tau_2$ for each $\kappa \Rightarrow \tau_1 \preceq \tau_2$ in C.

5.3.17 Example: $[0/j, \omega/k, 0/l]$ is a solution of $\{j \Rightarrow [\,]^{(1,\, \omega)} \preceq [\,]^{(k,\, l)}\}$ but $[1/j, \omega/k, 0/l]$ is not.

5.3.18 Definition: The relation $\rightsquigarrow$ between quadruples $(C, \Theta_1, \Theta_2, S)$ is the least relation closed under the following rules:

$$(C, \Theta_1 \uplus \{\kappa_1 \geq \kappa_2\}, \Theta_2, S) \;\rightsquigarrow\; (C, \Theta_1, \Theta_2 \cup \{\kappa_1 \geq \kappa_2\}, S)$$

$$
\begin{array}{rcl}
& & \text{if } S\kappa_1 \geq S\kappa_2 \\
(C, \Theta_1 \uplus \{j \geq \kappa\}, \Theta_2, S) & \rightsquigarrow & (C, \Theta_1 \cup \Theta_2, \{j \geq \kappa\}, [S\kappa/j]S) \\
& & \text{if } Sj \not\geq S\kappa \\
(C \uplus \{[\tilde{\tau}]^{(\kappa_{11},\, \kappa_{12})} \preceq [\tilde{\tau'}]^{(\kappa_{21},\, \kappa_{22})}\}, \Theta_1, \Theta_2, S) & \rightsquigarrow & (C \cup \{\kappa_{21} \Rightarrow \tau_i \preceq \tau'_i \,|\, 1 \leq i \leq n\} \\
& & \quad \cup \{\kappa_{22} \Rightarrow \tau'_i \preceq \tau_i \,|\, 1 \leq i \leq n\}, \\
& & \Theta_1 \cup \{\kappa_{11} \geq \kappa_{21}, \kappa_{12} \geq \kappa_{22}\}, \Theta_2, S) \\
(C \uplus \{\kappa \Rightarrow \tau_1 \preceq \tau_2\}, \Theta_1, \Theta_2, S) & \rightsquigarrow & (C \cup \{\tau_1 \preceq \tau_2\}, \Theta_1, \Theta_2, S) \\
& & \text{if } S\kappa \geq 1
\end{array}
$$

5.3.19 Example: The quadruple $(\{j \Rightarrow [\,]^{(k,\, l)} \preceq [\,]^{(0,\, 1)}\}, \{j \geq 1\}, \emptyset, [0/j, 0/k, 0/l])$ rewrites to $(\emptyset, \emptyset, \{j \geq 1, k \geq 0, l \geq 1\}, [1/j, 0/k, 1/l])$ in the following steps:

$$
\begin{array}{l}
(\{j \Rightarrow [\,]^{(k,\, l)} \preceq [\,]^{(0,\, 1)}\}, \{j \geq 1\}, \emptyset, [0/j, 0/k, 0/l]) \\
\quad \rightsquigarrow (\{j \Rightarrow [\,]^{(k,\, l)} \preceq [\,]^{(0,\, 1)}\}, \emptyset, \{j \geq 1\}, [1/j, 0/k, 0/l]) \\
\quad \rightsquigarrow (\{[\,]^{(k,\, l)} \preceq [\,]^{(0,\, 1)}\}, \emptyset, \{j \geq 1\}, [1/j, 0/k, 0/l]) \\
\quad \rightsquigarrow (\emptyset, \{k \geq 0, l \geq 1\}, \{j \geq 1\}, [1/j, 0/k, 0/l]) \\
\quad \rightsquigarrow \cdots \rightsquigarrow (\emptyset, \emptyset, \{j \geq 1, k \geq 0, l \geq 1\}, [1/j, 0/k, 1/l])
\end{array}
$$

A quadruple $(C, \Theta_1, \Theta_2, S)$ is called a normal form if there is no $(C', \Theta'_1, \Theta'_2, S')$ such that $(C, \Theta_1, \Theta_2, S) \rightsquigarrow (C', \Theta'_1, \Theta'_2, S')$. We can obtain a minimal solution of a matching constraint by using the rewriting system.

5.3.20 Lemma: Let C be a matching constraint such that for any constraint expression $\tau \preceq \rho^{(\kappa_1,\, \kappa_2)} \in C$, every use appearing in τ or ρ is either a use variable or constant. Then, $([[\,]/\alpha_1, \ldots, [\,]/\alpha_n]C, \emptyset, \emptyset, [0/j_1, \ldots, 0/j_n])$, where $\alpha_1, \ldots, \alpha_n$ are the type variables and $j_1, \ldots, j_n$ are the use variables in C, always rewrites to a normal form $(C', \Theta_1, \Theta_2, S)$ by $\rightsquigarrow$. If C has a solution, then Θ_1 is empty and $S[[\,]/\alpha_1, \ldots, [\,]/\alpha_n]$ is a minimal solution of C. If C does not have any solutions, then Θ_1 is not empty.

Proof: See Appendix C. □

By combining $\mathcal{M}$ and the above rewriting system, we obtain an algorithm to compute a minimal solution of a subtyping constraint.

5.3.21 Theorem [Minimal Solution (II)]: Let C be a subtyping constraint such that for any constraint expression $\tau \preceq \rho^{(\kappa_1,\, \kappa_2)}$ in C, every use appearing in τ or ρ is either a use variable or constant. If there is a solution for C, then $S_1 (= \mathcal{M}(C))$ and a minimal solution S_2 of S_1C is successfully obtained by the rewriting system, and S_2S_1 is a minimal solution of C. If there is no solution for C, then $\mathcal{M}(C)$ outputs *fail*, or the rewriting system reports that there is no solution for $(\mathcal{M}(C))C$.

Proof: Follows from Lemmas 5.3.13 and 5.3.20. □

5.4 An Example of Detection and Optimization

We show an example of detection of linear channels using $\preceq_{NonStr}$. Consider the following (closed) process expression, which computes the n-th Fibonacci number sequentially (the language has been extended with integer values, boolean values and several operations like **if** or $+$):

$$
\begin{array}{l}
\textbf{def } \mathit{fib}[n, c] = \textbf{if } n < 2 \textbf{ then } c![1] \\
\qquad\qquad\qquad \textbf{else } (\nu c_1\colon \alpha^{(j,\,k)})(\nu c_2) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{fib}![n-1, c_1] \mid c_1?[x].(\mathit{fib}![n-2, c_2] \mid c_2?[y].c![x+y])) \\
\textbf{in } \mathit{fib}![n, \mathit{output}] \textbf{ end}
\end{array}
$$

The variable *output* denotes a special channel of type $[Int]^{(0,\,\omega)}$. Let us infer the values of use variables (j, k) attached to the channel c_1. PTU outputs the pair $(\emptyset, C)$ where

$$
C = \left\{ \begin{array}{rcl}
\alpha^{(j,\,k)} & \preceq & \alpha^{(j_1+j_2,\,k_1+k_2)}, \\
\beta^{(l_1,\,m_1)} & \preceq & [Int, \alpha^{(j_1,\,k_1)}]^{(0,\,1)}, \\
\alpha^{(j_2,\,k_2)} & \preceq & [Int]^{(1,\,0)}, \\
[Int, \gamma^{(j_3,\,k_3)}]^{(0,\,m)} & \preceq & \beta^{(l_1+l_2,\,m_1+m_2)}, \\
\gamma^{(j_3,\,k_3)} & \preceq & [Int]^{(0,\,1)}, \\
 & \vdots &
\end{array} \right\}.
$$

(We show a slightly simplified constraint.) The type variables β and γ denote the bare types of *fib* and c, respectively. The use variables j_1 and k_1 denote how often c_1 is used for input and output in the process $\mathit{fib}![n-1, c_1]$. Similarly, j_2 and k_2 denote the uses of c_1 in the process $c_1?[x].(\cdots)$. By the rule for parallel composition, we obtain the expression $\alpha^{(j,\,k)} \preceq \alpha^{(j_1+j_2,\,k_1+k_2)}$. The expression $\alpha^{(j_2,\,k_2)} \preceq [Int]^{(1,\,0)}$ is also obtained from the input prefix $c_1?[x].(\cdots)$. Similarly, $\beta^{(l_1,\,m_1)} \preceq [Int, \alpha^{(j_1,\,k_1)}]^{(0,\,1)}$ is obtained from the expression $\mathit{fib}![n-1, c_1]$, and $\gamma^{(j_3,\,k_3)} \preceq [Int]^{(0,\,1)}$ from $c![1]$. In the first step, we know $\alpha = \gamma = [Int]$ and $\beta = [Int, [Int]^{(j_1,\,k_1)}]$ and $j_1 = j_3$ and $k_1 = k_3$, and obtain the inequalities:

$$
j \geq j_1 + j_2,\ k \geq k_1 + k_2,\ j_2 \geq 1,\ k_1 \geq 1,\ \ldots
$$

By solving them, we obtain $j = k = 1$, which implies that c_1 is a linear channel. Similarly, we know that c_2 is also linear.

By using the information obtained above, we can replace the process with the following optimized one:

$$
\begin{array}{l}
\textbf{def } \mathit{fibopt}[n, c]\colon [Int, [Int]^{(0,\,1)}]^{(\omega,\,\omega)} = \\
\quad \textbf{if } n < 2 \textbf{ then } c![1] \\
\quad \textbf{else def } c_1[x]\colon [Int]^{(\omega,\,1)} = (\textbf{def } c_2[y]\colon [Int]^{(\omega,\,1)} = c![x+y] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \mathit{fibopt}![n-2, c_2] \textbf{ end}) \\
\qquad\quad \textbf{in } \mathit{fibopt}![n-1, c_1] \textbf{ end} \\
\textbf{in } \mathit{fibopt}![n, \mathit{output}] \textbf{ end}
\end{array}
$$

In the optimized program, the channels c_1 and c_2 for receiving results of recursive calls are created by **def**. Since a value sent to those channels will always be received immediately, variable-sized buffers for storing sent values or blocked receivers are no longer required in implementing channels created by **def**, thus communication on them can be implemented more efficiently. Moreover, since c_1 and c_2 are linear channels, the memory space for them can be reclaimed immediately after they are once used. Note that the optimized process corresponds to the continuation passing style representation [App92] of the functional Fibonacci program (the channels c_1 and c_2 can be viewed as continuations).

6 Experimental Results

In this section, we show results of simple experiments with a *HACL* compiler[2] to evaluate performance improvement obtained by our analysis and show elapsed time for analysis. Application programs include the example of Fibonacci function described in the previous section, and concurrent objects expressed in *HACL*.

6.1 Encoding and its Optimization of Concurrent Objects

Before showing the results, we explain how concurrent objects are realized in our language, and what optimization is enabled by our analysis. The state of a concurrent object is implemented by using a channel, while each method is implemented by a process which first extracts the current state from the channel, executes the method, replies a result, and puts the new state into the channel. So, the following fragment of a process expression corresponds to a typical method definition:

$$\mathbf{def}\ m[arg, r] = state?[s].(\cdots \mid r![result] \mid state![news])\ \mathbf{in} \ldots$$

On receiving from m the argument *arg* of the method and the channel r for replying the result, it extracts the current state from the channel *state*. After some computation, the result is replied to r and the new state *news* of the object is put into *state*. A caller of the method is typically of the form:

$$(\nu r)(m![v, r] \mid r?[x].P)$$

It first creates the channel r for receiving the result, and invoke a method m with the argument v. Finally, it waits for the result from r and carry out the rest of computation P.

With our analysis, it can be translated to $\mathbf{def}\ r[x] = P\ \mathbf{in}\ m![v, r]\ \mathbf{end}$ in most cases. Similarly to the example of the Fibonacci function described in the previous section, the transformed process can be implemented more efficiently.

6.2 Experiment Results and Evaluation

We evaluate performance improvement through six programs: a sequential Fibonacci program `sfib25` (the one shown in the optimization example where $n = 25$), a parallel Fibonacci program `pfib25` (which performs recursive calls in parallel), a counter increment program `counter10000` (which creates a counter object and increments its value 10000 times), a tree summation program `tree14` (which creates a binary tree of which each node is a concurrent object, and computes the summation of the values of its leaf nodes), a simulation of Conway's life game `life`, and the Knuth-Bendix completion algorithm `kb`.

In this experiment, we applied two kinds of optimization:

- Source-level program transformation: every process expression of the form $(\nu x)(P \mid x?[\tilde{y}].\,Q)$ where x is known to be a linear channel, is transformed to $\mathbf{def}\ x[\tilde{y}] = Q\ \mathbf{in}\ P\ \mathbf{end}$. As a special case, if $Q = z![\tilde{y}]$, then the entire expression is replaced with $[z/x]P$. After this transformation, we may be able to *hoist* up [App92] the process definition towards the top-level, reducing the number of dynamic process creation. For example, $z?[\tilde{w}].\,\mathbf{def}\ x[\tilde{y}] = Q\ \mathbf{in}\ P\ \mathbf{end}$ can be transformed to $\mathbf{def}\ x[\tilde{y}] = Q\ \mathbf{in}\ z?[\tilde{w}].\,P\ \mathbf{end}$ when

[2] The *HACL* compiler translates a *HACL* program to a C program in a manner similar to sml2c [TAL90]. The compiler is available both for a single processor workstation and for network of workstations. We just show performance on a single processor workstation.

Table 1: Running time for the benchmark programs

	naive (sec)	opt. (sec)	func. (sec)
`sfib25`	1.45	0.57	0.41
`pfib25`	1.76	0.93	—
`counter10000`	0.26	0.17	—
`tree14`	1.55	1.36	—
`life`	2.49	2.45	—
`kb`	27.8	22.6	—

no w_i is a free variable of Q; as a result, the receiver process waiting on x can be created statically. It has been applied to most of the method invocations of concurrent objects or the function calls.

- Refinement of the run-time representation of linear channels. Ordinary channels were implemented by using two variable-sized buffers, one for storing sent values, and the other for storing blocked processes waiting on the channel for values to be received. On the other hand, linear channels were implemented by using one-place buffers. This saves the memory space required for channels.

  Moreover, send/receive operations for linear channels were optimized. In the send function for an ordinary channel, it is first checked whether the receiver buffer of the channel is empty: if so, the sent value is put into the buffer, and otherwise a receiver is extracted from the buffer and executed. So, each call of the send function involves channel status check and update of value/receiver buffers. On the other hand, for a linear channel, those costs were substantially reduced. First, buffers were not updated once communication has occurred. Second, the channel status check was eliminated. Notice that, when a value is sent, there are only two status: the channel is empty, or a receiver is ready. So, we can prepared two versions of send functions, and just switched them (by using function pointers) after a receiver was put into the buffer.

We have not yet applied immediate reclamation of memory space for used linear channels.

Each row in Table 1 shows the result for each program. The first column ("naive") shows the running times of unoptimized programs written with concurrency primitives, and the second column ("opt.") shows the running times of programs optimized with our analysis. In addition, we show in the third column ("func.") the running time of a program written with function primitives for the sequential Fibonacci. Note that all the programs are executed on a single processor machine (Sun Sparc Station 20 (Hyper SPARC 150Mhz $\times 1$)): therefore, `pfib25` is slower than `sfib25` because of overheads.

The result of the sequential Fibonacci program `sfib25` indicates that even if programmers implement functional computations using concurrency primitives, the compiler can generate an optimized code which is comparable to the one written by directly using function primitives. The speedup ratio of the parallel Fibonacci program `pfib25` is relatively smaller because of overheads of multi-threading, but it is still large. Note that the speedup (100–200%) of `sfib25` and `pfib25` itself should not be deemed important, because the execution time of the Fibonacci program is dominated by communications and/or function calls rather than local computations (integer comparison and summation). Because `life` and `kb` perform communications less frequently than `sfib25` or `pfib25`, their speedups are much smaller.

Table 2: Elapsed time for analysis

	size (nodes)	reconst. (msec)	solve (msec)	standard (msec)
`pfib25`	67	43	8	6
`sfib25`	68	40	8	9
`tree14`	145	96	17	14
`counter10000`	153	103	25	17
`life`	613	849	91	62
`kb`	2019	13220	630	620

The two programs `counter` and `tree14` are used for estimating performance improvement of typical concurrent object-oriented programs. They represent the two extreme cases: in `counter`, method invocations are much more frequent than creations of concurrent objects, while in `tree14`, creations of concurrent objects happen as frequently as method invocations do.

Table 2 shows the elapsed time by our analysis for the same set of benchmark programs. We have implemented our analysis with the subtyping relation $\preceq_{NonStr}$, written in Standard ML of New Jersey 0.93 running on SS20 (Hyper SPARC 150 MHz). The first column ("size") shows the sizes of the parse trees of the programs. The second column ("reconst.") and the third column ("solve") show the elapsed times for type reconstruction and constraint solving, respectively. Actually, our implementation integrates the unification phase of constraint solving with type reconstruction, so the third column shows the elapsed times only for solving inequalities on uses. The fourth column ("standard") shows the elapsed times of standard (i.e., without use information) type reconstruction. Since the type system of *HACL* includes the ML-style let-polymorphism, the complexity of type reconstruction is theoretically exponential. Except `kb`, our type reconstruction algorithm is about 10 times slower than the standard one. We believe that it can be made faster by efficient implementation. The cost of solving inequalities is much less than that of type reconstruction.

7 Related Work

Our work has its origin in the I/O channel type system with subtyping proposed by Pierce and Sangiorgi [PS93], and the linear channel type system by Kobayashi, Pierce, and Turner [KPT96]. However, their results have been rather theoretical: they were mainly concerned about checking channel usage and reasoning about program behavior. Since their type reconstruction problems have been open, applications of the type systems to concurrent programming languages have been limited: in [KPT96], although they claimed that linear channels would be potentially useful for program optimization, they have not applied it to actual compilers.

Turner, Wadler, and Mossin [TWM95] proposed a similar static analysis technique for finding use-once values in functional programming languages. In their type system, a use can only be either 1 or ω and much simpler constraints on use variables are used; as a result, if a variable has more than one syntactic occurrence, its use is always inferred to be ω. Therefore, it is not possible to apply their technique directly to the detection of linear channels and it is not trivial either to refine their technique accordingly: notice that a communication channel has normally at least *two* syntactic occurrences (one occurrence for input and the other for output).

Nielson and Nielson [NN95] proposed another technique that can be used for finding some linear channels based on their effect-based analysis [NN94]. However, their analysis is not so effective for detection of linear channels because it counts operations on channels *region-wise*, where a region is a (possibly infinite) set of communication channels. For example, in **def** $f[\,] = m?[x].x![\,] \mid f![\,]$ **in** $f![\,] \mid m![n]$ **end**, the number of output operations performed to the channel x would be counted as ω in their analysis while it is counted as 1 in our type-based analysis. Colby [Col95] also proposed a technique for analyzing communication based on abstract interpretation, which is potentially applicable to the detection of linear channels. However, his analysis would not be effective for the detection of linear channels, either: if it were applied to the detection of linear channels, an infinite number of channels (which are uniquely identified by control paths in the concrete semantics) would be mapped by the abstraction function to the same abstract control path, and therefore his method would give rise to the same problem as mentioned above.

Kobayashi, Nakade and Yonezawa [KNY95] proposed a technique for finding linear channels (and *linearized* channels [KPT96]). However, it is rather complex: as far as linear channels are concerned, our type-based analysis presented here gives *more accurate* results with *much less costs*.

Mercouroff [Mer91] also proposed an algorithm for analyzing communication; however, its target language is very restricted (channels are not first-class values, and moreover, dynamic process creation is not allowed).

8 Conclusions

We have developed a type reconstruction algorithm for a linear channel type system with subtyping of input-only/output-only channel types. Our technique can be used for performing source-level program transformations (such as tail-call optimization) and also for reducing run-time costs of communications; indeed, the analysis (without structural subtyping) has been applied to the compiler of the concurrent language *HACL* and the performance improvement gained by those optimizations has been measured. We believe that the technique proposed here is applicable to other similar concurrent programming languages.

Future work includes further evaluation of the analysis through more realistic (especially distributed) applications. Although this paper focused on channel usage, the usage information about other values (tuples, function closures, etc.) could also be obtained by our analysis [Iga97]; it is left for future work to utilize such information for program optimization or efficient memory management.

Acknowledgements

This work was originally motivated by discussions on linear channels with Benjamin C. Pierce and David N. Turner. Active discussions with them have been of great help to us, especially in finding the right direction at the initial stage of this work. Takayasu Ito carefully read an earlier draft of this paper and gave us a number of useful suggestions. We are also grateful to Kenjiro Taura and Akinori Yonezawa for their comments. Toshihiro Shimizu helped us experiment with his *HACL* compiler. We thank the anonymous referees for their comments, which helped us improve the presentation. Igarashi is a research fellow of the Japan Society of the Promotion of Science.

References

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Col95] Christopher Colby. Analyzing the communication topology of concurrent programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, 1995.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *ACM SIGACT / SIGPLAN Symposium on Principles of Programming Languages*, pages 429–438, 1993.

[Iga97] Atsushi Igarashi. Type-based analysis of usage of values for concurrent programming languages. Master's thesis, University of Tokyo, February 1997.

[IK97] Atsushi Igarashi and Naoki Kobayashi. Type-based analysis of communication for concurrent programming languages. In *Fourth International Static Analysis Symposium (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, September 1997.

[KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag, 1995.

[Kob98] 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.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *ACM SIGACT / SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371, January 1996.

[KY95] Naoki Kobayashi and Akinori Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer-Verlag, 1995.

[Mer91] Nicolas Mercouroff. An algorithm for analyzing communicating processes. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 312–325. Springer-Verlag, 1991.

[Mil93] 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.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *ACM SIGACT / SIGPLAN Symposium on Principles of Programming Languages*, pages 84–97, 1994.

[NN95] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *TAPSOFT'95: Theory and Practice of Software Development*, Lecture Notes in Computer Science, pages 590–604. Springer-Verlag, 1995.

[PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *IEEE Symposium on Logic in Computer Science*, pages 376–385, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

[PS97] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *ACM SIGACT / SIGPLAN Symposium on Principles of Programming Languages*, pages 242–255, Paris, France, January 1997.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997. To appear in Milner *Festschrift*, MIT Press, 1997.

[Rep91] 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.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12, 1965.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 90.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, pages 1–11, San Diego, California, 1995.

[VH93] 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.

[Yon90] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

[YT87] Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, 1987. 282 pages.

A Proof of Subject Reduction Theorem (Theorem 3.2.1)

We need several lemmas to prove Theorem 3.2.1. In what follows, we write $\Gamma_1 \preceq \Gamma_2$ if $\Gamma_1(x) \preceq \Gamma_2(x)$ for any $x \in dom(\Gamma_2)$.

A.1 Lemma: Suppose $\Gamma \vdash P$.

1. If $P = P_1 \mid P_2$, then there exist Γ_1 and Γ_2 such that $\Gamma_i \vdash P_i$ for $i = 1, 2$ and $\Gamma \preceq \Gamma_1 + \Gamma_2$.

2. If $P = P_1 + \cdots + P_n$, then $\Gamma \vdash P_i$ for every i.

3. If $P = x![\tilde{y}]$, then there exist $\tilde{\tau}$ and τ such that $\Gamma \preceq x\colon [\tilde{\tau}]^{(0,\,1)} + y_1\colon \tau_1 + \cdots + y_n\colon \tau_n$.

4. If $P = x?[\tilde{y}].P'$, then there exist Γ' and $\tilde{\tau}$ such that $\Gamma', \tilde{y}\colon\tilde{\tau} \vdash P'$ and $\Gamma \preceq \Gamma' + x\colon[\tilde{\tau}]^{(1,\,0)}$.

5. If $P = \textbf{def } x[\tilde{y}]\colon[\tilde{\tau}]^{(\omega,\,\kappa)} = P_1 \textbf{ in } P_2 \textbf{ end}$, then there exist Γ_i and κ_i (for $i = 1,2$) such that $\Gamma_1, x\colon[\tilde{\tau}]^{(0,\,\kappa_1)}, \tilde{y}\colon\tilde{\tau} \vdash P_1$ and $\Gamma_2, x\colon[\tilde{\tau}]^{(0,\,\kappa_2)} \vdash P_2$ where $\Gamma \preceq (\kappa_2 \cdot (\kappa_1 + 1) \cdot \Gamma_1) + \Gamma_2$ and $\kappa = \kappa_2 \cdot (\kappa_1 + 1)$.

6. If $P = (\nu x\colon\tau)P'$, then $\Gamma, x\colon\tau \vdash P'$.

Proof: Easy. The derivation for each case ends with an application of the typing rule corresponding to the form of P, followed by several consecutive applications of rules T-Sub and T-Weak. Notice that those consecutive applications of T-Sub and T-Weak can be replaced by an application of the following rule:

$$\frac{\Gamma \vdash P \qquad \Gamma' \preceq \Gamma}{\Gamma' \vdash P}$$

For parts 2 and 6, T-Sub and T-Weak may be applied to the premise of the application of T-Choice (or T-New). $\square$

A.2 Lemma: Both $\preceq_{NonStr}$ and $\preceq_{Str}$ satisfy the following conditions:

1. If $\rho_1^{(\kappa_{11},\,\kappa_{12})} \preceq \rho_2^{(\kappa_{21},\,\kappa_{22})}$, then $\kappa_{1i} \geq \kappa_{2i}$ for $i \in \{1,2\}$.
2. If $\kappa_1 \geq \kappa_2$ and $\kappa_3 \geq \kappa_4$, then $\rho^{(\kappa_1,\,\kappa_3)} \preceq \rho^{(\kappa_2,\,\kappa_4)}$.
3. Suppose $\rho_1^{(\kappa_{11},\,\kappa_{12})} \preceq \rho_2^{(\kappa_{21},\,\kappa_{22})}$. If $\kappa_{21} \geq 1$ and $\kappa_{22} \geq 1$, then $\rho_1^{(\kappa_{11}^-,\,\kappa_{12}^-)}$ is well-defined, and $\rho_1^{(\kappa_{11}^-,\,\kappa_{12}^-)} \preceq \rho_2^{(\kappa_{21}^-,\,\kappa_{22}^-)}$.

Proof: Easy from the definitions of $\preceq_{NonStr}$ and $\preceq_{Str}$. $\square$

A.3 Lemma: If $\Gamma, \tilde{y}\colon\tilde{\tau} \vdash P$ and there exist $\tilde{\tau}'$ such that $\tau_i' \preceq \tau_i$ and $\Gamma + z_1\colon\tau_1' + \cdots + z_n\colon\tau_n'$ is well defined, then $\Gamma + z_1\colon\tau_1' + \cdots + z_n\colon\tau_n' \vdash [\tilde{z}/\tilde{y}]P$.

Proof: To prove it, we need to prove the following stronger statement:

> If $\Gamma, z_1\colon\tau_{z_1} + \cdots + z_n\colon\tau_{z_n}, \tilde{y}\colon\tilde{\tau}_y \vdash P$ and there exist $\tilde{\tau}_z'$ and $\tilde{\tau}_y'$ such that $z_1\colon\tau_{z_1}' + \cdots + z_n\colon\tau_{z_n}'$ is well defined and $\tau_{z_i}' \preceq \tau_{z_i}$ and $\tau_{y_i}' \preceq \tau_{y_i}$ where $\tau_{z_i}' + \tau_{y_i}'$ is well defined for $i \in \{1,\ldots,n\}$, then $\Gamma, z_1\colon(\tau_{z_1}' + \tau_{y_1}') + \cdots z_n\colon(\tau_{z_n}' + \tau_{y_n}') \vdash [\tilde{z}/\tilde{y}]P$.

It is proved by straightforward induction on the derivation of $\Gamma, z_1\colon\tau_{z_1} + \cdots + z_n\colon\tau_{z_n}, \tilde{y}\colon\tilde{\tau}_y \vdash P$ using T-Sub in the T-Out and T-In cases. Now the lemma itself is a special case of the statement above. Note that, without loss of generality, we can assume $dom(\Gamma) \supseteq \{\tilde{z}\}$: otherwise we can use T-Weak. $\square$

A.4 Lemma: If $P_1 \cong P_2$, then $\Gamma \vdash P_1$ if and only if $\Gamma \vdash P_2$.

Proof: Straightforward induction on the proof of $P_1 \cong P_2$: the cases for $P_1 \,|\, P_2 \cong P_2 \,|\, P_1$ and $P_1 \,|\, (P_2 \,|\, P_3) \cong (P_1 \,|\, P_2) \,|\, P_3$ follow from commutativity and associativity of the operation $+$ on type environments. The other base cases are also easy, as are the induction steps. $\square$

A.5 Lemma: If $\Gamma \vdash \textbf{def } x[\tilde{y}]\colon\rho^{(\omega,\,\kappa)} = Q \textbf{ in } P \textbf{ end}$ and $P \xrightarrow{x[\tilde{y}]=Q} P'$, then $\Gamma \vdash \textbf{def } x[\tilde{y}]\colon\rho^{(\omega,\,\kappa^-)} = Q \textbf{ in } P' \textbf{ end}$.

Proof: By structural induction on the derivation of $P \xrightarrow{x[\tilde{y}]=Q} P'$ with case analysis by the last rule used. Since the cases of induction steps are easy, we show the only base case R-CALL.

By the assumption, we have

$$\begin{array}{l} P = x![\tilde{z}] \\ P' = [\tilde{z}/\tilde{y}]Q \\ \Gamma \vdash \textbf{def } x[\tilde{y}]\colon \rho^{(\omega,\,\kappa)} = Q \textbf{ in } x![\tilde{z}] \textbf{ end}. \end{array}$$

By Lemma A.1, we obtain

$$\begin{array}{l} \Gamma_1, x\colon [\tilde{\tau}]^{(0,\,\kappa_1)}, \tilde{y}\colon \tilde{\tau} \vdash Q \\ \Gamma_2, x\colon [\tilde{\tau}]^{(0,\,\kappa_2)} \vdash x![\tilde{z}] \\ \kappa = \kappa_2 \cdot (\kappa_1 + 1) \\ \Gamma \preceq \kappa \cdot \Gamma_1 + \Gamma_2. \end{array}$$

Now we have two cases according to the value of κ, which is either 1 or ω. We show the case for $\kappa = 1$. Then, it must be the case that $\kappa_1 = 0$ and $\kappa_2 = 1$ by $\kappa = \kappa_2 \cdot (\kappa_1 + 1)$ and the definitions of $\cdot$ and $+$. Again, by Lemma A.1, we have

$$\Gamma_2, x\colon [\tilde{\tau}]^{(0,\,\kappa_2)} \preceq x\colon [\tilde{\tau}']^{(0,\,1)} + \Gamma'_2$$

where $\Gamma'_2 = z_1\colon \tau'_1 + \cdots + z_n\colon \tau'_n$. Then, whether $\preceq$ is $\preceq_{NonStr}$ or $\preceq_{Str}$, we have $\tau'_i \preceq \tau_i$ for all i since $[\tilde{\tau}]^{(0,\,\kappa_2)} \preceq [\tilde{\tau}']^{(0,\,1)}$. Thus, $\Gamma_2 \preceq z_1\colon \tau_1 + \cdots + z_n\colon \tau_n$. By using Lemma A.3 and rule T-WEAK, we have

$$\Gamma_1 + \Gamma_2, x\colon [\tilde{\tau}]^{(0,\,0)} \vdash [\tilde{z}/\tilde{y}]Q$$

By rule T-DEF, we have

$$(0 \cdot \Gamma_1) + (\Gamma_1 + \Gamma_2) \vdash \textbf{def } x[\tilde{y}]\colon \rho^{(\omega,\,0)} = Q \textbf{ in } [\tilde{z}/\tilde{y}]Q \textbf{ end},$$

finishing the subcase. The other subcase (where $\kappa = \omega$) is similar. □

Proof of Theorem 3.2.1: We prove by structural induction on the proof of $P \xrightarrow{l} P'$ (where l is a variable or ε) with case analysis by the last rule used.

Case R-COMM:

$$\begin{array}{l} l = x \\ P = (\cdots + x?[\tilde{y}].\,Q + \cdots) \mid x![\tilde{z}] \\ P' = [\tilde{z}/\tilde{y}]Q \\ \Gamma, x\colon \rho^{(\kappa_1,\,\kappa_2)} \vdash (\cdots + x?[\tilde{y}].\,Q + \cdots) \mid x![\tilde{z}]. \end{array}$$

By Lemma A.1, we obtain

$$\begin{array}{l} \Gamma'_1, x\colon [\tilde{\tau}]^{(\kappa_{11},\,\kappa_{12})}, \tilde{y}\colon \tilde{\tau} \vdash Q \\ x\colon [\tau_{z_1}, \ldots, \tau_{z_n}]^{(0,\,1)} + \Gamma'_2 \vdash x![\tilde{z}] \\ \Gamma'_2 = z_1\colon \tau_{z_1} + \cdots + z_n\colon \tau_{z_n} \\ \Gamma_1 \preceq \Gamma'_1, x\colon [\tilde{\tau}]^{(\kappa_{11}+1,\,\kappa_{12})} \\ \Gamma_2 \preceq x\colon [\tau_{z_1}, \ldots, \tau_{z_n}]^{(0,\,1)} + \Gamma'_2 \\ \Gamma, x\colon \rho^{(\kappa_1,\,\kappa_2)} \preceq \Gamma_1 + \Gamma_2 \end{array}$$

By Lemma A.2, $\kappa_1 \geq 1$ and $\kappa_2 \geq 1$. We have four cases according to the values of κ_1 and κ_2. We only show the case where $\kappa_1 = \kappa_2 = 1$. By Lemma A.2, we have $\kappa_{11} = \kappa_{12} = 0$. We have $\tau_{z_i} \preceq \tau_i$ for $i \in \{1, \ldots, n\}$ whether $\preceq$ is $\preceq_{NonStr}$ or $\preceq_{Str}$ since $\rho^{(\kappa_1,\,\kappa_2)} \preceq [\tilde{\tau}]^{(\kappa_{11}+1,\,\kappa_{12})}$ and $\rho^{(\kappa_1,\,\kappa_2)} \preceq [\tau_{z_1}, \ldots, \tau_{z_n}]^{(0,\,1)}$. Thus, $\Gamma_2 \setminus \{x\} \preceq z_1\colon \tau_1 + \cdots + z_n\colon \tau_n$. By rule T-WEAK and T-SUB,

$$(\Gamma_1 \setminus \{x\}), x\colon [\tilde{\tau}]^{(0,\,0)}, \tilde{y}\colon \tilde{\tau} \vdash Q.$$

Then, by using Lemma A.3 and rule T-Weak, we have

$$(\Gamma_1 + \Gamma_2) \setminus \{x\}, x\colon [\tilde{\tau}]^{(0,\,0)} \vdash [\tilde{z}/\tilde{y}]Q$$

Finally, rules T-Sub and T-Weak finishes the subcase.

The other three subcases are similar.

Case R-Call**:**

This case is excluded because the label l is not $x[\tilde{y}] = P$.

Case R-Cong**:**

It follows from Lemma A.4.

Case R-Par**:**

We show the case where l is a variable x. By assumption, we have $\Gamma, x\colon \rho_1^{(\kappa_1,\,\kappa_2)} \vdash P_1 \,|\, P_2$. By Lemma A.1, there exist some Γ_1 and Γ_2 such that $\Gamma_1, x\colon \rho_2^{(\kappa_3,\,\kappa_4)} \vdash P_1$ and $\Gamma_2 \vdash P_2$ where $\Gamma, x\colon \rho_1^{(\kappa_1,\,\kappa_2)} \preceq (\Gamma_1, x\colon \rho_2^{(\kappa_3,\,\kappa_4)}) + \Gamma_2$. By induction hypothesis, $\Gamma_1, x\colon \rho_2^{(\kappa_3^-,\,\kappa_4^-)} \vdash P_1'$. By Lemma A.2, it is easy to show that $\Gamma, \rho_1^{(\kappa_1^-,\,\kappa_2^-)} \preceq (\Gamma_1, x\colon \rho_2^{(\kappa_3^-,\,\kappa_4^-)}) + \Gamma_2$. Thus, $\Gamma, \rho_1^{(\kappa_1^-,\,\kappa_2^-)} \vdash P_1' \,|\, P_2$ finishes the case.

Case R-Def1**:**

It follows from Lemma A.5.

Case R-New1, R-New2, R-Def2**:**

These cases are easy. □

B Proof of Theorem 4.2.5

To prove Theorem 4.2.5, we need several lemmas. The three lemmas below ensure that the auxiliary functions $\bigoplus$, $\bigsqcup$, and $\bigodot$ output the most general type environment and subtyping constraint. Intuitively, we would expect $\Gamma_1 \bigoplus \Gamma_2$ outputs the most general Γ and C such that $C \models \Gamma \preceq \Gamma_1 + \Gamma_2$, but since it is not always the case that $\Gamma_1 + \Gamma_2$ is well defined, we may not have such a pair; instead of taking a summation of $\Gamma_1 + \Gamma_2$ directly, we introduce extra type environments Γ_1' and Γ_2' such that their summation is well defined and each of them satisfies $C \models \Gamma_i' \preceq \Gamma_i$. Then, Γ and C are the most general pair which satisfies $C \models \Gamma \preceq \Gamma_1' + \Gamma_2'$.

B.1 Lemma: Suppose $(\Gamma, C) = \Gamma_1 \bigoplus \Gamma_2$. Then, $C \models (\Gamma \preceq \Gamma_1'' + \Gamma_2'') \cup (\Gamma_1'' \preceq \Gamma_1) \cup (\Gamma_2'' \preceq \Gamma_2)$ for some Γ_1'' and Γ_2''. Moreover, if $C' \models \Gamma' \preceq \Gamma_1' + \Gamma_2'$ and $\Gamma_i' \supseteq S\Gamma_i$ (for $i \in \{1,2\}$) for some $S, \Gamma', \Gamma_1', \Gamma_2'$, and C', then there exists a substitution S' such that $\Gamma' \supseteq S'\Gamma$ and $C' \models S'C$ and $S' \supseteq S$, where $S' \supseteq S$ denotes $dom(S') \supseteq dom(S)$ and, for any type/use variable α (and j) in the domain of S, $S(\alpha) = S'(\alpha)$ (and $S(j) = S'(j)$).

Proof: Let Γ_{11}'' and Γ_{21}'' be respectively Γ_1' and Γ_2' in the definition of $\bigoplus$, and Γ_{12}'' and Γ_{22}'' be $\Gamma_1 \setminus dom(\Gamma_{11}'')$ and $\Gamma_2 \setminus dom(\Gamma_{21}'')$, respectively, and Γ_i'' be $\Gamma_{i1}'' \cup \Gamma_{i2}''$ for $i \in \{1,2\}$. Then, the first part $C \models (\Gamma \preceq \Gamma_1'' + \Gamma_2'') \cup (\Gamma_1'' \preceq \Gamma_1) \cup (\Gamma_2'' \preceq \Gamma_2)$ is trivial.

Now, let S'' be a substitution such that $dom(S'')$ is type/use variables in Γ, Γ_{11}'' and Γ_{21}'', and $S''\Gamma_{i1}'' \subseteq \Gamma_i'$ for $i \in \{1,2\}$ and $S''\Gamma \subseteq \Gamma'$. Such a substitution always exists since every type/use variable in $\Gamma_{11}'', \Gamma_{21}''$ and Γ is fresh. Since $dom(S''S\Gamma_i'') = dom(\Gamma_i'') = dom(\Gamma_i) = dom(S''S\Gamma_i)$ and $S''S\Gamma_i'' \subseteq \Gamma_i'$ and $S''S\Gamma_i = S\Gamma_i \subseteq \Gamma_i'$, we have $S''S\Gamma_i'' = S''S\Gamma_i$ for $i \in \{1,2\}$. Note that the domain of S'' includes only fresh variables. Then, it is easy to show that $C' \models \Gamma' \preceq \Gamma_1' + \Gamma_2'$ implies $C' \models S''S(\Gamma \preceq \Gamma_1'' + \Gamma_2'')$, which is equivalent to $C' \models S''S(\Gamma \preceq \Gamma_1'' + \Gamma_2'') \cup S''S(\Gamma_1'' \preceq \Gamma_1) \cup S''S(\Gamma_2'' \preceq \Gamma_2)$. Letting $S' = S''S$ finishes the proof. □

B.2 Lemma: Suppose $\Gamma_1 \bigsqcup \Gamma_2 = (\Gamma, C)$. Then, $C \models (\Gamma \preceq \Gamma_1) \cup (\Gamma \preceq \Gamma_2)$. Moreover, if there exist $S', \Gamma', \Gamma'_1, \Gamma'_2$, and C' such that $C' \models (\Gamma' \preceq \Gamma'_1) \cup (\Gamma' \preceq \Gamma'_2)$, and $\Gamma'_i \supseteq S'\Gamma_i$ (for $i \in \{1,2\}$), then $C' \models SC$ and $\Gamma' \supseteq S'\Gamma$ for some S.

B.3 Lemma: Suppose $\kappa \bigodot \Gamma_1 = (\Gamma_2, C)$. Then, $C \models \Gamma_2 \preceq \kappa \cdot \Gamma_1$. Moreover, if there exist $\Gamma'_1, \Gamma'_2, C', \kappa'$, and S' such that $C' \models \Gamma'_2 \preceq \kappa' \cdot \Gamma'_1$ and $\Gamma'_1 \supseteq S'\Gamma_1$ and $\kappa' = S'\kappa$, then $\Gamma'_2 \supseteq S\Gamma_2$ and $C' \models SC$ for some S.

Proofs of Lemmas B.2 and B.3 are similar to that of Lemma B.1.

B.4 Lemma: If $\Gamma; C \vdash_{\mathcal{STR}} P$ and $C' \models C$ and $C' \models \Gamma' \preceq \Gamma$, then $\Gamma'; C' \vdash_{\mathcal{STR}} P$.

Proof: By induction on the derivation of $\Gamma; C \vdash_{\mathcal{STR}} P$ using transitivity of $\models$ and the fact that $C \models \Gamma_1 \preceq \Gamma_2$ and $C \models \Gamma_2 \preceq \Gamma_3$ implies $C \models \Gamma_1 \preceq \Gamma_3$. □

Proof of Theorem 4.2.5: Structural induction on the derivation of $\Gamma'; C' \vdash_{\mathcal{STR}} SP$, with case analysis on the last rule used. We show only a few cases because the other cases are similar.

Case ST-PAR: $P = P_1 \mid P_2$ $\quad \Gamma'_1; C'_1 \vdash_{\mathcal{STR}} SP_1$ $\quad \Gamma'_2; C'_2 \vdash_{\mathcal{STR}} SP_2$
$C' \models \Gamma' \preceq \Gamma'_1 + \Gamma'_2$ $\quad C' \models C'_1$ $\quad C' \models C'_2$

We first show $\Gamma; C \vdash_{\mathcal{STR}} P$ where $(\Gamma, C) = PTU(P)$. Let $(\Gamma_i, C_i) = PTU(P_i)$ for $i \in \{1,2\}$. By induction hypothesis, $\Gamma_i; C_i \vdash_{\mathcal{STR}} P_i$ for $i \in \{1,2\}$. Then, let $(\Gamma, C) = \Gamma_1 \bigoplus \Gamma_2$. By Lemma B.1, (Γ, C) satisfies

$$C \models (\Gamma \preceq \Gamma''_1 + \Gamma''_2) \cup (\Gamma''_1 \preceq \Gamma_1) \cup (\Gamma''_2 \preceq \Gamma_2)$$

for some Γ''_1 and Γ''_2. By Lemma B.4, $\Gamma''_i; C \cup C_1 \cup C_2 \vdash_{\mathcal{STR}} P_i$ for $i \in \{1,2\}$. By ST-PAR, we have $\Gamma; C \cup C_1 \cup C_2 \vdash_{\mathcal{STR}} P$.

Now, we must show that there exists a substitution S' such that $C' \models S'(C \cup C_1 \cup C_2)$ and $\Gamma'_i \supseteq S'\Gamma_i$ where $SP = S'P$. By induction hypothesis, there exists a substitution S_i such that $SP_i = S_iP_i$ and $C'_i \models S_iC_i$, and $\Gamma'_i \supseteq S_i\Gamma_i$ for $i \in \{1,2\}$. Since the domains of S_1 and S_2 are disjoint, and each Γ_i includes only fresh type/use variables, $\Gamma'_i \supseteq (S_1 \cup S_2)\Gamma_i$ for $i \in \{1,2\}$, where $S_1 \cup S_2$ stands for the union of two mappings whose domains are disjoint. By Lemma B.1, (Γ, C) satisfies

$$\Gamma' \supseteq S''\Gamma$$
$$C' \models S''C$$
$$S'' \supseteq S_1 \cup S_2$$

for some S''. Letting $S' = S''$ finishes the case since $S'P = SP$ and $\Gamma' \supseteq S'\Gamma$ and $C' \models S'(C \cup C_1 \cup C_2)$, which is equivalent to $C' \models S''C \cup S_1C_1 \cup S_2C_2$. Note that $dom(S') \backslash dom(S_1 \cup S_2)$ is disjoint from type variables in $C_1 \cup C_2$: in the proof of Lemma B.1, S' is constructed by extending $S_1 \cup S_2$ with a mapping from fresh variables.

Case ST-NEW: $P = (\nu x{:}\, \tau_x)P'$ $\quad \Gamma', x{:}\, \tau'_x; C' \vdash_{\mathcal{STR}} S'P'$ $\quad C' \models S'\tau_x \preceq \tau'_x$.

We first show $\Gamma; C \vdash_{\mathcal{STR}} P$ where $(\Gamma, C) = PTU(P)$. Let $(\Gamma'', C'') = PTU(P')$. By induction hypothesis, $\Gamma''; C'' \vdash_{\mathcal{STR}} P'$. We show the subcase where $x \in dom(\Gamma'')$ here. Since $C'' \cup \{\tau_x \preceq \Gamma''(x)\} \models \tau_x \preceq \Gamma''(x)$, we have

$$(\Gamma'' \setminus \{x\}); C'' \cup \{\tau_x \preceq \Gamma''(x)\} \vdash_{\mathcal{STR}} (\nu x{:}\, \tau_x)P'$$

by rule ST-NEW. Now, we must show there exists S such that $C' \models S(C'' \cup \{\tau_x \preceq \Gamma''(x)\})$ and $\Gamma' \supseteq S(\Gamma'' \setminus \{x\})$. By induction hypothesis, we have S'' such that $C' \models S''C''$ and $\Gamma', x{:}\, \tau'_x \supseteq S''\Gamma''$. We have S by extending S'' so that $S\tau_x = S'\tau_x$; it is possible since type/use variables in τ_x are distinct from any other variables. Finally, $C' \models SC'' \cup \{S\tau_x \preceq S\Gamma''(x)\}$ and $\Gamma' \supseteq S(\Gamma'' \setminus \{x\})$.

The other subcase, where $x \notin dom(\Gamma)$, is similar. □

C Proof of Lemma 5.3.20

First, we prove termination of the rewriting system by introducing a well-founded order $>$ on quadruples $(C, \Theta_1, \Theta_2, S)$. We begin with several preliminary definitions. The size $|C|$ of an extended subtyping constraint is defined as follows:

$$
\begin{array}{lcl}
|\emptyset| & = & 0 \\
|C \uplus \{\tau_1 \preceq \tau_2\}| & = & |C| + |\tau_1| + |\tau_2| \\
|C \uplus \{\kappa \Rightarrow \tau_1 \preceq \tau_2\}| & = & |C \uplus \{\tau_1 \preceq \tau_2\}| + 1 \\
|\alpha^{(\kappa_1,\, \kappa_2)}| & = & 1 \\
|[\tilde{\tau}]^{(\kappa_1,\, \kappa_2)}| & = & (|\tau_1| + \cdots + |\tau_n|) + n + 1
\end{array}
$$

We extend $\geq$ between uses pointwise to substitutions of uses: $S_1 \geq S_2$ if and only if $dom(S_1) = dom(S_2)$, and $S_1 j \geq S_2 j$ for any use variable $j \in dom(S_2)$. The proper order $S_1 > S_2$ on substitutions of uses for use variables is defined by: $S_1 > S_2$ if and only if $S_1 \geq S_2$ and $S_1 \neq S_2$. Finally, $(C_1, \Theta_{11}, \Theta_{12}, S_1) > (C_2, \Theta_{21}, \Theta_{22}, S_2)$ if and only if either (1) $S_2 > S_1$, (2) $S_1 = S_2$ and $|C_1| > |C_2|$, or (3) $S_1 = S_2$ and $|C_1| = |C_2|$ and $\Theta_{11} \supset \Theta_{21}$.

C.1 Lemma: The set of quadruples $(C, \Theta_1, \Theta_2, S)$ is well-founded under $>$.

Proof: $>$ is a lexicographic product of well-founded orderings $>$ on S, $>$ on integers and $\supset$ on Θ. □

C.2 Lemma: The rewriting system $((C, \Theta_1, \Theta_2, S), \rightsquigarrow)$ is strong normalizing.

Proof: It is easy to show that if $(C, \Theta_1, \Theta_2, S) \rightsquigarrow (C', \Theta'_1, \Theta'_2, S')$, then $(C, \Theta_1, \Theta_2, S) > (C', \Theta'_1, \Theta'_2, S')$ by inspecting the rules for $\rightsquigarrow$. Note that, for the third rule, we have

$$
\begin{array}{cl}
 & |C \uplus \{[\tilde{\tau}]^{(\kappa_{11},\, \kappa_{12})} \preceq [\tilde{\tau'}]^{(\kappa_{21},\, \kappa_{22})}\}| \\
= & |C| + (|\tau_1| + \cdots + |\tau_n| + n + 1) + (|\tau'_1| + \cdots + |\tau'_n| + n + 1) \\
> & |C| + (|\tau_1| + \cdots + |\tau_n|) + (|\tau'_1| + \cdots + |\tau'_n|) + 2n \\
= & |C \uplus \{\kappa_{21} \Rightarrow \tau_i \preceq \tau'_i \,|\, 1 \leq i \leq n\} \cup \{\kappa_{22} \Rightarrow \tau'_i \preceq \tau_i \,|\, 1 \leq i \leq n\}|
\end{array}
$$

Therefore, strong normalization follows from well-foundedness of $>$. □

Rewriting steps preserve the following properties:

C.3 Lemma: Suppose $(C, \Theta_1, \Theta_2, S) \rightsquigarrow^* (C', \Theta'_1, \Theta'_2, S')$.

(1) If C is a matching constraint that contains no type variables and, for any constraint expression $\tau \preceq \rho^{(\kappa_1, \kappa_2)} \in C$, every use in τ and ρ is either a use variable or constant, and, for any $\kappa_1 \geq \kappa_2 \in \Theta_1$, the left-hand side κ_1 is either a use variable or constant, then C' and Θ'_1 satisfy the same conditions.

(2) S'' is a solution of $C \cup \Theta_1 \cup \Theta_2 \cup \{j \geq Sj \mid j \in dom(S)\}$ iff S'' is a solution of $C' \cup \Theta'_1 \cup \Theta'_2 \cup \{j \geq S'j \mid j \in dom(S')\}$.

(3) if S is a solution of Θ_2, then S' is a solution of Θ'_2.

Proof: By induction on the length of the rewriting steps with inspection of the rewriting rules. □

Proof of Lemma 5.3.20: By Lemma C.2, $(C, \emptyset, \emptyset, [0/j_1, \ldots, 0/j_n])$ rewrites to a normal form $(C', \Theta_1, \Theta_2, S)$. If Θ_1 is empty, then S is the least solution of $C' \cup \Theta_2 \cup \{j \geq Sj \mid j \in dom(S)\}$ since C' includes a constraint expression of the form $\kappa \Rightarrow \tau_1 \preceq \tau_2$ where $S\kappa = 0$ (otherwise the quadruple is not a normal form), and S is a solution of Θ_2 by Lemma C.3 (3). By Lemma C.3 (2), S is the least solution of C. On the other hand, if Θ_1 is not empty, then S is not a solution of Θ since Θ_1 includes only constraints of the form $c \geq \kappa$ (by Lemma C.3 (1)) such that $c \not\geq S\kappa$ (otherwise the quadruple is not a normal form); by Lemma C.3 (2), C has no solutions. $\square$

D Benchmark Programs

This section shows some of the programs used in the experiments of Section 6. The benchmark programs are translated to those written in the target language of this paper (extended with several mathematical operations, conditional expressions, and records). In the programs below, *output* is a special channel which represents an output device such as a display, $\{l_1 = x_1, \ldots, l_n = x_n\}$ constructs a record of $x_1, \ldots, x_n$ with field names $l_1, \ldots, l_n$, respectively, and $x.l$ selects a field l from a record x.

D.1 Fibonacci Functions

Naive and optimized sequential fibs are described as the example of our analysis (Section 5.4). Naive parallel fib:

$$
\begin{array}{l}
\textbf{def } \mathit{pfib}[n, c] = \\
\quad \textbf{if } n < 2 \textbf{ then } c![1] \\
\quad \textbf{else } (\nu c_1)(\nu c_2)(\mathit{pfib}![n-1, c_1] \mid \mathit{pfib}![n-2, c_2] \mid c_1?[x].c_2?[y].c![x+y]) \\
\textbf{in } \mathit{pfib}![25, \mathit{output}] \textbf{ end}
\end{array}
$$

is optimized to the following expression:

$$
\begin{array}{l}
\textbf{def } \mathit{pfibopt}(n, c) = \\
\quad \textbf{if } n < 2 \textbf{ then } c![1] \\
\quad \textbf{else } (\nu c_2)(\textbf{def } c_1[x] = c_2[y].c![x+y] \textbf{ in} \\
\qquad \mathit{pfibopt}![n-1, c_1] \mid \mathit{pfibopt}![n-2, c_2] \textbf{ end}) \\
\textbf{in } \mathit{pfibopt}![25, \mathit{output}] \textbf{ end}
\end{array}
$$

D.2 Counter Objects

The naive implementation of a counter object is:

$$
\begin{array}{l}
\textbf{def } \mathit{newcounter}[\mathit{init}, c] = \\
\quad (\nu st)(st![\mathit{init}] \\
\qquad \mid \textbf{def } g[c_1] = st?[x].(c_1![x] \mid st![x]) \textbf{ in} \qquad \textit{(* get the state of the counter *)} \\
\qquad\quad \textbf{def } i[\mathit{ack}] = st?[x].(\mathit{ack}![\,] \mid st![x+1]) \textbf{ in} \qquad \textit{(* increment the counter *)} \\
\qquad\qquad c![\{\mathit{get} = g, \mathit{inc} = i\}] \textbf{ end end}) \\
\textbf{in def } \mathit{incr}[n, o, \mathit{ack}] = \qquad \textit{(* invoke inc method n times *)} \\
\quad \textbf{if } n = 0 \textbf{ then } \mathit{ack}![\,] \\
\quad \textbf{else } (\nu r)(o.\mathit{inc}![r] \mid r?[\,].(\mathit{incr}![n-1, o, \mathit{ack}])) \\
\textbf{in } (\nu r_1)(\mathit{newcounter}![0, r_1] \\
\qquad \mid r_1?[o].(\nu r_2)(\mathit{incr}![10000, o, r_2] \mid r_2?[\,].(o.\mathit{get}![\mathit{output}]))) \\
\textbf{end end}
\end{array}
$$

Then, the above program is optimized to:

$$
\begin{array}{l}
\textbf{def } \mathit{newcounter}[\mathit{init}, c] = \\
\quad (\nu st)(st![\mathit{init}] \\
\qquad |\, \textbf{def } g[c_1] = st?[x].(c_1![x] \,|\, st![x]) \textbf{ in} \\
\qquad\quad \textbf{def } i[\mathit{ack}] = st?[x].(\mathit{ack}![\,] \,|\, st![x+1]) \textbf{ in} \\
\qquad\qquad\qquad c![\{\mathit{get} = g, \mathit{inc} = i\}] \textbf{ end end}) \\
\textbf{in def } \mathit{incr}[n, o, \mathit{ack}] = \\
\qquad \textbf{if } n = 0 \textbf{ then } \mathit{ack}![\,] \\
\qquad \textbf{else def } r[\,] = \mathit{incr}![n-1, o, \mathit{ack}] \textbf{ in } o.\mathit{inc}![r] \textbf{ end} \\
\textbf{in def } r_1[o] = \textbf{def } r_2[\,] = o.\mathit{get}![\mathit{output}] \textbf{ in } \mathit{incr}![10000, o, r_2] \textbf{ end} \\
\qquad\qquad\qquad \textbf{in } \mathit{newcounter}![0, r_1] \textbf{ end} \\
\textbf{end end}
\end{array}
$$

D.3 Tree Summation

The naive implementation of tree object is:

$$
\begin{array}{l}
\textbf{def } \mathit{newnode}[n, \mathit{left}, \mathit{right}, \mathit{rep}] = \qquad \textit{(* } n = 0 \textit{ : node object, } n \neq 0 \textit{ : leaf object holding } n \textit{ *)} \\
\quad (\nu st)(st![n, \mathit{left}, \mathit{right}] \\
\qquad |\, \textbf{def } s[c] = \\
\qquad\qquad st?[n, l, r].(st![n, l, r] \\
\qquad\qquad\qquad |\, \textbf{if } n <> 0 \textbf{ then } c![n] \\
\qquad\qquad\qquad\quad \textbf{else } (\nu r_1)(\nu r_2)(r.\mathit{sum}![r_1] \,|\, l.\mathit{sum}![r_2] \,|\, r_1?[x_1].r_2?[x_2].c![x_1 + x_2])) \\
\qquad \textbf{in } \mathit{rep}![\{\mathit{sum} = s\}] \textbf{ end}) \\
\textbf{in def } \mathit{gentree}[n, \mathit{rep}] = \\
\qquad \textbf{if } n = 0 \textbf{ then } (\nu l)(\nu r)\mathit{newnode}![1, l, r, \mathit{rep}] \\
\qquad \textbf{else } (\nu r_1)(\nu r_2)(\mathit{gentree}![n-1, r_1] \,|\, \mathit{gentree}![n-1, r_2] \\
\qquad\qquad\qquad |\, r_1?[\mathit{left}].r_2?[\mathit{right}].\mathit{newnode}![0, \mathit{left}, \mathit{right}, \mathit{rep}]) \\
\textbf{in } (\nu r)(\mathit{gentree}[14, r] \,|\, r?[o].(o.\mathit{sum}![\mathit{output}])) \textbf{ end end}
\end{array}
$$

Then, the optimized program is:

$$
\begin{array}{l}
\textbf{def } \mathit{newnode}[n, \mathit{left}, \mathit{right}, \mathit{rep}] = \\
\quad (\nu st)(st![n, \mathit{left}, \mathit{right}] \\
\qquad |\, \textbf{def } s[c] = \\
\qquad\qquad st?[n, l, r].(st![n, l, r] \\
\qquad\qquad\qquad |\, \textbf{if } n <> 0 \textbf{ then } c![n] \\
\qquad\qquad\qquad\quad \textbf{else } (\nu r_2) \textbf{ def } r_1[x_1] = r_2?[x_2].c![x_1 + x_2] \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } r.\mathit{sum}![r_1] \,|\, l.\mathit{sum}![r_2] \textbf{ end} \\
\qquad \textbf{in } \mathit{rep}![\{\mathit{sum} = s\}] \textbf{ end)} \\
\textbf{in def } \mathit{gentree}[n, \mathit{rep}] = \\
\qquad \textbf{if } n = 0 \textbf{ then } (\nu r_1)(\nu r_2)\mathit{newnode}![1, r_1, r_2, \mathit{rep}] \\
\qquad \textbf{else } (\nu r_2)(\textbf{def } r_1[\mathit{left}] = c_2?[\mathit{right}].\mathit{newnode}![0, \mathit{left}, \mathit{right}, \mathit{rep}] \\
\qquad\qquad\qquad \textbf{in } \mathit{gentree}![n-1, r_1] \,|\, \mathit{gentree}![n-1, r_2] \textbf{ end}) \\
\textbf{in def } r[o] = o.\mathit{sum}![\mathit{output}] \textbf{ in } \mathit{gentree}[14, r] \textbf{ end end}
\end{array}
$$