TECHNICAL REPORT 97-03

Type-based Analysis of Communication for Concurrent Programming Languages

Atsushi Igarashi Naoki Kobayashi

June 1997

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TECHNICAL REPORT 97-03

TITLE
Type-based Analysis of Communication for Concurrent Programming Languages

AUTHORS
Atsushi Igarashi Naoki Kobayashi

KEY WORDS AND PHRASES
linear type, linear channel, static analysis, concurrent language

ABSTRACT
Powerful concurrency primitives in recent concurrent languages and thread libraries provide the 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 aggressive code optimization. We propose a static analysis method for inferring how many times each communication channel is used during execution of concurrent programs: a type system that is sensitive to usage information is constructed for this purpose and a type reconstruction algorithm is developed. Our analysis can, in particular, automatically detect *linear channels* (communication channels used just once): as studied by Kobayashi, Pierce, and Turner, they are very useful for reasoning about program behavior and aggressive code optimization. Our analysis has already been implemented and applied to the compiler of a concurrent language HACL; we present the results of simple benchmarks to show performance improvement gained by our analysis.

REPORT DATE	**WRITTEN LANGUAGE**
June 1997	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
28	19

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
A summary of this paper will appear in Proceedings of the 1997 International Static Analysis Symposium.

DISTRIBUTION STATEMENT
First issue 40 copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Type-based Analysis of Communication for Concurrent Programming Languages

Atsushi Igarashi Naoki Kobayashi

Department of Information Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan
TEL: +81-3-5800-6913
FAX: +81-3-5689-4365
{igarashi, koba}@is.s.u-tokyo.ac.jp

1 Introduction

Background. 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 information about their behavior may be lost and as a result, it is difficult to check global correctness of programs or to perform aggressive code optimization.

Linear Types for Concurrent Languages. In order to overcome the above problems, a number of techniques have been recently proposed for analyzing the communication topology [10, 2], checking usage of communication channels [12, 6], detecting deadlock [4], etc. Among them, the linear channel type system [6], which guarantees that certain channels are used at most once, is potentially useful both for ensuring the correct program behavior and for optimizing concurrent programs. In order to illustrate the ideas, we consider the following asynchronous process calculus:

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

For example, the process $x?[z].\, y![z]$ receives a value along the channel x and then forwards it to the channel y. Earlier type systems for concurrent languages (including CML [14]) were only concerned with the type of values transmitted along channels; so $x?[z].\, y![z]$ has been typed as:

$$x : [\tau], y : [\tau] \vdash x?[z].\, y![z],$$

where $[\tau]$ denotes the type of channels used for transmitting values of type τ.

The idea of the linear type system is to annotate channel type constructors with information about how often channels can be used for input or output (we use here a slightly different formalization from the original linear type system [6].); so, the above type judgement is replaced by:

$$x : [\tau]^{(1,0)}, y : [\tau]^{(0,1)} \vdash x?[z].\, y![z].$$

The superscript pair (κ_1, κ_2) of integers or ω (infinity) specifies how often a channel can be used for input (by κ_1) and for output (by κ_2). (Throughout the paper, '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.) Let us consider a more complex example:

$$f : [int, []^{(0,1)}]^{(0,1)}, r : []^{(1,1)}, r' : []^{(0,1)}, n : int \vdash f![n, r] \,|\, r?[].\, r'![].$$

The type of f indicates that f can be used at most once for sending a pair of an integer and a channel and also that the channel may be used by the receiver on f at most once for sending a null tuple. By using such type information, we can safely replace the process $(\nu r)\,(f![n, r] \,|\, r?[].\, r'![])$ with the more efficient process $f![n, r']$. As we have argued elsewhere [5, 6], this optimization corresponds to tail-call optimization of functions.

However, the previous linear type system [6] has been only concerned with *checking* usage information; so programmers must put heavy annotations in order to make optimization work. The main reason for it is that the type system apparently has no principal typing property: in fact, there are lots of other possible typings for the above process:

$$\begin{array}{l} f : [int, []^{(1,0)}]^{(0,1)}, r : []^{(2,0)}, r' : []^{(0,1)}, n : int \vdash f![n, r] \,|\, r?[].\, r'![] \\ f : [int, []^{(1,1)}]^{(0,1)}, r : []^{(2,1)}, r' : []^{(0,1)}, n : int \vdash f![n, r] \,|\, r?[].\, r'![] \\ \cdots \end{array}$$

Our Goal and Approach. The main goal of the present paper was to refine the above type system so that usage information can be *automatically* inferred. The key idea to achieve that goal is to introduce *use variables* and constraint on them, so that *partial* information on channel usage can be expressed. The new type judgement is of the form $\Gamma; \Theta \vdash P$ where Γ is a mapping from variables to types annotated with use information, and Θ is a set (which we call a *use constraint set*) of constraints on use variables. The two processes given above are typed in our new type system as follows:

$$\begin{array}{l} x : [\tau]^{(j_x, k_x)}, y : [\tau]^{(j_y, k_y)}; \{j_x \geq 1, k_y \geq 1\} \vdash x?[z].\, y![z] \\ f : [int, []^{(j,k)}]^{(j_f, k_f)}, r : []^{(j_r, k_r)}, r' : []^{(j_{r'}, k_{r'})}, n : int; \\ \qquad \{j_r \geq j + 1, k_r \geq k, k_f \geq 1, k_{r'} \geq 1\} \vdash f![n, r] \,|\, r?[].\, r'![] \end{array}$$

The use constraint sets specify the values taken by use variables $(j, k, \ldots)$. In the process $f![n, r] \,|\, r?[].\, r'![]$, r is used once for input by $r?[].\, r'![]$ and it may also be used by the receiver on f at most j times for input; therefore, as specified by the inequation $j_r \geq j + 1$, the total number j_r of input uses of r may be greater than or equal to $j + 1$. It can be shown that both of the above type judgements are *principal* in the sense that they have all the possible typings as their instances. If we let use variables to range over a finite set of values, the least solution of a use constraint set can always be computed efficiently. (In fact, here we let them range over the set $\{0, 1, \omega\}$: as we have argued elsewhere [6], few benefits would be gained by introducing other values: $2, 3, \ldots$)

Applications. Applications of our analysis include:

(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; moreover, as shown in a later section, if we implement functions or concurrent objects in terms of concurrency primitives, similar optimization can also be applied to most of the function or method calls.

(2) Reduction of the cost for communication: we can optimize run-time representation of a used-once channel (which we call a *linear channel*) and also reduce run-time check of its state (and sometimes we can allocate it in 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 [14], Pict [13], and HACL [7] 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 and Overview of the Paper. 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 a polynomial time algorithm to infer usage information, which consists of an algorithm to compute the principal typing and an algorithm to solve a use constraint set, and (3) evaluation of our analysis via simple benchmarks. For clarity and space restriction, we use here a pure process calculus as a target language; however, we believe that our analysis is applicable to many other concurrent languages [14, 19, 18, 13, 7]: in fact, we have already incorporated our analysis 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 [3]). As will be sketched in Section 7, our type system can be extended with subtyping to improve the accuracy of our analysis and it still has a type reconstruction algorithm; it would be worth mentioning that the algorithm can be viewed as a type reconstruction algorithm for a variant of the input/output channel type with subtyping [12], for which no complete algorithm has been known to our knowledge.

The rest of this paper is organized as follows. Section 2 introduces the syntax of our target language. After presenting typing rules in Section 3, we give a type reconstruction algorithm and show how to detect linear channels in Section 4. Section 5 shows the correctness of our type judgement on usage information with respect to operational semantics. Section 6 reports experimental results of applying our analysis to compile-time optimizations of concurrent programs. Section 7 discusses an extension of our type system with subtyping. After discussing related work in Section 8, we conclude in Section 9.

2 Notational Preliminaries

In this section, we introduce the syntax of types, process expressions, and type judgements. The target language can be considered an asynchronous fragment of the polyadic π-calculus [9], and it is close to the core languages of HACL [7] and Pict [13].

2.1 Types with Uses

As already mentioned, *uses* are associated to the channel type constructor and denote how many times each channel is used. The set of *uses*, ranged over by κ, is defined by:

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

The metavariable $j(, k, l, \ldots)$ ranges over a countably infinite set of *use variables*. $0, 1$, and ω are often called *use constants*. The use constant 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. Expressions $\kappa_1 + \kappa_2$, $\kappa_1 \sqcup \kappa_2$, and $\kappa_1 \cdot \kappa_2$ are called *summation*, *upper-bound*, and *product*, respectively. The syntax of types is defined as follows.

Definition 2.1 (bare types, types) The set of *bare types*, ranged over by ρ, and the set of *types*, ranged over by τ, are given by the following syntax:

$$\begin{array}{rcll} \rho & ::= & \alpha & (\textit{type variables}) \\ & \mid & \textit{bool} & (\textit{boolean type}) \\ & \mid & [\tau_1, \ldots, \tau_n] & (\textit{channel types}) \\ \tau & ::= & \rho^{(\kappa_1, \kappa_2)} & \end{array}$$

The metavariable $\alpha(, \beta, \ldots)$ ranges over a countably infinite set of *type variables*. 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* (*output*, resp.) *use*, denotes how often the channel is used for input (output, resp.). For example, $[\tau]^{(0,1)}$ denotes the type of channels used *at most once* for sending a value of type τ.

For the convenience of type reconstruction, we attach uses also to the boolean type; since they are not important, we always assume that they are ω and often write *bool* for $\textit{bool}^{(\omega,\omega)}$.

Several operations on use constants and types are defined below.

Definition 2.2 The relation $\geq$ between use constants is the total order defined by $\omega \geq 1 \geq 0$. We extend $\geq$ to a partial order between types: $\rho_1^{(\kappa_1,\kappa_2)} \geq \rho_2^{(\kappa_3,\kappa_4)}$ if and only if $\rho_1 = \rho_2$, $\kappa_1 \geq \kappa_3$, and $\kappa_2 \geq \kappa_4$.

We often write $\tilde{\tau} \geq \tilde{\tau}'$ instead of $\tau_1 \geq \tau_1', \ldots, \tau_n \geq \tau_n'$.

Definition 2.3 The binary operator $+$ on use constants, called *summation*, 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: $\rho^{(\kappa_1,\kappa_2)} + \rho^{(\kappa_3,\kappa_4)} = \rho^{(\kappa_1+\kappa_3,\kappa_2+\kappa_4)}$.

Definition 2.4 The binary operator $\sqcup$ on use constants, called *upper-bound*, is the commutative and associative operation defined by: $\kappa_1 \sqcup \kappa_2 = \kappa_1$ if $\kappa_1 \geq \kappa_2$, and $\kappa_1 \sqcup \kappa_2 = \kappa_2$ otherwise. The upper-bound of two types, written $\tau_1 \sqcup \tau_2$, is defined only when their bare types are identical: $\rho^{(\kappa_1,\kappa_2)} \sqcup \rho^{(\kappa_3,\kappa_4)} = \rho^{(\kappa_1 \sqcup \kappa_3, \kappa_2 \sqcup \kappa_4)}$.

Definition 2.5 The binary operator $\cdot$ on use constants, called *product*, 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 use constants and types by: $\kappa \cdot \rho^{(\kappa_1,\kappa_2)} = \rho^{(\kappa\cdot\kappa_1, \kappa\cdot\kappa_2)}$.

2.2 Process Expressions

The definition of process expressions is given below.

Definition 2.6 The set of *process expressions*, ranged over by P, is defined by:

$$\begin{array}{rcll} P & ::= & P_1 \mid P_2 & (\textit{parallel composition}) \\ & \mid & (\nu x : \tau)P & (\textit{channel creation}) \\ & \mid & x![y_1, \ldots, y_n] & (\textit{output atom}) \\ & \mid & x?[y_1, \ldots, y_n].P & (\textit{input prefix}) \\ & \mid & \textbf{fix}\ x[y_1, \ldots, y_n] : \tau = P_1\ \textbf{in}\ P_2\ \textbf{end} & (\textit{local process definition}) \\ & \mid & \textbf{if}\ x\ \textbf{then}\ P_1\ \textbf{else}\ P_2 & (\textit{conditional expression}) \end{array}$$

The metavariable $x(, y, z, \ldots)$ ranges over a countably infinite set of *variables*. We assume that the set of variables contains special variables *true* and *false*.

The intuitive meanings of the expressions which were not introduced in Section 1 are as follows. $\textbf{fix}\ x[y_1, \ldots, y_n] : \rho^{(\kappa_1, \kappa_2)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$ first creates a fresh channel x and spawns a process that repeatedly receives values $z_1, \ldots, z_n$ from the channel x and spawns $[z_1/y_1, \ldots, z_n/y_n]P_1$ at most κ_1 times; it then executes the process P_2.[1] We will later ensure by typing rules that x can be used only for output in P_1 and P_2 at most κ_2 times and that $\kappa_1 = \kappa_2$; therefore, $x[y_1, \ldots, y_n] = P_1$ can be regarded as a process definition since $x![z_1, \ldots, z_n]$ in P_1 or P_2 is always reduced to $[z_1/y_1, \ldots, z_n/y_n]P_1$. We often call x a *receptive channel*[2] and P_1 the *body*. $\textbf{if}\ x\ \textbf{then}\ P_1\ \textbf{else}\ P_2$ is a conditional expression that executes P_1 if x is *true* and executes P_2 if x is *false*.

A sequence of variables $y_1, \ldots, y_n$ (n may be 0) is often written as $\tilde{y}$. Similarly, a sequence of channel creations $(\nu x_1 : \tau_1) \ldots (\nu x_n : \tau_n)$ is often abbreviated to $(\nu \tilde{x} : \tilde{\tau})$. We give $(\nu x : \tau)$ and $x?[\tilde{y}].$ a higher precedence than $\mid$; for example, $x?[z].y![z] \mid (\nu w : \tau)P_1 \mid P_2$ means $(x?[z].y![z]) \mid ((\nu w : \tau)P_1) \mid P_2$. The type annotations in ν-prefix and **fix** will be used for program transformation or for efficient code generation for channels. Note that a programmer need not write them: they can be automatically recovered by the type reconstruction algorithm given in Section 4. We often omit type annotations. The bound variables of a process expression can be defined in a customary fashion, i.e., (1) a variable x is bound in P of $(\nu x)P$ and in both P_1 and P_2 of $\textbf{fix}\ x[\tilde{y}] = P_1\ \textbf{in}\ P_2\ \textbf{end}$ and, (2) variables $y_1, \ldots, y_n$ are bound in P of $x?[y_1, \ldots, y_n].P$ and $\textbf{fix}\ x[y_1, \ldots, y_n] = P\ \textbf{in}\ P_2\ \textbf{end}$. A variable which 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.

2.3 Type Judgement Form

We define type environments, several operations on them, use constraint sets, and type judgement form.

Type Environments. A *type environment* Γ is a mapping from a finite set of variables to the set of types such that $\Gamma(true) = bool$ and $\Gamma(false) = bool$. $dom(\Gamma)$ denotes the domain of Γ. We write $x_1 : \tau_1, \ldots, x_n : \tau_n$, abbreviated to $\tilde{x} : \tilde{\tau}$, for the type environment Γ such that

[1] Thus, $\textbf{fix}\ x[y_1, \ldots, y_n] : \rho^{(\kappa_1, \kappa_2)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$ is similar to

$(\nu x : \rho^{(\kappa_1, \kappa_2)})\ (\overbrace{x?[y_1, \ldots, y_n].P_1 \mid \cdots \mid x?[y_1, \ldots, y_n].P_1}^{\kappa_1} \mid P_2)$

[2] Named after Sangiorgi's *receptive name* [16, 15]

$dom(\Gamma) = \{x_1, \ldots, x_n, true, false\}$ and $\Gamma(x_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Γ' such that $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ if $x \neq y$.

The operations '+' and '$\sqcup$' on types are pointwise extended to type environments: that is, $\Gamma_1 + \Gamma_2$ ($\Gamma_1 \sqcup \Gamma_2$, resp.) is a type environment Γ such that $dom(\Gamma) = dom(\Gamma_1) \cup dom(\Gamma_2)$, $\Gamma(x) = \Gamma_1(x) + \Gamma_2(x)$ ($\Gamma_1(x) \sqcup \Gamma_2(x)$, resp.) if $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$, $\Gamma(x) = \Gamma_1(x)$ if $x \in dom(\Gamma_1) \setminus dom(\Gamma_2)$, and $\Gamma(x) = \Gamma_2(x)$ if $x \in dom(\Gamma_2) \setminus dom(\Gamma_1)$. $\kappa \cdot \Gamma$ is defined by: $dom(\kappa \cdot \Gamma) = dom(\Gamma)$ and $(\kappa \cdot \Gamma)(x) = \kappa \cdot (\Gamma(x))$. $\Gamma_1 \geq \Gamma_2$ is defined by: $\Gamma_1 \geq \Gamma_2$ if and only if $dom(\Gamma_1) \supseteq dom(\Gamma_2)$ and $\forall x \in dom(\Gamma_2).\Gamma_1(x) \geq \Gamma_2(x)$.

Use Constraint Set. A *use constraint set* Θ is a set of inequalities of the form $\kappa_1 \geq \kappa_2$. The relation $\geq$ is extend to the relation $\geq_\Theta$ between uses:

Definition 2.7 A substitution S_κ of use constants for use variables *respects* a use constraint set Θ if and only if for any inequality $\kappa_1 \geq \kappa_2$ in Θ, $S_\kappa \kappa_1 \geq S_\kappa \kappa_2$ holds. The binary relation $\geq_\Theta$ is defined by: $\kappa_1 \geq_\Theta \kappa_2$ if and only if, for any substitution that respects Θ, $S_\kappa \kappa_1 \geq S_\kappa \kappa_2$. $\kappa_1 \geq_\Theta \kappa_2$ is extended to a binary relation on types by: $\rho_1^{(\kappa_1, \kappa_2)} \geq_\Theta \rho_2^{(\kappa_3, \kappa_4)}$ if and only if $\rho_1 = \rho_2$, $\kappa_1 \geq_\Theta \kappa_3$ and $\kappa_2 \geq_\Theta \kappa_4$.

For example, $j \geq_\Theta 1$ holds for $\Theta = \{j \geq k + 1\}$.

We do not present syntactic inference rules to derive the (semantic) relation $\kappa_1 \geq_\Theta \kappa_2$. Note that, however, obvious sound inference rules such as:

$$\frac{\kappa_1 \geq \kappa_2 \in \Theta}{\kappa_1 \geq_\Theta \kappa_2} \qquad \frac{\kappa_1 \geq_\Theta \kappa_2}{S_\kappa \kappa_1 \geq_{S_\kappa \Theta} S_\kappa \kappa_2} \qquad \frac{\kappa_1 \geq_\Theta \kappa_2 \quad \kappa_2 \geq_\Theta \kappa_3}{\kappa_1 \geq_\Theta \kappa_3}$$

where S_κ is a substitution for use variables, are enough for our reconstruction algorithm.

Type Judgement Form. A type judgement is of the form $\Gamma; \Theta \vdash P$, read as "P is well-typed under the type environment Γ and the use constraint set Θ." 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 judgement $\Gamma, x : [\tau]^{(1,0)}; \Theta \vdash P$ means that P uses *x at most once for receiving a value of the type τ, and it never uses x for sending a value.*

3 Typing Rules

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

$$\frac{\Gamma_1; \Theta \vdash P_1 \quad \Gamma_2; \Theta \vdash P_2}{\Gamma_1 + \Gamma_2; \Theta \vdash P_1 \mid P_2} \text{ (T-PAR)}$$

On the other hand, in a conditional expression **if** x **then** P_1 **else** P_2, only either P_1 or P_2 is executed. So, both two expressions should be typed under the same type environment. Thus, the rule for conditional expressions is:

$$\frac{\Gamma; \Theta \vdash P_1 \quad \Gamma; \Theta \vdash P_2}{x : bool + \Gamma; \Theta \vdash \textbf{if } x \textbf{ then } P_1 \textbf{ else } P_2} \text{ (T-IF)}$$

Because x is used for output in $x![\tilde{y}]$, the output use of x must be greater than 0. Similarly, since x is used for input in $x?[\tilde{y}].P$, the input use of x must be greater than 0. Thus, the rules for an output atom and an input prefix are:

$$\frac{\kappa_2 \geq_\Theta 1}{x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} + y_1 : \tau_1 + \cdots + y_n : \tau_n; \Theta \vdash x![\tilde{y}]} \text{ (T-Out)} \quad \frac{\Gamma, \tilde{y} : \tilde{\tau}; \Theta \vdash P \quad \kappa_1 \geq_\Theta 1}{\Gamma + x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}; \Theta \vdash x?[\tilde{y}].P} \text{ (T-In)}$$

Unlike the case for an input prefix, P_1 of **fix** $x[\tilde{y}] = P_1$ **in** P_2 **end** may be executed more than once. The typing rule for a local process definition is:

$$\frac{\Gamma_1, x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})}, \tilde{y} : \tilde{\tau}; \Theta \vdash P_1 \quad \Gamma_2, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})}; \Theta \vdash P_2 \quad 0 \geq_\Theta \kappa_{11} \quad 0 \geq_\Theta \kappa_{21}}{(\kappa_{22} \cdot (\kappa_{12} + 1) \cdot \Gamma_1) + \Gamma_2; \Theta \vdash \textbf{fix } x[\tilde{y}] : [\tilde{\tau}]^{(\kappa_{22}\cdot(\kappa_{12}+1),\kappa_{22}\cdot(\kappa_{12}+1))} = P_1 \textbf{ in } P_2 \textbf{ end}} \text{ (T-Proc)}$$

The conditions $0 \geq_\Theta \kappa_{11}$ and $0 \geq_\Theta \kappa_{21}$ ensure that x is not used for input in P_1 or P_2. The multiplication of Γ_1 by $\kappa_{22}\cdot(\kappa_{12}+1)$ and the type annotation for x are explained as follows. The premise $\Gamma_1, x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})}, \tilde{y} : \tilde{\tau}; \Theta \vdash P_1$ indicates that x may be used κ_{12} 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_{12} times for output; moreover, each use of x in P_1 again spawns P_1 and causes x to be used κ_{12} times; thus, each use of x for output in P_2 may cause x to be used $1 + \kappa_{12} + \kappa_{12}^2 + \cdots$ times. Since the premise $\Gamma_2, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})}; \Theta \vdash P_2$ indicates that x may be used κ_{22} times for output in P_2, the total number of uses of x for output is bound by $\kappa_{22}\cdot(1+\kappa_{12}+\kappa_{12}^2+\cdots) = \kappa_{22}\cdot(1+\kappa_{12})$. For example, in the expression **fix** $x[\,] = (x![\,] \,|\, x![\,])$ **in** $x![\,]$ **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.

$$\frac{\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}; \Theta \vdash P}{\Gamma; \Theta \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P} \text{ (T-New)}$$

We can weaken the assumption on uses. For example, if $x : [\tau]^{(0,1)}, y : \tau; \Theta \vdash x![y]$ holds, then we allow $x : [\tau]^{(0,\omega)}, y : \tau; \Theta \vdash x![y]$: that is, if P is well-typed under the assumption that x is used at most once, then P is also well-typed under the weaker assumption that x is used an arbitrary number of times. The rule (T-Weak) allows such weakening on uses.

$$\frac{\Gamma'; \Theta \vdash P \quad \Gamma \geq_\Theta \Gamma'}{\Gamma; \Theta \vdash P} \text{ (T-Weak)}$$

4 Type Reconstruction and Detection of Linear Channels

This section shows an algorithm to detect linear channels. Our algorithm consists of two phases: in the first phase, the most general typing (principal typing) is computed for a given process, and in the second phase, the least solution for the use constraint set is obtained and substituted. After defining the notion of principal typing in Subsection 4.1, we present a type reconstruction algorithm to compute the principal typing in Subsection 4.2, and then show how to solve a use constraint set in Subsection 4.3. As discussed in Subsection 4.4, our algorithm runs in time polynomial in the size of a process. A simple example of the detection of linear channels is also presented.

4.1 Principal Typing

The principal typing of a process expression P is the most general triple (Γ, Θ, P') such that $P' = SP$ and $\Gamma; \Theta \vdash P'$ for some substitution S for type variables and use variables. We need a few preliminary definitions.

Definition 4.1 *A* type-use substitution *is a pair consisting of a substitution S_ρ of bare types for type variables and a substitution S_κ of uses for use variables. $(S_\rho, S_\kappa)(\tau)$ is defined as $S_\kappa S_\rho \tau$. $(S_\rho, S_\kappa)(\kappa)$ is defined as $S_\kappa \kappa$.*

We write S for a type-use substitution and often call it a *substitution* unless confusing.

The relation $\Theta_1 \models \Theta_2$ below means intuitively that Θ_1 is a stronger constraint set than Θ_2. Note that if Θ_1 is not satisfiable, $\Theta_1 \models \Theta_2$ holds for any use constraint set Θ_2.

Definition 4.2 $\Theta_1 \models \Theta_2$ holds if and only if $\forall(\kappa_1 \geq \kappa_2) \in \Theta_2, \kappa_1 \geq_{\Theta_1} \kappa_2$.

Definition 4.3 (Γ, Θ, P') is a *principal typing* of P if and only if the following three conditions are satisfied: (1) $P' = SP$ for some substitution S, (2) $\Gamma; \Theta \vdash P'$, and (3) if $P'' = SP$ for some substitution S and $\Gamma'; \Theta' \vdash P''$, then there exists a substitution S' such that $P'' = S'P'$, $\Theta' \models S'\Theta$ and $\Gamma' \supseteq S'\Gamma$.

Note that a principal typing of P may exist even when there is no Γ and S such that $\Gamma; \emptyset \vdash SP$. For example, consider the process **fix** $x[\,] : [\,]^{(j,k)} = x![\,]$ **in** $x?[\,].x![\,]$ **end** : although it violates the rule that a channel x bound by **fix** cannot be used for input, it has $(\emptyset, \{0 \geq j, j \geq 1\}, \textbf{fix}\ x[\,] : [\,]^{(j,j)} = x![\,]\ \textbf{in}\ x?[\,].x![\,]\ \textbf{end})$ as the principal typing; this kind of process will be rejected in the second phase, in which it is checked whether the use constraint set is satisfiable.

4.2 Type Reconstruction Algorithm

As usual, our type reconstruction algorithm is obtained by reading typing rules in a bottom-up manner. However, the typing rules presented in the previous section are not syntax-directed. So, we combine the rule (T-Weak) with the other rules and obtain syntax-directed rules that are equivalent to the original typing rules.

For example, the rule (T-Par) is replaced by:

$$\frac{\Gamma_1; \Theta_1 \vdash P_1 \quad \Gamma_2; \Theta_2 \vdash P_2 \quad \Gamma \geq_\Theta \Gamma_1 + \Gamma_2 \quad \Theta \models \Theta_1 \quad \Theta \models \Theta_2}{\Gamma; \Theta \vdash P_1 \mid P_2} \text{ (ST-Par)}$$

It tells us to first compute the principal typings $(\Gamma_1, \Theta_1, P_1')$ and $(\Gamma_2, \Theta_2, P_2')$ for P_1 and P_2, and then compute the most general type environment Γ and use constraint set Θ such that $\Gamma \geq_\Theta \Gamma_1 + \Gamma_2$, $\Theta \models \Theta_i$ for $i = 1, 2$.

The whole syntax-directed rules are shown in Figure 1. To avoid confusion, we write $\Gamma; \Theta \vdash_{\mathcal{TU}} P$ if $\Gamma; \Theta \vdash P$ is derived from the typing rules of Section 3, and $\Gamma; \Theta \vdash_{\mathcal{STU}} P$ if $\Gamma; \Theta \vdash P$ is derived from the typing rules of Figure 1.

As will be stated in Theorem 4.6, the two typing rules are essentially equivalent. We present several lemmas for the theorem:

Lemma 4.4 Suppose $\Theta' \models \Theta$. Then: (1) $\kappa_1 \geq_\Theta \kappa_2$ implies $\kappa_1 \geq_{\Theta'} \kappa_2$, (2) $\tau_1 \geq_\Theta \tau_2$ implies $\tau_1 \geq_{\Theta'} \tau_2$, and (3) $\Gamma; \Theta \vdash_{\mathcal{TU}} P$ implies $\Gamma; \Theta' \vdash_{\mathcal{TU}} P$.

Proof The first two statements are trivial from the definition of $\geq_\Theta$. (3) is proved by straightforward induction on the structure of the derivation of $\Gamma; \Theta \vdash_{\mathcal{TU}} P$. □

Lemma 4.5 If $\Gamma; \Theta \vdash_{\mathcal{STU}} P$ and $\Gamma' \geq_\Theta \Gamma$, then $\Gamma'; \Theta \vdash_{\mathcal{STU}} P$

$$\frac{\Gamma_1;\Theta_1 \vdash P_1 \quad \Gamma_2;\Theta_2 \vdash P_2 \quad \Gamma \geq_\Theta \Gamma_1 + \Gamma_2 \quad \Theta \models \Theta_1 \quad \Theta \models \Theta_2}{\Gamma;\Theta \vdash P_1 \mid P_2} \text{ (ST-Par)}$$

$$\frac{\kappa_2 \geq_\Theta 1 \quad \tau_1' \geq_\Theta \tau_1 \quad \cdots \quad \tau_n' \geq_\Theta \tau_n \quad \Gamma \geq_\Theta x : [\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)} + y_1 : \tau_1' + \cdots + y_n : \tau_n'}{\Gamma;\Theta \vdash x![y_1, \ldots, y_n]} \text{ (ST-Out)}$$

$$\frac{\Gamma, y_1 : \tau_1, \ldots, y_n : \tau_n;\Theta \vdash P \quad \kappa_1 \geq_{\Theta'} 1 \quad \Gamma' \geq_{\Theta'} \Gamma + x : [\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)} \quad \Theta' \models \Theta}{\Gamma';\Theta' \vdash x?[y_1, \ldots, y_n].P} \text{ (ST-In)}$$

$$\frac{\begin{array}{c}\Gamma_1, x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})}, \tilde{y} : \tilde{\tau};\Theta_1 \vdash P_1 \quad \Gamma_2, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})};\Theta_2 \vdash P_2 \\ 0 \geq_\Theta \kappa_{11} \quad 0 \geq_\Theta \kappa_{21} \quad \kappa \geq_\Theta \kappa_{22} \cdot (\kappa_{12} + 1) \\ \Gamma \geq_\Theta (\kappa \cdot \Gamma_1) + \Gamma_2 \quad \Theta \models \Theta_1 \quad \Theta \models \Theta_2\end{array}}{\Gamma;\Theta \vdash \textbf{fix } x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1 \textbf{ in } P_2 \textbf{ end}} \text{ (ST-Proc)}$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)};\Theta \vdash P}{\Gamma;\Theta \vdash (\nu x^{(\kappa_1,\kappa_2)})P} \text{ (ST-New)}$$

$$\frac{\Gamma_1;\Theta_1 \vdash P_1 \quad \Gamma_2;\Theta_2 \vdash P_2 \quad \Gamma \geq_\Theta x : \mathit{bool} + (\Gamma_1 \sqcup \Gamma_2) \quad \Theta \models \Theta_1 \quad \Theta \models \Theta_2}{\Gamma;\Theta \vdash \textbf{if } x \textbf{ then } P_1 \textbf{ else } P_2} \text{ (ST-If)}$$

Figure 1: Syntax-directed Typing Rules

Proof Straightforward induction on the structure of the proof of $\Gamma;\Theta \vdash_{\mathcal{STU}} P$. □

Theorem 4.6 $\Gamma;\Theta \vdash_{\mathcal{TU}} P$ if and only if $\Gamma;\Theta \vdash_{\mathcal{STU}} P$.

Proof The 'only if' part is proved by structural induction on the proof of $\Gamma;\Theta \vdash_{\mathcal{TU}} P$ with case analysis on the last rule used. The case for (T-Weak) follows from Lemma 4.5. The other cases are trivial by the reflectivity of $\models$ and $\geq_\Theta$. The 'if' part follows from the fact that each syntax-directed rule corresponds to a combination of rules of Section 3. □

In our reconstruction algorithm, we use several subprocedures including unification of types. If every use of the types to be unified is either a constant or a variable, unification can be done in the ordinary first-order manner. We show the definitions of a unifier and the most general unifier, and the theorem for the existence of the most general unifier.

Definition 4.7 (unifier) A substitution S is a *unifier* of a set of pairs of types or bare types $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2}), (\rho_{11}, \rho_{12}), \ldots, (\rho_{m1}, \rho_{m2})\}$ if and only if $S\tau_{i1} = S\tau_{i2}$ for $i \in \{1, \ldots, n\}$ and $S\rho_{i1} = S\rho_{i2}$ for $i \in \{1, \ldots, m\}$.

Definition 4.8 (most general unifier) A unifier S of $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2}), (\rho_{11}, \rho_{12}), \ldots, (\rho_{m1}, \rho_{m2})\}$ is the *most general unifier* if and only if for any other unifier S' of the same set, there exists S'' such that $S' = S''S$.

Theorem 4.9 (Unification) Given $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2}), (\rho_{11}, \rho_{12}), \ldots, (\rho_{m1}, \rho_{m2})\}$ of which every use is either a constant or a variable, there exists an algorithm U which computes the most general unifier of the set $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2}), (\rho_{11}, \rho_{12}), \ldots, (\rho_{m1}, \rho_{m2})\}$, or reports failure if it does not exist.

Other subprocedures $\bigoplus$, $\bigodot$, and $\bigsqcup$ are defined below. $(\Gamma_1, \Theta_1)\bigoplus(\Gamma_2, \Theta_2)$ computes the most general triple (S, Γ, Θ) such that $\Gamma \geq_\Theta S\Gamma_1 + S\Gamma_2$ and $\Theta \models S\Theta_i$ for $i = 1, 2$. $\bigsqcup$ and $\bigodot$ are the corresponding procedures used for solving $\Gamma \geq_\Theta \Gamma_1 \sqcup \Gamma_2$ and $\Gamma \geq_\Theta \kappa \cdot \Gamma'$. In the definitions, $bare(\rho^{(\kappa_1,\kappa_2)})$, $\mathcal{I}(\rho^{(\kappa_1,\kappa_2)})$, and $\mathcal{O}(\rho^{(\kappa_1,\kappa_2)})$ denote ρ, κ_1, and κ_2, respectively.

Definition 4.10 $(\Gamma_1, \Theta_1)\bigoplus(\Gamma_2, \Theta_2)$ is defined as the following procedure:

$(\Gamma_1, \Theta_1)\bigoplus(\Gamma_2, \Theta_2) =$
  let
    $S = U(\{(bare(\Gamma_1(x)), bare(\Gamma_2(x)))|x \in dom(\Gamma_1) \cap dom(\Gamma_2)\})$
    Γ be the type environment such that
      $dom(\Gamma) = dom(\Gamma_1) \cup dom(\Gamma_2)$,

$$\Gamma(x) = \begin{cases} S(bare(\Gamma_1(x)))^{(j_x,k_x)} & (x \in dom(\Gamma_1) \cap dom(\Gamma_2)) \\ & \text{where } j_x \text{ and } k_x \text{ are fresh} \\ S(\Gamma_1(x)) & (x \in dom(\Gamma_1) \setminus dom(\Gamma_2)) \\ S(\Gamma_2(x)) & (x \in dom(\Gamma_2) \setminus dom(\Gamma_1)) \end{cases}$$

    $\Theta = \Theta_1 \cup \Theta_2 \cup \{\mathcal{I}(\Gamma(x)) \geq \mathcal{I}(\Gamma_1(x)) + \mathcal{I}(\Gamma_2(x))|x \in dom(\Gamma_1) \cap dom(\Gamma_2)\}$
      $\cup\{\mathcal{O}(\Gamma(x)) \geq \mathcal{O}(\Gamma_1(x)) + \mathcal{O}(\Gamma_2(x))|x \in dom(\Gamma_1) \cap dom(\Gamma_2)\}$
  in $(S, \Gamma, S\Theta)$

$\bigoplus$ is left-associative and $(S, \Gamma_1, \Theta_1)\bigoplus(\Gamma_2, \Theta_2)$ is defined as $(S'S, \Gamma, \Theta)$ where $(S', \Gamma, \Theta) = (\Gamma_1, \Theta_2)\bigoplus(\Gamma_2, \Theta_2)$

Definition 4.11 $\kappa\bigodot(\Gamma, \Theta)$ is defined as the following procedure:

$\kappa\bigodot(\Gamma, \Theta) =$
  let Γ' be the type environment such that
    $dom(\Gamma') = dom(\Gamma)$ and $\Gamma'(x) = (bare(\Gamma(x)))^{(j_x,k_x)}$ where j_x and k_x are fresh
  $\Theta' = \Theta \cup \{\mathcal{I}(\Gamma'(x)) \geq \kappa \cdot \mathcal{I}(\Gamma(x))|x \in dom(\Gamma)\}$
    $\cup\{\mathcal{O}(\Gamma'(x)) \geq \kappa \cdot \mathcal{O}(\Gamma(x))|x \in dom(\Gamma)\}$
  in (Γ', Θ')

The definition of $(\Gamma_1, \Theta_1)\bigsqcup(\Gamma_2, \Theta_2)$ is obtained from the definition of $\bigoplus$ by replacing every $+$ with $\sqcup$.

The algorithm PTU, shown in Figure 2, takes a process expression P as an input, and returns the principal typing of P. As 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 fresh type and use variables $\alpha^{(j,k)}$ into a place where type annotations are omitted. It can be proved by induction on the structure of P that $PTU(P)$ computes the principal typing of P:

Theorem 4.12 (Correctness of PTU) Let P be a process expression such that every use in P is either a use constant or a use variable. If $\Gamma; \Theta \vdash SP$ for some S, Γ, and Θ, then $PTU(P)$ outputs the principal typing of P; otherwise, $PTU(P)$ fails (by the failure of the unification algorithm U.)

Proof See Appendix A. □

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 $dom(\Gamma) \setminus \{x_1, \ldots, x_n\}$.

$$
\begin{array}{l}
PTU(P_1 \mid P_2) = \text{let } (\Gamma_1, \Theta_1, P_1') = PTU(P_1) \\
\qquad\qquad (\Gamma_2, \Theta_2, P_2') = PTU(P_2) \\
\qquad\qquad (S, \Gamma, \Theta) = (\Gamma_1, \Theta_1) \bigoplus (\Gamma_2, \Theta_2) \\
\qquad \text{in } (\Gamma, \Theta, S(P_1' \mid P_2'))
\end{array}
$$

$$
\begin{array}{l}
PTU((\nu x : \tau)P) = \text{let } (\Gamma, \Theta, P') = PTU(P) \\
\qquad \text{in if } x \in dom(\Gamma) \text{ then } (\Gamma \setminus \{x\}, \Theta, S((\nu x : \Gamma(x))P')) \\
\qquad\qquad\qquad \text{where } S = U(\{(\tau, \Gamma(x))\}) \\
\qquad\qquad \text{else } (\Gamma, \Theta, (\nu x : \tau)P')
\end{array}
$$

$$
\begin{array}{l}
PTU(x![y_1, \ldots, y_n]) = \\
\qquad \text{let } (S, \Gamma, \Theta) = (x : [\alpha_1^{(j_{11}, j_{12})}, \ldots, \alpha_n^{(j_{n1}, j_{n2})}]^{(j,k)}, \emptyset) \\
\qquad\qquad\qquad \bigoplus (y_1 : \alpha_1^{(k_{11}, k_{12})}, \emptyset) \bigoplus \cdots \bigoplus (y_n : \alpha_n^{(k_{n1}, k_{n2})}, \emptyset) \\
\qquad \text{in } (\Gamma, \Theta \cup S(\{k \geq 1\} \cup \{k_{ij} \geq j_{ij} | 1 \leq i \leq n, j = 1, 2\}), x![y_1, \ldots, y_n])
\end{array}
$$

$$
\begin{array}{l}
PTU(x?[y_1, \ldots, y_n].P) = \\
\qquad \text{let } (\Gamma_1, \Theta_1, P') = PTU(P) \\
\qquad\quad (S_1, \Gamma, \Theta) = (x : [\tau_1, \ldots, \tau_n]^{(j,k)}, \{j \geq 1\}) \bigoplus (\Gamma_1 \setminus \{y_1, \ldots, y_n\}, \Theta_1) \\
\qquad\qquad \text{where } \tau_i \text{ is } \Gamma_1(y_i) \text{ if } y_i \in dom(\Gamma_1), \text{ or } \alpha_{y_i}^{(j_{y_i}, k_{y_i})} \text{ otherwise (for } i \in \{1, \ldots, n\}) \\
\qquad \text{in } (\Gamma, \Theta, S_1(x?[y_1, \ldots, y_n].P'))
\end{array}
$$

$$
\begin{array}{l}
PTU(\textbf{if } x \textbf{ then } P_1 \textbf{ else } P_2) = \\
\qquad \text{let } (\Gamma_1, \Theta_1, P_1') = PTU(P_1) \\
\qquad\quad (\Gamma_2, \Theta_2, P_2') = PTU(P_2) \\
\qquad\quad (S_1, \Gamma_3, \Theta_3) = (\Gamma_1, \Theta_2) \bigsqcup (\Gamma_2, \Theta_2) \\
\qquad\quad (S_2, \Gamma_4, \Theta_4) = (x : bool^{(\omega, \omega)}, \emptyset) \bigoplus (\Gamma_3, \Theta_3) \\
\qquad \text{in } (\Gamma_4, \Theta_4, S_2 S_1(\textbf{if } x \textbf{ then } P_1' \textbf{ else } P_2'))
\end{array}
$$

$$
\begin{array}{l}
PTU(\textbf{fix } x[y_1, \ldots, y_n] : \tau = P_1 \textbf{ in } P_2 \textbf{ end}) = \\
\qquad \text{let } (\Gamma_1, \Theta_1, P_1') = PTU(P_1) \\
\qquad\quad (\Gamma_2, \Theta_2, P_2') = PTU(P_2) \\
\qquad\quad \rho_i^{(\kappa_{i1}, \kappa_{i2})} \text{ be } \Gamma_i(x) \text{ if } x \in dom(\Gamma_i), \text{ or } \alpha_i^{(j_i, k_i)} \text{ otherwise (for } i = 1, 2) \\
\qquad\quad (\Gamma', \Theta') = (\kappa_{22} \cdot (\kappa_{12} + 1)) \bigodot (\Gamma_1 \setminus \{x, y_1, \ldots, y_n\}, \Theta_1) \\
\qquad\quad (S_1, \Gamma, \Theta) = (\Gamma', \Theta') \bigoplus (\Gamma_2 \setminus \{x\}, \Theta_2) \\
\qquad\quad E = \{(S_1 \rho_1^{(j,j)}, S_1 \tau), (S_1 \rho_1^{(j,j)}, S_1 \rho_2^{(j,j)}), (S_1 \rho_2^{(j,j)}, S_1 [\tau_1, \ldots, \tau_n]^{(j,j)})\} \\
\qquad\qquad \text{where } \tau_i \text{ is } \Gamma_1(y_i) \text{ if } y_i \in dom(\Gamma_1), \text{ or } \alpha_{y_i}^{(j_{y_i}, k_{y_i})} \text{ otherwise (for } i \in \{1, \ldots, n\}) \\
\qquad\quad S_2 = U(E) \\
\qquad \text{in } (S_2 \Gamma, S_2(\Theta \cup \{0 \geq \kappa_{11}, 0 \geq \kappa_{21}, j \geq \kappa_{22} \cdot (\kappa_{12} + 1)\}), \\
\qquad\quad S_2 S_1(\textbf{fix } x[y_1, \ldots, y_n] : \tau = P_1' \textbf{ in } P_2' \textbf{ end}))
\end{array}
$$

Figure 2: Type Reconstruction Algorithm PTU

4.3 Solving Use Constraint Set

Given a pair (Γ, Θ) such that $\Gamma; \Theta \vdash P$, we obtain Γ' such that $\Gamma'; \emptyset \vdash P'$ by solving Θ. (P' is the process expression obtained by substituting the solution for the use variables of P.) A solution of Θ is obtained by: (1) dividing Θ into two parts $\{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$ and $\{c_{n+1} \geq \kappa_{n+1}, \ldots, c_m \geq \kappa_m\}$ where each c_i is a use constant (note that each inequality output by PTU is of the form $j \geq \kappa$ or $c \geq \kappa$), (2) solving the first part of inequalities and (3) checking whether the solution satisfies the second part. (If the check fails, Θ has no solutions). Because every operation on uses is monotonic and because $j_1, \ldots, j_n$ can range over finite space, the least solution of the first part is calculated by the following simple iterative method[3]:

Theorem 4.13 (Least Solution) Let $\Theta = \{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$. 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 $M \geq 0$, $j_1 = \kappa_1^{(M)}, \ldots, j_n = \kappa_n^{(M)}$ is the least solution of Θ.

Proof We regard κ_i as an n-ary function on $j_1, \ldots, j_n$, and write $\kappa_i(\kappa'_1, \ldots, \kappa'_n)$ instead of $[\kappa'_1/j_1, \ldots, \kappa'_n/j_n]\kappa_i$.

Since use can range over the finite space and the operations for uses are monotonic, there exists M such that $\kappa_i^{(M)} = \kappa_i(\kappa_1^{(M)}, \ldots, \kappa_n^{(M)})$ holds for all $i \in \{1, \ldots, n\}$. Suppose that $\kappa'_i \geq \kappa_i(\kappa'_1, \ldots, \kappa'_n)$ holds for some $\kappa'_1, \ldots, \kappa'_n$. Then, by the monotonicity of each κ_i,

$$\kappa'_i \geq \kappa_i(\kappa'_1, \ldots, \kappa'_n) \geq \ldots \geq (\kappa_i)^M(\kappa'_1, \ldots, \kappa'_n)$$

Moreover, by $\kappa'_i \geq 0$,

$$(\kappa_i)^M(\kappa'_i) \geq (\kappa_i)^M(0) (= \kappa_i^{(M)})$$

from which we obtain $\kappa'_i \geq \kappa_i^{(M)}$ for $i \in \{1, \ldots, n\}$. □

Remark: If use variables occur in the type environment output by the algorithmPTU, then those variables should be kept undetermined, because they may be constrained outside the process expression. When such a process expression is required to be compiled at that time (for separate compilation, etc.), a programmer can declare their uses, or the compiler can assign ω to the undetermined use variables.

4.4 Complexity of Analysis

In this section, we informally show that the time complexity for solving the use constraint set of the principal typing is polynomial in the size of the process expression. In the discussion below, we use the notations in Theorem 4.13 and define the size $size(\kappa)$ of a use as 1 if κ is either a constant or a variable, or $1 + size(\kappa_1) + size(\kappa_2)$ if κ is $\kappa_1 + \kappa_2$, $\kappa_1 \sqcup \kappa_2$, or $\kappa_1 \cdot \kappa_2$.

Since a $\kappa_i^{(j)}$ increases monotonically as j increases, the number of steps M of iteration is at most $2 \cdot n$. In each step, the total cost for computing $\kappa_1^{(j)}, \ldots, \kappa_n^{(j)}$ is $O(\sum_i size(\kappa_i))(\leq O(n \cdot \max_i(size(\kappa_i))))$.

The number n of inequalities in a use constraint set is $O(p^2)$ where p is the size of a process expression because the subprocedures like $\bigoplus$ adds at most twice as many inequalities as the

[3]Of course, we could apply some symbolic simplification methods instead of the naive iteration.

number of free variables in a process expression. Apparently the size of each inequality generated by the subprocedures is $O(1)$.

As a result, the computational complexity for solving the constraint set 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.

4.5 An Example of Analysis and Optimization

We show an example of our analysis and optimization. Consider the following process, which computes the nth Fibonacci number sequentially (the language has been extended with integer values and several mathematical operations):

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

Let us infer the values of use variables (j, k) attached to the channel c_1. The reconstruction algorithm outputs the triple $(\Gamma, \Theta, \textbf{fix}\ fib \ldots)$ where $\Theta = \{j \geq j_1 + j_2, k \geq k_1 + k_2, k_1 \geq k_3 \geq 1, j_2 \geq 1, \ldots\}$. j_1 and k_1 denote how often c_1 are used for input and output in the process $fib![n-1, c_1]$. 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 inequalities $j \geq j_1 + j_2, k \geq k_1 + k_2$. The inequality $j_2 \geq 1$ is also obtained from the input expression $c_1?[x].(\cdots)$. Furthermore, during the reconstruction, a type of the form $[int, [int]^{(j_3,k_3)}]^{(j_4,k_4)}$ is assigned to the receptive channel fib; then the inequality $k_1 \geq k_3$ is obtained from the expression $fib![n-1, c_1]$, and $k_3 \geq 1$ is obtained from the expression $c![1]$. By solving Θ, 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 by the following optimized one:

$$
\begin{array}{l}
\textbf{fix}\ fibopt[n,c] = \\
\quad \textbf{if}\ n < 2\ \textbf{then}\ c![1] \\
\quad \textbf{else fix}\ c_1[x] : [int]^{(1,1)} = (\textbf{fix}\ c_2[y] : [int]^{(1,1)} = c![x+y] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ fibopt![n-2, c_2])\ \textbf{end} \\
\qquad \textbf{in}\ fibopt![n-1, c_1]\ \textbf{end} \\
\textbf{in}\ P\ \textbf{end}
\end{array}
$$

Note that the optimized process corresponds to the continuation passing style representation [1] of the functional Fibonacci program (the channels c_1 and c_2 can be viewed as continuations).

5 Correctness

In this section, we show the soundness of the type system with respect to operational semantics: it must be guaranteed, for example, that if $x : [\tilde{\tau}]^{(0,1)}; \emptyset \vdash P$, then x can only be used at most once for output during evaluation of P. In order to check the correctness, the reduction relation is annotated with some use information. Then, we show the correctness of the type system with respect to this reduction semantics. Since we are mainly interested in the correctness of linear channels detected by constraint solving, we assume that no use variables appear in Γ and P. So, Θ is always assumed to be empty and omitted in type judgements.

5.1 Reduction Semantics with Uses

Following the standard presentation for process calculi[9], the reduction semantics for process expressions is defined via two relations: a *structural congruence* $P_1 \cong P_2$ and a *reduction relation*.

Definition 5.1 *Structural congruence* $P_1 \cong P_2$ is the least congruence on process expressions closed under the following rules.

$$\begin{array}{rcl}
P_1 \mid P_2 & \cong & P_2 \mid P_1 \\
(P_1 \mid P_2) \mid P_3 & \cong & P_1 \mid (P_2 \mid P_3) \\
(\nu x)(P_1 \mid P_2) & \cong & P_1 \mid (\nu x)P_2 \\
 & & (\text{if } x \text{ is not free in } P_1) \\
\textbf{fix } x[\tilde{y}] = P_1 \textbf{ in } P_2 \mid P_3 \textbf{ end} & \cong & P_2 \mid \textbf{fix } x[\tilde{y}] = P_1 \textbf{ in } P_3 \textbf{ end} \\
 & & (\text{if } x \text{ is not free in } P_2) \\
(\nu x)\textbf{fix } y[\tilde{z}] = P_1 \textbf{ in } P_2 \textbf{ end} & \cong & \textbf{fix } y[\tilde{z}] = P_1 \textbf{ in } (\nu x)P_2 \textbf{ end} \\
 & & (\text{if } x \neq y \text{ and } x \text{ is not free in } P_1)
\end{array}$$

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 two environments Γ, Γ' and a label l, and written as $\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma'$. Intuitively, the label l memorizes on which channel communication is performed: l is a variable x if P is reduced to P' by communication on the free channel x and l is the special label ε if the reduction comes from communication on a bound channel. The type environments Γ and Γ' represents how often each channel can be used before and after the reduction, respectively: Γ' is obtained from Γ by decreasing uses according to l.

To define the relation $\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma'$, we define an auxiliary relation $\mathcal{D}; \Gamma \vdash P \xrightarrow{l} P'$ where $\mathcal{D}$, called a *receptive channel environment*, keeps information on the free receptive channels, and define Γ^{-l}, which denotes the type environment after a reduction with the label l.

Definition 5.2 A *receptive channel environment*, denoted by $\mathcal{D}$, is a mapping from the set of variables to the set of quadruples consisting of a sequence of variables, a type, a process expression, and a type environment.

We write $x_1[\tilde{y}_1] : \tau_1 = P_1 : \Gamma_1, \ldots, x_n[\tilde{y}_n] : \tau_n = P_n : \Gamma_n$ for a receptive channel environment $\mathcal{D}$ such that $dom(\mathcal{D}) = \{x_1, \ldots, x_n\}, \Gamma(x_i) = (\tilde{y}_i, \tau_i, P_i, \Gamma_i)$, and for all i, j such that $j < i$, x_i is not a free variable of P_j. When $x \notin dom(\mathcal{D})$, $\mathcal{D}, x[\tilde{y}] : \tau = P : \Gamma$ represents the receptive channel environment $\mathcal{D}'$ such that $\mathcal{D}'(x) = (\tilde{y}, \tau, P, \Gamma)$ and $\mathcal{D}'(y) = \mathcal{D}(y)$ if $x \neq y$.

Definition 5.3 The relation $\Gamma; \mathcal{D} \vdash P \xrightarrow{l} P'$ is the least relation closed under the following rules:

$$\mathcal{D}; \Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash x?[y_1, \ldots, y_n].P \mid x![z_1, \ldots, z_n] \xrightarrow{x} [z_1/y_1, \ldots, z_n/y_n]P \ (\text{R-Comm})$$

$$\frac{\mathcal{D}(x) = (\tilde{y}, \tau_x, P, \Gamma_x)}{\mathcal{D}; \Gamma, x : \rho^{(\kappa_1,\kappa_2)} \vdash x![z_1, \ldots, z_n] \xrightarrow{x} [z_1/y_1, \ldots, z_n/y_n]P} \ (\text{R-Proc})$$

$$\frac{P_1 \cong P_2 \quad \mathcal{D}; \Gamma \vdash P_1 \xrightarrow{l} P_1' \quad P_1' \cong P_2'}{\mathcal{D}; \Gamma \vdash P_2 \xrightarrow{l} P_2'} \ (\text{R-Cong})$$

$$\frac{\mathcal{D};\Gamma_1 \vdash P_1 \xrightarrow{l} P_1'}{\mathcal{D};\Gamma_1+\Gamma_2 \vdash P_1 \mid P_2 \xrightarrow{l} P_1' \mid P_2}\ (\text{R-PAR})$$

$$\frac{\mathcal{D};\Gamma, x:[\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P \xrightarrow{x} P' \quad x \notin dom(\mathcal{D})}{\mathcal{D};\Gamma \vdash (\nu x:[\tilde{\tau}]^{(\kappa_1,\kappa_2)})P \xrightarrow{\varepsilon} (\nu x:[\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)})P'}\ (\text{R-NEW1})$$

$$\frac{\mathcal{D};\Gamma, x:[\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P \xrightarrow{l} P' \quad x \neq l \quad x \notin dom(\mathcal{D})}{\mathcal{D};\Gamma \vdash (\nu x:[\tilde{\tau}]^{(\kappa_1,\kappa_2)})P \xrightarrow{l} (\nu x:[\tilde{\tau}]^{(\kappa_1,\kappa_2)})P'}\ (\text{R-NEW2})$$

$$\frac{\mathcal{D}, x[\tilde{y}]:\rho^{(\kappa_{21},\kappa_{22})} = P_1 : \Gamma_1; \Gamma, x:\rho^{(\kappa_{11},\kappa_{12})} \vdash P_2 \xrightarrow{x} P_2'}{\begin{array}{l}\mathcal{D};\Gamma \vdash \textbf{fix}\ x[\tilde{y}]:\rho^{(\kappa_{22}\cdot(\kappa_{12}+1),\kappa_{22}\cdot(\kappa_{12}+1))} = P_1\ \textbf{in}\ P_2\ \textbf{end} \\ \quad \xrightarrow{\varepsilon} \textbf{fix}\ x[\tilde{y}]:\rho^{((\kappa_{22}\cdot(\kappa_{12}+1))^-,(\kappa_{22}\cdot(\kappa_{12}+1))^-)} = P_1\ \textbf{in}\ P_2'\ \textbf{end}\end{array}}\ (\text{R-FIX1})$$

$$\frac{\mathcal{D}, x[\tilde{y}]:\rho^{(\kappa_{21},\kappa_{22})} = P_1 : \Gamma_1; \Gamma, x:\rho^{(\kappa_{11},\kappa_{12})} \vdash P_2 \xrightarrow{l} P_2' \quad x \neq l}{\begin{array}{l}\mathcal{D};\Gamma \vdash \textbf{fix}\ x[\tilde{y}]:\rho^{(\kappa_{22}\cdot(\kappa_{12}+1),\kappa_{22}\cdot(\kappa_{12}+1))} = P_1\ \textbf{in}\ P_2\ \textbf{end} \\ \quad \xrightarrow{l} \textbf{fix}\ x[\tilde{y}]:\rho^{(\kappa_{22}\cdot(\kappa_{12}+1),\kappa_{22}\cdot(\kappa_{12}+1))} = P_1\ \textbf{in}\ P_2'\ \textbf{end}\end{array}}\ (\text{R-FIX2})$$

$$\mathcal{D};\Gamma \vdash \textbf{if}\ true\ \textbf{then}\ P_1\ \textbf{else}\ P_2 \xrightarrow{\varepsilon} P_1\ (\text{R-IFT})$$

$$\mathcal{D};\Gamma \vdash \textbf{if}\ false\ \textbf{then}\ P_1\ \textbf{else}\ P_2 \xrightarrow{\varepsilon} P_2\ (\text{R-IFF})$$

where $1^- = 0$, $\omega^- = \omega$.

Definition 5.4 Γ^{-l} is defined by: $\Gamma^{-\varepsilon} = \Gamma$ and $\Gamma^{-x} = \Gamma'$ such that $dom(\Gamma') = dom(\Gamma)$, $\Gamma'(x) = \rho^{(\kappa_1^-,\kappa_2^-)}$ where $\Gamma(x) = \rho^{(\kappa_1,\kappa_2)}$, and $\Gamma'(y) = \Gamma(y)$ $(x \neq y)$.

Definition 5.5 The relation $\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma'$ is defined by: $\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma'$ if and only if (1) Γ^{-l} is well-defined, (2) $\Gamma' = \Gamma^{-l}$, and (3) $\emptyset;\Gamma \vdash P \xrightarrow{l} P'$ where $\emptyset$ is the empty receptive channel environment.

For example, the process expression $(\nu x:[\tau]^{(1,1)})(y![u] \mid y?[z].x![z] \mid x?[w].v![w])$ is reduced under the type environment $\Gamma, y:[\tau]^{(1,1)}$ by communication on the free channel y:

$$\begin{array}{l}\Gamma, y:[\tau]^{(1,1)} \vdash (\nu x:[\tau]^{(1,1)})(y![u] \mid y?[z].x![z] \mid x?[w].v![w]) \\ \qquad \xrightarrow{y} (\nu x:[\tau]^{(1,1)})(x![u] \mid x?[w].v![w]) \dashv \Gamma, y:[\tau]^{(0,0)}\end{array}$$

It is further reduced by communication on the bound channel x:

$$\Gamma, y:[\tau]^{(0,0)} \vdash (\nu x:[\tau]^{(1,1)})(x![u] \mid x?[w].v![w]) \xrightarrow{\varepsilon} (\nu x:[\tau]^{(0,0)})v![u] \dashv \Gamma, y:[\tau]^{(0,0)}$$

On well-typed processes, our reduction relation coincides with the usual (untyped) reduction relation $\mathcal{D} \vdash P \rightarrow P'$, which is defined inductively by the rules obtained from our rules by dropping all types, type environments and labels:

Theorem 5.6 If $\Gamma \vdash P$, then $\exists \Gamma', l.(\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma')$ if and only if $\emptyset \vdash Erase(P) \rightarrow Erase(P')$ where $Erase(P)$ is an expression obtained from P by erasing all type annotations.

Proof The 'only if' part is trivial. The 'if' part follows from the fact that we can construct a derivation for $\emptyset;\Gamma \vdash P \xrightarrow{l} P'$ from the derivation for $\emptyset \vdash Erase(P) \rightarrow Erase(P')$ and the fact that if $\emptyset;\Gamma \vdash P \xrightarrow{l} P'$ and $\Gamma \vdash P$ then Γ^{-l} is well defined. □

5.2 Correctness about Uses of Channels

As usual, correctness of the type system is shown by the subject reduction theorem (Theorem 5.7), which implies that well-typedness of a process is preserved during reduction, together with the lack of immediate misuse of uses by any well-typed process expressions (Theorem 5.8). By the lack of immediate misuse of uses, we mean, for example, that there is no case where $z : [\tau]^{(0,1)} \vdash P$ but $P \cong (\nu \tilde{x})$ **fix** $y[\tilde{w}] = P_1$ **in fix** $\ldots$ **in** $(z![v_1] \,|\, z![v_2] \,|\, P')$ **end** $\ldots$ **end** (which means that P is now trying to send *two* messages along the channel z though z is declared as a channel used just once for output). These two theorems ensure the lack of misuse of channels in a well-typed process expression during execution.

Theorem 5.7 (Subject Reduction) If $\Gamma \vdash P$ and $\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma'$, then $\Gamma' \vdash P'$.

Proof See Appendix B. □

Theorem 5.8 (Run-time Safety) If $\Gamma \vdash P$ and $P \cong (\nu \tilde{w} : \tilde{\tau})$ **fix** $y_1[\tilde{z}_1] : \tau_{y_1} = P_1'$ **in fix** $\ldots$ **in** $P_1\{\mid P_2\}$ **end** $\ldots$ **end** then[4]:

1. If P_1 is $x![y_1, \ldots, y_n] \,|\, x?[z_1, \ldots, z_m].P'$, then $n = m$ and for the use pair (κ_1, κ_2) of the binding of x in either Γ or ν, $(\kappa_1, \kappa_2) \geq (1, 1)$. x is not bound by **fix**.
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 **fix**) is greater than 0. Moreover, if x is bound by **fix** $x[z_1, \ldots, z_m] = P'$, then $n = m$.
3. If P_1 is $x?[\tilde{y}].P'$, then the input use of the binding of x is greater than 0. x is not bound by **fix**.
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 $x?[\tilde{y}].P' \,|\, x?[\tilde{z}].P''$ then the input use of the binding of x is ω.

Proof Trivial from the typing rules. □

6 Experimental Results

In this section, we show results of simple experiments with HACL compiler[5] to evaluate performance improvement obtained by our analysis. Application programs are the example of Fibonacci function described in the previous section, and concurrent objects expressed by HACL.

Before showing the results, we mention encoding of concurrent objects first.

6.1 Encoding and its Optimization of Concurrent Objects

We explain how concurrent objects are realized in our language, and what optimization we can do for the resulting code. The state of a concurrent object is implemented by 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 back the new state into the channel. So, the following fragment of a process expression corresponds to a typical method definition:

[4] We use the shorthand $\{\mid P\}$ to stand for either nothing or a parallel composition with P.

[5] The HACL compiler translates a HACL program to a C program in a similar manner to sml2c. 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 and reduced time for the benchmark programs

	naive (sec)	optimized (sec)	function (sec)	reduced time/call (μsec)
`sfib25`	1.45	0.57	0.41	3.6
`pfib25`	1.76	0.93	—	3.4
`life`	2.49	2.45	—	17.2
`counter10000`	0.26	0.17	—	9.0
`tree14`	1.55	1.36	—	5.8

fix $m[arg, r] = state?[s].(\ldots \mid r![result] \mid state![news])$
in ...

A caller of the method is typically of the form:

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

With our analysis, it can be translated to **fix** $r[x] = P$ **in** $m![v, r]$ **end** in most cases.

6.2 Experiment Results and Evaluation

We evaluate performance improvement through five kinds of 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 simulation of Conway's life game `life`, a counter increment program `counter10000` (which creates a counter object and increments its value 10000 times), and a tree summation program `tree14` (which creates a binary tree of which each node is a concurrent object, and computes the summation of values of its leaf nodes). Program transformation shown in the previous section has been applied to most of the method invocations of concurrent objects or the function calls.

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 ("optimized") shows the running times of programs optimized by our analysis. The rightmost column ("reduced time/call") shows speedup per one function call or method invocation, which is calculated by (naive − optimized) / (the number of communications over linear channels). In addition, we show in the third column 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[6]: 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%) in this experiment itself should not be taken 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` performs communications less frequently than `sfib25` or `pfib25`, its speedup is much smaller than theirs.

[6]Sun Sparc Station 20 (Hyper SPARC 150Mhz ×1)

The last 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.

7 Extension with Subtyping

Our analysis can be improved by introducing subtyping. An extension with subtyping is sketched very briefly in this section.

Our analysis based on the simple type system sometimes fails to detect some linear channels. It comes mainly from the rule (T-Out), which requires that the bare type of a value sent along a channel must be exactly what is specified by the type of the channel. Sometimes it propagates unnecessary information. For example, consider the following expression:

$$\begin{array}{l}\textbf{fix } x[y] = y![\,] \textbf{ in} \\ \quad \textbf{fix } z[y] = (y![\,] \mid y![\,]) \textbf{ in} \\ \qquad m![x] \mid m![z] \mid (\nu u)(x![u] \mid u?[\,].P) \\ \textbf{end end}\end{array}$$

It should be clear that the channel u is used once for input and output. However, our analysis cannot judge that u is a linear channel. Notice that x and z must have the same bare type because they are sent along the same channel m. From the expression $\textbf{fix } z[y] = (y![\,] \mid y![\,]) \textbf{ in } \cdots$, z must have the bare type $[[\,]^{(0,\omega)}]$; so the bare type of x must also be $[[\,]^{(0,\omega)}]$. Since u is sent along x, the output use of u must be ω.

To solve the above problem, we can introduce a subtyping relation on channel types and refine the rule (T-Out). The rules for the subtyping relation $\tau_1 \preceq_\Theta \tau_2$, and the refined rule for an output atom would be:

$$\frac{\kappa_1 \geq_\Theta \kappa_1' \qquad \kappa_2 \geq_\Theta \kappa_2' \qquad \rho \text{ is } \mathit{bool} \text{ or a type variable}}{\rho^{(\kappa_1,\kappa_2)} \preceq_\Theta \rho^{(\kappa_1',\kappa_2')}} \qquad \text{(Sub-BaseT)}$$

$$\frac{\begin{array}{cc}\kappa_1 \geq_\Theta \kappa_1' & \kappa_2 \geq_\Theta \kappa_2' \\ \kappa_1' \geq_\Theta 1 \Rightarrow (\tau_1 \preceq_\Theta \tau_1' \wedge \cdots \wedge \tau_n \preceq_\Theta \tau_n') & \kappa_2' \geq_\Theta 1 \Rightarrow (\tau_1' \preceq_\Theta \tau_1 \wedge \cdots \wedge \tau_n' \preceq_\Theta \tau_n)\end{array}}{[\tau_1,\ldots,\tau_n]^{(\kappa_1,\kappa_2)} \preceq_\Theta [\tau_1',\ldots,\tau_n']^{(\kappa_1',\kappa_2')}} \qquad \text{(Sub-ChanT)}$$

$$\frac{\tau_x \preceq_\Theta [\tilde{\tau}]^{(0,1)}}{\Gamma + x : \tau_x + y_1 : \tau_1 + \cdots + y_n : \tau_n; \Theta \vdash x![\tilde{y}]} \qquad \text{(T-Out')}$$

The subtyping rule (Sub-ChanT) indicates that an output-only channel type $[\tilde{\tau}]^{(0,\kappa)}$ is contravariant in the argument types $\tau_1,\ldots,\tau_n$, that an input-only channel type is covariant, and that an input-output channel type is invariant (as in Pierce and Sangiorgi's input/output channel type system with subtyping [12]). In the refined type system, we can assign the bare type $[[\,]^{(0,1)}]$ to x and conclude that u is linear.

By extending the syntax of constraints on use variables (with operators $\wedge$ and $\Rightarrow$), we can still define a principal typing and compute it in the same manner as for the simple type system; moreover, extended constraints can also be solved by a similar (but slightly more complex) iterative method. Notice that the resulting algorithm can also be viewed as a type reconstruction algorithm for a variant of the input/output channel type system with subtyping [12].

8 Related Work

Our analysis has its origin in the linear channel type system developed by Kobayashi, Pierce, and Turner [6]. However, their results have been rather theoretical: they were mainly concerned about checking channel usage and reasoning about program behavior. Although they claimed that linear channels would be potentially useful for program optimization, they have neither shown how to detect linear channels nor applied it to actual compilers.

Turner, Wadler, and Mossin [17] 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 [11] proposed another technique that can be used for finding some linear channels based on their effect-based analysis [10]. 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 **fix** $f[] = m?[x].x![] \mid f![]$ **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 [2] 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 [5] proposed a technique for finding linear channels (and *linearized* channels [6]). 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 [8] 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).

9 Conclusions

We have proposed a type-based static analysis method for finding linear channels in concurrent programs. The analysis can be used for performing source-level program transformations (such as the tail-call optimization) and also for reducing run-time costs for communications; indeed, the analysis has been applied to the compiler of a 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 the refinement of our analysis (by using subtyping and polymorphism), and 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 [3]; it is left for future work to utilize such information for program optimization.

Acknowledgement

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.

References

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

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

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

[4] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1997.

[5] 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 *LNCS*, pages 225–242. Springer-Verlag, 1995.

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

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

[8] 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 *LNCS*, pages 312–325. Springer-Verlag, 1991.

[9] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, 1991.

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

[11] 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*, LNCS, pages 590–604. Springer-Verlag, 1995.

[12] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 376–385, 1993.

[13] 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.

[14] 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.

[15] Davide Sangiorgi. The name discipline of uniform receptiveness. draft, 1997.

[16] Davide Sangiorgi. Typed π-calculus at work: a proof of Jones's parallelisation transformation on concurrent objects. In *Fourth Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, 1997.

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

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

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

A Proof of Theorem 4.12

To prove Theorem 4.12, we need several lemmas. At first, the following lemma ensures type-use substitutions preserve the typeability of a process expression.

Lemma A.1 (Type-Use Substitution) If $\Gamma; \Theta \vdash P$, then $S\Gamma; S\Theta \vdash SP$ for any type-use substitutions S.

Proof Straightforward induction on the structure of the proof of $\Gamma, \Theta \vdash P : \tau$. □

Then, the following three lemmas ensure that $\bigoplus$, $\bigsqcup$, $\bigodot$ output the most general type environment and use constraint set:

Lemma A.2 Given $\Gamma_1, \Theta_1, \Gamma_2$, and Θ_2 where every use in Γ_1 and Γ_2 is either a use constant or a use variable, suppose that for some $S', \Gamma', \Gamma'_1, \Gamma'_2, \Theta'$, $\Gamma' \geq_{\Theta'} \Gamma'_1 + \Gamma'_2$, $\Theta' \models S'\Theta_i$, and $\Gamma'_i \supseteq S'\Gamma_i$ (for $i = 1, 2$). Then, $(\Gamma_1, \Theta_1)\bigoplus(\Gamma_2, \Theta_2)$ outputs the triple (S, Γ, Θ) such that (1) $\Gamma \geq_\Theta S\Gamma_1 + S\Gamma_2$, (2) $\Theta \models S\Theta_i$ for $i = 1, 2$, and (3) there exists a substitution S'' such that $S' = S''S$, $\Gamma' \supseteq S''\Gamma$ and $\Theta' \models S''\Theta$. Otherwise, it reports failure.

Proof By the well-definedness of $\Gamma'_1 + \Gamma'_2$ and Theorem 4.9, $U(\{bare(\Gamma_1(x)), bare(\Gamma_2(x)) | x \in dom(\Gamma_1) \cap dom(\Gamma_2)\}$ succeeds and outputs the most general unifier S. By the definition of $\bigoplus$, (1) and (2) are trivial. By Definition 4.8, there exists a substitution S'' such that $S' = S''S$. It is trivial that $\Gamma' \supseteq S''\Gamma$ and $\Theta' \models S''\Theta$. □

Lemma A.3 Given $\Gamma_1, \Theta_1, \Gamma_2$, and Θ_2 where every use in Γ_1 and Γ_2 is either a use constant or a use variable, suppose that for some $S', \Gamma', \Gamma'_1, \Gamma'_2, \Theta'$, $\Gamma' \geq_{\Theta'} \Gamma'_1 \sqcup \Gamma'_2$, $\Theta' \models S'\Theta_i$, and $\Gamma'_i \supseteq S'\Gamma_i$ (for $i = 1, 2$). Then, $(\Gamma_1, \Theta_1)\bigsqcup(\Gamma_2, \Theta_2)$ outputs the triple (S, Γ, Θ) such that (1) $\Gamma \geq_\Theta S\Gamma_1 \sqcup S\Gamma_2$, (2) $\Theta \models S\Theta_i$ for $i = 1, 2$, and (3) there exists a substitution S'' such that $S' = S''S$, $\Gamma' \supseteq S''\Gamma$ and $\Theta' \models S''\Theta$. Otherwise, it reports failure.

Lemma A.4 Given $\Gamma_1, \Theta_1, \kappa$ where every use in Γ_1 is either a use constant or a use variable, suppose that for some $\Gamma'_1, \Gamma'_2, \Theta', \kappa', S$, $\Gamma'_2 \geq_{\Theta'} \kappa' \cdot \Gamma'_1$, $\Gamma'_1 \supseteq S\Gamma_1$, $\Theta' \models S\Theta_1$, $\kappa' = S\kappa$. Then, $\kappa\bigodot(\Gamma_1, \Theta_1) = (\Gamma_2, \Theta_2)$ satisfies: (1) $\Gamma_2 \geq_{\Theta_2} \kappa \cdot \Gamma_1$, (2) $\Theta_2 \models \Theta_1$, (3) $\Gamma'_2 \supseteq S\Gamma_2$, and (4) $\Theta' \models \Theta_2$.

Proof of Lemmas A.3, A.4 are similar to that of Lemma A.2.

Proof of Theorem 4.12 Structural induction on the derivation of $\Gamma'; \Theta' \vdash_{\mathcal{STU}} SP$, with case analysis on the last rule used.

(ST-Out) Suppose that $\Gamma'; \Theta' \vdash x![y_1, \ldots, y_n]$. By the rule (ST-Out), $\Gamma' \geq_{\Theta'} x : [\rho_1^{(\kappa_{11}, \kappa_{12})}, \ldots, \rho_n^{(\kappa_{n1}, \kappa_{n2})}]^{(\kappa_1, \kappa_2)} + y_1 : \rho_1^{(\kappa'_{11}, \kappa'_{12})} + \cdots + y_n : \rho_n^{(\kappa'_{n1}, \kappa'_{n2})}$ such that $\kappa'_{i1} \geq_{\Theta'} \kappa_{i1}$ and $\kappa'_{i2} \geq_{\Theta'} \kappa_{i2}$ for all $i \in \{1, \ldots, n\}$. Let $S = (S_\rho, S_\kappa)$ where $S_\rho = [\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]$ and $S_\kappa = [\kappa_{11}/j_{11}, \ldots, \kappa_{n2}/j_{n2}, \kappa'_{11}/k_{11}, \ldots, \kappa'_{n2}/k_{n2}]$. By Lemma A.2, $(x : [\alpha_1^{(j_{11}, j_{12})}, \ldots, \alpha_n^{(j_{n1}, j_{n2})}]^{(j,k)}, \emptyset)\bigoplus(y_1 : \alpha_1^{(k_{11}, k_{12})}, \emptyset)\bigoplus \cdots \bigoplus(y_n : \alpha_n^{(k_{n1}, k_{n2})}, \emptyset)$ succeeds and outputs (S', Γ, Θ) such that: (1) $\Gamma \geq_\Theta S'(x : [\alpha_1^{(j_{11}, j_{12})}, \ldots, \alpha_n^{(j_{n1}, j_{n2})}]^{(j,k)}) + S'(y_1 : \alpha_1^{(k_{11}, k_{12})}) + \cdots + S'(y_n : \alpha_n^{(k_{n1}, k_{n2})})$, (2) there exists S'' such that $S = S''S'$, $\Gamma' \supseteq S''\Gamma$, and $\Theta' \models S''\Theta$. Let Θ'' be $\Theta \cup S'(\{k \geq 1\} \cup \{k_{ij} \geq j_{ij} | 1 \geq i \geq n, j = 1, 2\})$. Then, (1) $\Gamma; \Theta'' \vdash x![y_1, \ldots, y_n]$, and (2) S'' satisfies $\Theta' \models S''\Theta''$ (since $\Theta' \models S''\Theta$ and $\Theta' \models S''(\{k \geq 1\} \cup \{k_{ij} \geq j_{ij} | 1 \leq i \leq n, j = 1, 2\})$ and $\Gamma' \supseteq S''\Gamma$.

(ST-Par) Suppose that $\Gamma'; \Theta' \vdash S(P_1 \mid P_2)$ for some substitution S. From the rule (ST-Par), we have $\Gamma'_1; \Theta'_1 \vdash SP_1$, $\Gamma'_2; \Theta'_2 \vdash SP_2$, $\Gamma' \geq_{\Theta'} \Gamma'_1 + \Gamma'_2$, and $\Theta' \models S\Theta'_i$ for $i = 1, 2$. By induction hypothesis, $PTU(P_1)$ and $PTU(P_2)$ succeeds and outputs their prinicpal typings $(\Gamma_1, \Theta_1, P'_1)$ and $(\Gamma_2, \Theta_2, P'_2)$, respectively. By Definition 4.3, for $i = 1, 2$, $S'_i P_i = P'_i$ for some S'_i, $\Gamma_i; \Theta_i \vdash P'_i$, and there exists a substitution S_i such that $SP_i = S_i P'_i$, $\Theta'_i \models S_i\Theta_i$, and $\Gamma'_i \supseteq S_i\Gamma_i$. Let $S' = S_1 S_2$. S' satisfies the premises of Lemma A.2. Therefore, $(\Gamma_1, \Theta_1) \bigoplus (\Gamma_2, \Theta_2)$ succeeds and outputs (S'', Γ, Θ) such that $\Gamma \geq_\Theta S''\Gamma_1 + S''\Gamma_2$, $\Theta \models S''\Theta_i$ for $i = 1, 2$. Then, $(\Gamma, \Theta, S''(P'_1 \mid P'_2))$ is the principal typing: by Lemma A.2, A.1, and the rule (ST-Par), $\Gamma; \Theta \vdash S''(P'_1 \mid P'_2)$. By Lemma A.2, there exists S''' such that $S' = S'''S''$, $\Gamma' \supseteq S'''\Gamma$, and $\Theta' \models S'''\Theta$.

(ST-If) Suppose that $\Gamma'; \Theta' \vdash S(\textbf{if } x \textbf{ then } P_1 \textbf{ else } P_2)$ for some substitution S. From the rule (ST-If), we have $\Gamma'_1; \Theta'_1 \vdash SP_1$, $\Gamma'_2; \Theta'_2 \vdash SP_2$, $\Gamma' \geq_{\Theta'} x : \mathit{bool} + (\Gamma'_1 \sqcup \Gamma'_2)$, and $\Theta' \models \Theta'_i$ for $i = 1, 2$. By induction hypothesis, $PTU(P_1)$ and $PTU(P_2)$ succeeds and outputs their principal typings $(\Gamma_1, \Theta_1, P'_1)$ and $(\Gamma_2, \Theta_2, P'_2)$, respectively. By Definition 4.3, for $i = 1, 2$, $S'_i P_i = P'_i$ for some S'_i, $\Gamma_i; \Theta_i \vdash P'_i$, there exists a substitution S_i such that $SP_i = S_i P'_i$, $\Theta'_i \models S_i\Theta_i$, and $\Gamma'_i \supseteq S_i\Gamma_i$. Let $S' = S_1 S_2$. By Lemma A.3 and Lemma A.2, $(\Gamma_1, \Theta_1) \bigsqcup (\Gamma_2, \Theta_2)$ succeeds and outputs $(S_3, \Gamma_3, \Theta_3)$, and then $(\Gamma_3, \Theta_3) \bigoplus (x : \mathit{bool}, \emptyset)$ suceeds and outputs $(S_4, \Gamma_4, \Theta_4)$. They satisfy $\Gamma_4 \geq_{\Theta_4} S_4\mathit{bool} + S_4(S_3\Gamma_1 \sqcup S_3\Gamma_2)$, $\Theta_4 \models S_4 S_3 \Theta_i$ for $i = 1, 2$. Moreover, there exists a substitution S_5 such that $S' = S_5 S_4 S_3$, $\Gamma' \supseteq S_5\Gamma_4$, and $\Theta' \models S_5\Theta_4$. Thus, $(\Gamma_4, \Theta_4, S_4 S_3(\textbf{if } x \textbf{ then } P_1 \textbf{ else } P_2))$ is the principal typing.

(ST-In) Suppose that $\Gamma''; \Theta'' \vdash S(x?[y_1, \ldots, y_n].P)$ for some substitution S. By the rule (ST-In), we obtain $\Gamma', y_1 : \rho_1^{(\kappa_{11}, \kappa_{12})}, \ldots, y_n : \rho_n^{(\kappa_{n1}, \kappa_{n2})}; \Theta' \vdash SP$, $\Gamma'' \geq_{\Theta''} \Gamma' + x : [\rho_1^{(\kappa_{11}, \kappa_{12})}, \ldots, \rho_n^{(\kappa_{n1}, \kappa_{n2})}]^{(\kappa'_1, \kappa'_2)}$, $\kappa'_1 \geq_{\Theta'} 1$, and $\Theta'' \models \Theta'$. By induction hypothesis, $PTU(P)$ succeeds without failure and outputs the principal typing (Γ, Θ, P'). Then, by Definition 4.3, $S''P = P'$ for some S'', $\Gamma; \Theta \vdash P'$, and there exists a substitution S' such that $SP = S'P'$, $\Theta' \models S'\Theta$, and $\Gamma', y_i : \rho_1^{(\kappa'_{11}, \kappa'_{12})}, \ldots, y_n : \rho_n^{(\kappa'_{n1}, \kappa'_{n2})} \supseteq S'\Gamma$. Let $S_0 = S'([\rho_1/\alpha_{y_1}, \ldots, \rho_n/\alpha_{y_n}], [\kappa_1/j, \kappa_2/k, \kappa_{11}/j_{y_1}, \ldots, \kappa_{n1}/j_{y_n}, \kappa_{12}/k_{y_1}, \ldots, \kappa_{n2}/k_{y_n}])$, and τ_i be $\Gamma(y_i)$ if $y_i \in \mathit{dom}(\Gamma)$, or $\alpha_{y_i}^{(j_{y_i}, k_{y_i})}$ otherwise for $i \in \{1, \ldots, n\}$. By Lemma A.2, $(x : [\tau_1, \ldots, \tau_n]^{(j,k)}, \{j \geq 1\}) \bigoplus (\Gamma \setminus \{y_1, \ldots, y_n\}, \Theta)$ succeeds and outputs $(S_1, \Gamma_1, \Theta_1)$ such that: (1) $\Gamma_1 \geq_{\Theta_1} S_1(x : [\tau_1, \ldots, \tau_n]^{(j,k)}) + S_1(\Gamma \setminus \{y_1, \ldots, y_n\})$, (2) $\Theta_1 \models S_1\Theta$, (3) $\Theta_1 \models S_1\{j \geq 1\}$, and (4) there exists S_2 such that $S_0 = S_2 S_1$, $\Gamma' \supseteq S_2\Gamma_1$, and $\Theta' \models S_2\Theta_1$. We have $S_1(\Gamma \setminus \{y_1, \ldots, y_n\}, y_1 : \tau_1, \ldots, y_n : \tau_n); S_1\Theta \vdash S_1 P$ by Lemma A.1, and then $\Gamma_1; \Theta_1 \vdash S_1(x?[\tilde{y}].P)$ by the rule (ST-In). Thus, $(\Gamma_1, \Theta_1, S_1(x?[y_1, \ldots, y_n].P))$ is the principal typing.

(ST-Proc) Suppose that $\Gamma'; \Theta' \vdash S(\textbf{fix } x[\tilde{y}] : \rho^{(\kappa,\kappa)} = P_1 \textbf{ in } P_2 \textbf{ end})$ for some substitution S. By the rule (ST-Proc), we obtain $\Gamma'_1, x : [\tilde{\tau}]^{(\kappa'_{11}, \kappa'_{12})}, \tilde{y} : \tilde{\tau}; \Theta'_1 \vdash SP_1$, $\Gamma'_2, x : [\tilde{\tau}]^{(\kappa'_{21}, \kappa'_{22})}; \Theta'_2 \vdash P_2$, $S\kappa \geq_{\Theta'} \kappa'_{22} \cdot (\kappa'_{12} + 1)$, $0 \geq_{\Theta'} \kappa'_{11}$, $0 \geq_{\Theta'} \kappa'_{21}$, $\Gamma' \geq_{\Theta'} (\kappa \cdot \Gamma'_1) + \Gamma'_2$, $S\rho = [\tilde{\tau}]$, and $\Theta' \models \Theta'_i$ for $i = 1, 2$. By induction hypothesis, $PTU(P_1)$ and $PTU(P_2)$ succeeds and outputs their principal typings $(\Gamma_1, \Theta_1, P'_1)$ and $(\Gamma_2, \Theta_2, P'_2)$, respectively. By Definition 4.3, for $i = 1, 2$, $S'_i P_i = P'_i$ for some S'_i, $\Gamma_i; \Theta_i \vdash P'_i$, there exists a substitution S_i such that $SP_i = S_i P'_i$, $\Theta'_i \models S_i\Theta_i$, and $\Gamma'_i \supseteq S_i\Gamma_i$. Let $S_0 = SS_1S_2(S_\rho, S_\kappa)$ where $S_\rho = [[\tilde{\tau}]/\alpha_1, [\tilde{\tau}]/\alpha_2, \mathit{bare}(\tau_1)/\alpha_{y_1}, \ldots, \mathit{bare}(\tau_n)/\alpha_{y_n}]$ and $S_\kappa = [\kappa'_{11}/j_1, \kappa'_{12}/k_1, \kappa'_{21}/j_2, \kappa'_{22}/k_2, \mathcal{I}(\tau_1)/j_{y_1}, \ldots, \mathcal{I}(\tau_n)/j_{y_n}, \mathcal{O}(\tau_1)/k_{y_1}, \ldots, \mathcal{O}(\tau_n)/k_{y_n}]$. By Lemma A.4 and Lemma A.2, $j \bigodot (\Gamma_1 \setminus \{x, \tilde{y}\})$ succeeds and outputs (Γ_3, Θ_3), and $(\Gamma_3, \Theta_3) \bigoplus (\Gamma_2 \setminus \{x\}, \Theta_2)$ succeeds and outputs $(S_4, \Gamma_4, \Theta_4)$. They satisfy $\Gamma_4 \geq_{\Theta_4}$

$S_4(j \cdot \Gamma_1 \setminus \{x, \tilde{y}\}) + S_4(\Gamma_2 \setminus \{x\})$, $\Theta_4 \models S_4\Theta_i$ for $i = 1, 2$. Moreover, there exists a substitution S_5 such that $S_0 = S_5S_4$, $\Gamma' \supseteq S_5\Gamma_4$, and $\Theta' \models S_5\Theta_4$. Let $E = \{(S_4\rho_1^{(j,j)}, S_4\rho^{(\kappa,\kappa)}), (S_4\rho_1^{(j,j)}, S_4\rho_2^{(j,j)}), (S_4\rho_2^{(j,j)}, S_4[\tau_1, \ldots, \tau_n]^{(j,j)})\}$ where τ_i is $\Gamma_1(y_i)$ if $y_i \in dom(\Gamma_1)$, or $\alpha_{y_i}^{(j_{y_i}, k_{y_i})}$ otherwise (for $i \in \{1, \ldots, n\}$) and $\rho_i^{(\kappa_{i1}, \kappa_{i2})} = \Gamma_i(x)$ if $x \in dom(\Gamma_i)$, or $\alpha_i^{(j_i, k_i)}$ otherwise (for $i = 1, 2$). By Theorem 4.9, $U(E)$ succeeds and outputs the most general unifier S_6 and $S_5 = S_7S_6$ for some substitution S_7. Then, by Lemma 4.5, 4.4, and A.1, $S_6S_4(\Gamma_1 \setminus \{x, \tilde{y}\}), x : S_6S_4\rho_1^{(j,j)}, \tilde{y} : S_6S_4\tilde{\tau}; S_6\Theta_5 \vdash S_6S_4P_1'$, and $S_6S_4(\Gamma_2 \setminus \{x\}), x : S_6S_4\rho_2^{(j,j)}; S_6\Theta_5 \vdash S_6S_4P_2'$ where $\Theta_5 = \Theta_4 \cup \{0 \geq \kappa_{11}, 0 \geq \kappa_{21}, j \geq \kappa_{22} \cdot (\kappa_{12} + 1)\}$. Thus, $S_6\Gamma_4; S_6\Theta_5 \vdash S_6S_4(\mathbf{fix}\ x[\tilde{y}] : \rho^{(j,j)} = P_1'\ \mathbf{in}\ P_2'\ \mathbf{end})$ and $(S_6\Gamma_4, S_6\Theta_5, S_6S_4(\mathbf{fix}\ x[\tilde{y}] : \tau = P_1'\ \mathbf{in}\ P_2'\ \mathbf{end}))$ is the principal typing.

(ST-New) Suppose that $\Gamma'; \Theta' \vdash S((\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P)$. From the rule (ST-New), we have $\Gamma', x : S([\tilde{\tau}]^{(\kappa_1,\kappa_2)}); \Theta' \vdash SP$. By induction hypothesis, $PTU(P)$ succeeds without failure and outputs (Γ, Θ, P'). By Definition 4.3, $P' = S'P$ for some S', $\Gamma; \Theta \vdash P'$, and there exists a substitution S'' such that $SP = S''P''$, $\Gamma', x : S([\tilde{\tau}]^{(\kappa_1,\kappa_2)}) \supseteq S''\Gamma$, and $\Theta' \models S''\Theta$. Because variables output by PTU contains only fresh ones, $SS'' = S''S$.

Consider the case that $x \in dom(\Gamma)$. By Theorem 4.9, $U(\{\Gamma(x), [\tilde{\tau}]^{(\kappa_1,\kappa_2)})\})$ succeeds and outputs S_3. Therefore, $S_3(\Gamma \setminus \{x\}); S_3\Theta \vdash (\nu x : S_3\Gamma(x))P$ is derived by Lemma A.1 and the rule (ST-New). By the property of the most general unifier, $SS'' = S_4S_3$ for some S_4 such that $\Gamma' \supseteq S_4S_3(\Gamma \setminus \{x\})$ and $\Theta' \models S_4S_3\Theta$.

The case of $x \notin dom(\Gamma)$ is similar.

□

B Proof of Subject Reduction Theorem (Theorem 5.7)

To prove Theorem 5.7, we need to strengthen the theorem so that induction works. Then, Theorem 5.7 is then obtained as a special case. Before stating the strengthened theorem (Theorem B.9), we give several preliminary definitions and lemmas.

Definition B.5 $Rem(\Gamma, \mathcal{D}, l)$ is the type environment defined by: $Rem(\Gamma, \mathcal{D}, \varepsilon) = \Gamma$, $Rem(\Gamma, \mathcal{D}, x) = \Gamma^{-x}$ if $x \notin dom(\mathcal{D})$, and $Rem(\Gamma, \mathcal{D}, x) = ((\Gamma \setminus \{x\}) + \Gamma_x), x : \rho^{(\kappa_1 + \kappa_1', \kappa_2^- + \kappa_2')})$ where $\Gamma(x) = \rho^{(\kappa_1,\kappa_2)}$ and $\mathcal{D}(x) = (\tilde{y}, \rho^{(\kappa_1',\kappa_2')}, P, \Gamma_x)$.

Definition B.6 The 3-place relation $\cdot; \cdot \vdash \cdot$ is defined by: $\mathcal{D}; \Gamma \vdash P$ if and only if (1) $dom(\mathcal{D}) \subseteq dom(\Gamma)$, (2) $\mathcal{D} = x_1[\tilde{y_1}] : \tau_1 = P_1 : \Gamma_1, \ldots, x_n[\tilde{y_n}] : \tau_n = P_n : \Gamma_n$, (3) $\Gamma \vdash P$, (4) for all $x \in dom(\mathcal{D})$, $0 \geq \mathcal{I}(\Gamma(x))$, and (5) $\Gamma_i, x_i : \tau_i, \tilde{y} : \tilde{\tau}_y \vdash P_i$ where $bare(\tau_i) = [\tau_{y_1}, \ldots, \tau_{y_n}]$ and $0 \geq \mathcal{I}(\tau_i)$ for $i \in \{1, \ldots, n\}$.

Lemma B.7 (Variable Substitution) If $\Gamma, z : \rho^{(\kappa_{11},\kappa_{12})}, y : \rho^{(\kappa_{21},\kappa_{22})}; \Theta \vdash P$, then $\Gamma, z : \rho^{(\kappa_{11}+\kappa_{21}, \kappa_{12}+\kappa_{22})}; \Theta \vdash [z/y]P$.

Proof Structural induction on the proof of $\Gamma; \Theta \vdash P$. □

Lemma B.8 If $P_1 \cong P_2$, then $\Gamma; \Theta \vdash P_1$ if and only if $\Gamma; \Theta \vdash P_2$.

Proof Structural induction on the proof of $P_1 \cong P_2$ □

Theorem B.9 If $\mathcal{D}; \Gamma \vdash P$ and $\mathcal{D}; \Gamma \vdash P \xrightarrow{l} P'$, then $Rem(\Gamma, \mathcal{D}, l)$ is well-defined and $\mathcal{D}; Rem(\Gamma, \mathcal{D}, l) \vdash P$.

Proof Structural induction on the proof of $\mathcal{D};\Gamma \vdash P \xrightarrow{l} P'$ with case analysis by the last rule used.

(R-Comm) Suppose $\mathcal{D};\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash x[\tilde{y}].P \mid x![\tilde{z}]$. By Definition B.6, $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P$ must be satisfied. By syntax-directed typing rules, we obtain $\Gamma_1, x : [\tilde{\tau}]^{(\kappa_{111},\kappa_{211})}, \tilde{y} : \tilde{\tau} \vdash P$, $x : [\tilde{\tau}]^{(\kappa_{12},\kappa_{22})} + z_1 : \tau_1' + \cdots + z_n : \tau_n' \vdash x![\tilde{z}]$, $\kappa_{22} \geq 1$, $\kappa_{112} \geq 1$, $\kappa_i \geq \kappa_{i11} + \kappa_{i12} + \kappa_{i2}$ for $i = 1, 2$, $\Gamma \geq \Gamma_1 + (x : [\tilde{\tau}]^{(\kappa_{12},\kappa_{22})} + z_1 : \tau_1' + \cdots + z_n : \tau_n') \setminus \{x\}$, and $\tau_i' \geq \tau_i$ for $i = \{1, \ldots, n\}$. We have four cases by κ_1 and κ_2. Consider the case where $\kappa_1 = \kappa_2 = 1$. From the above conditions, $\kappa_{22} = \kappa_{112} = 1$, $\kappa_{111} = \kappa_{12} = \kappa_{211} = \kappa_{212} = 0$. By Lemma 4.5, and B.7, $\Gamma, x : [\tilde{\tau}]^{(0,0)} \vdash [\tilde{z}/\tilde{y}]P$. Because $x \notin dom(\mathcal{D})$, $\mathcal{D}; Rem((\Gamma, x : [\tilde{\tau}]^{(1,1)}), \mathcal{D}, x) \vdash [\tilde{z}/\tilde{y}]P$ holds. The other cases are similar.

(R-Proc) Suppose $\mathcal{D};\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash x![\tilde{y}]$, $\kappa_2 \geq 1$, and $\mathcal{D}(x) = (\tilde{z}, [\tilde{\tau}]^{(\kappa_1',\kappa_2')}, P, \Gamma_x)$. By Definition B.6 and the rule (ST-Out), we obtain $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \geq x : [\tilde{\tau}]^{(\kappa_1'',\kappa_2'')} + y_1 : \tau_1' + \cdots y_n : \tau_n'$, where $\tau_i' \geq \tau_i$ for $i = \{1, \ldots, n\}$, $\kappa_2'' \geq 1$, and $\Gamma_x, x : [\tilde{\tau}]^{(\kappa_1',\kappa_2')}, \tilde{z} : \tilde{\tau} \vdash P$. By Lemma 4.5 and Lemma B.7, $(\Gamma + \Gamma_x), x : \rho^{(\kappa_1+\kappa_1', \kappa_2^- + \kappa_2')} \vdash [\tilde{y}/\tilde{z}]P$. Because $x \in dom(\mathcal{D})$, $\mathcal{D}; Rem((\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}), \mathcal{D}, x) \vdash [\tilde{y}/\tilde{z}]P$.

(R-Cong)**:** Follows from Lemma B.8.

(R-Par) Suppose $\mathcal{D};\Gamma \vdash P_1 \mid P_2$. By Definition B.6, we obtain $\Gamma \vdash P_1 \mid P_2$. From the rule (ST-Par), for some Γ_1 and Γ_2 such that $\Gamma \geq \Gamma_1 + \Gamma_2$, $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$. So, we obtain $\mathcal{D};\Gamma_1 \vdash P_1$. By induction hypothesis, $Rem(\Gamma_1, \mathcal{D}, x)$ is well-defined and $\mathcal{D}; Rem(\Gamma_1, \mathcal{D}, x) \vdash P_1'$ holds. It is trivial that $Rem(\Gamma_1 + \Gamma_2, \mathcal{D}, x) \geq Rem(\Gamma_1, \mathcal{D}, x) + \Gamma_2$. Hence, $Rem(\Gamma, \mathcal{D}, x) \vdash P_1' \mid P_2$ by Lemma 4.5 and so $\mathcal{D}; Rem(\Gamma, \mathcal{D}, x) \vdash P_1' \mid P_2$.

(R-New1) Suppose $\mathcal{D};\Gamma \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P$. By Definition B.6, $\Gamma \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P$. From the rule (ST-New), $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P$. So $\mathcal{D};\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P$. By induction hypothesis, $\mathcal{D}; Rem(\Gamma, x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)}, \mathcal{D}, x) \vdash P'$ and so $\Gamma, x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)} \vdash P'$. Therefore, $\Gamma (= Rem(\Gamma, \mathcal{D}, \varepsilon)) \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)})P'$. $dom(\mathcal{D}) \subseteq dom(\Gamma \setminus \{x\})$ holds since $x \notin dom(\mathcal{D})$. Therefore, $\mathcal{D}; Rem(\Gamma, \mathcal{D}, \varepsilon) \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)})P'$.

(R-New2) Suppose $\mathcal{D};\Gamma \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P$. By Definition B.6, $\Gamma \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}P$. From the rule (ST-New), $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P$ and so $\mathcal{D};\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash P$. By induction hypothesis, $\mathcal{D}; Rem((\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}), \mathcal{D}, l) \vdash P$ holds. Because $l \neq x$, $Rem((\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}), \mathcal{D}, l) = Rem(\Gamma, \mathcal{D}, l), x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}$ and so $Rem(\Gamma, \mathcal{D}, l) \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P'$. $dom(\mathcal{D}) \subseteq dom(\Gamma \setminus \{x\})$ holds since $x \notin dom(\mathcal{D})$. Therefore, $\mathcal{D}; Rem(\Gamma, \mathcal{D}, l) \vdash (\nu x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})P'$.

(R-Fix1) Suppose $\mathcal{D};\Gamma \vdash \textbf{fix}\ x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$. By Definition B.6, $\Gamma \vdash \textbf{fix}\ x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$. From the rule (ST-Fix), we obtain $\Gamma, x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})}, \tilde{y} : \tilde{\tau} \vdash P_1, \Gamma, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})} \vdash P_2$, $\kappa \geq \kappa_{22} \cdot (\kappa_{12} + 1)$, $\Gamma \geq (\kappa \cdot \Gamma_1) + \Gamma_2$, $0 \geq \kappa_{11}$, and $0 \geq \kappa_{21}$. Let $\Gamma' = \Gamma, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})}$ and $\mathcal{D}' = \mathcal{D}, x[\tilde{y}] : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})} = P_1 : \Gamma_1$. We obtain $\mathcal{D}';\Gamma' \vdash P_2$. By induction hypothesis, $\mathcal{D}'; Rem(\Gamma', \mathcal{D}', x); \vdash P_2$ and so $\Gamma, x : [\tilde{\tau}]^{(\kappa_{21}+\kappa_{11}, \kappa_{22}^- + \kappa_{12})} \vdash P_2$. By the well-definedness of $Rem(\Gamma', \mathcal{D}', x)$, $\kappa_{22} \geq 1$ holds. Therefore, $(\kappa_{22} \cdot (1 + \kappa_{12}))^- = (\kappa_{22}^- + \kappa_{12}) \cdot (1 + \kappa_{12})$. Then, $\Gamma \vdash \textbf{fix}\ x[\tilde{y}] : [\tilde{\tau}]^{(\kappa^-,\kappa^-)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$ and so $\mathcal{D}; Rem(\Gamma, \mathcal{D}, \varepsilon) \vdash \textbf{fix}\ x[\tilde{y}] : [\tilde{\tau}]^{(\kappa^-,\kappa^-)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$.

(R-Fix2) Suppose $\mathcal{D};\Gamma \vdash \textbf{fix}\ x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$. By Definition B.6, $\Gamma \vdash \textbf{fix}\ x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1\ \textbf{in}\ P_2\ \textbf{end}$. From the rule (ST-Fix), we obtain $\Gamma, x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})}, \tilde{y} : \tilde{\tau} \vdash P_1$, $\Gamma, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})} \vdash P_2$, $\kappa \geq \kappa_{22} \cdot (\kappa_{12} + 1)$, $\Gamma \geq (\kappa \cdot \Gamma_1) + \Gamma_2$, $0 \geq \kappa_{11}$,

and $0 \geq \kappa_{21}$. Let $\Gamma' = \Gamma, x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})}$ and $\mathcal{D}' = \mathcal{D}, x[\tilde{y}] : [\tilde{\tau}]^{(\kappa_{11},\kappa_{12})} = P_1 : \Gamma_1$. We obtain $\mathcal{D}'; \Gamma' \vdash P_2$. By induction hypothesis, $\mathcal{D}'; Rem(\Gamma', \mathcal{D}', l) \vdash P_2$ and so $Rem(\Gamma', \mathcal{D}', l) \vdash P_2$. Since $l \neq x$, $Rem(\Gamma', \mathcal{D}', l) = Rem(\Gamma, \mathcal{D}, l), x : [\tilde{\tau}]^{(\kappa_{21},\kappa_{22})}$. Then, $Rem(\Gamma, \mathcal{D}, l) \vdash$ **fix** $x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1$ **in** P_2 **end** and so $\mathcal{D}; Rem(\Gamma, \mathcal{D}, l) \vdash$ **fix** $x[\tilde{y}] : [\tilde{\tau}]^{(\kappa,\kappa)} = P_1$ **in** P_2 **end**.

(R-IfT) Suppose $\mathcal{D}; \Gamma \vdash$ **if** *true* **then** P_1 **else** P_2. By Definition B.6, we obtain $\Gamma \vdash$ **if** *true* **then** P_1 **else** P_2. From the rule (ST-If), $\Gamma_1 \vdash P_1$, $\Gamma_2 \vdash P_2$, and $\Gamma \geq \Gamma_1 \sqcup \Gamma_2$. By Lemma 4.5, we obtain $\Gamma \vdash P_1$ and $\mathcal{D}; Rem(\Gamma, \mathcal{D}, \varepsilon) \vdash P_1$.

(R-IfF) Similar to the case of the rule (R-IfT).

□

Proof of Theorem 5.7 Suppose $\Gamma \vdash P$ (i.e., $\emptyset; \Gamma \vdash P$) and $\Gamma \vdash P \xrightarrow{l} P' \dashv \Gamma'$. By Theorem B.9, $Rem(\Gamma, \emptyset, l) = \Gamma^{-l} = \Gamma'$ is well-defined and $Rem(\Gamma, \emptyset, l) \vdash P'$. □

C Benchmark Programs

This section shows the programs used in the experiments of Section 6 (except `life`). The benchmark programs are translated to those written in the target language of this paper (extended with several mathematical operations, 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.

C.1 Fibonacci Functions

Naive and optimized sequential fibs are described as the example of our analysis (Section 4.5). Naive parallel fib:

fix $pfib[n, c] =$
  if $n < 2$ **then** $c![1]$
  else $(\nu c_1)(\nu c_2)(pfib![n-1, c_1] \mid pfib![n-2, c_2] \mid c_1?[x].c_2?[y].c![x+y])$
in $pfib![25, output]$ **end**

is optimized to the following expression:

fix $pfibopt(n, c) =$
  if $n < 2$ **then** $c![1]$
  else (νc_2)(**fix** $c_1[x] = c_2[y].c![x+y]$ **in**
    $pfibopt![n-1, c_1] | pfibopt![n-2, c_2]$ **end**)
in $pfibopt![25, output]$ **end**

C.2 Counter Object

The naive implementation of a counter object is:

fix $newcounter[init, c] =$
  $(\nu st)(st![init]$
    | **fix** $g[c_1] = st?[x].(c_1![x] \mid st![x])$ **in** *(* get the state of the counter *)*
      fix $i[ack] = st?[x].(ack![] \mid st![x+1])$ **in** *(* increment the counter *)*

```
                c![{get = g, inc = i}] end end)
  in fix incr[n, o, ack] =        (* invoke inc method n times *)
       if n = 0 then ack![]
       else (νr)(o.inc![r] | r?[].(incr![n − 1, o, ack]))
  in (νr1)(newcounter![0, r1]
         | r1?[o].(νr2)(incr![1000000, o, r2] | r2?[].(o.get![output])))
  end
```

Then, the above program is optimized to:

```
  fix newcounter[init, c] =
    (νst)(st![init]
       | fix g[c1] = st?[x].(c1![x] | st![x]) in
         fix i[ack] = st?[x].(ack![] | st![x + 1]) in
              c![{get = g, inc = i}] end end)
  in fix incr[n, o, ack] =
       if n = 0 then ack![]
       else fix r[] = incr![n − 1, o, ack] in o.inc![r] end
  in fix r1[o] = fix r2[] = o.get![output] in incr![1000000, o, r2] end
              in newcounter![0, r1] end
  end
```

C.3 Tree Summation

The naive implementation of tree object is:

```
  fix newnode[n, left, right, rep] =      (* n=0 : node object, otherwise : leaf object holding n *)
    (νst)(st![n, left, right]
       | fix s[c] =
           st?[n, l, r].(st![n, l, r]
                | if n <> 0 then c![n]
                  else (νr1)(νr2)(r.sum![r1] | l.sum![r2] | r1?[x1].r2?[x2].c![x1 + x2]))
       in rep![{sum = s}] end)
  in fix gentree[n, rep] =
       if n = 0 then (νl)(νr)newnode![1, l, r, rep]
       else (νr1)(νr2)(gentree![n − 1, r1] | gentree![n − 1, r2]
                | r1?[left].r2?[right].newnode![0, left, right, rep])
  in (νr)(gentree[14, r] | r?[o].(o.sum![output])) end
```

Then, the optimized program is:

```
fix newnode[n, left, right, rep] =
  (νst)(st![n, left, right]
     | fix s[c] =
         st?[n, l, r].(st![n, l, r]
              | if n <> 0 then c![n]
                else (νr2) fix r1[x1] = r2?[x2].c![x1 + x2]
                           in r.sum![r1] | l.sum![r2] end
     in rep![{sum = s}] end)
in fix gentree[n, rep] =
     if n = 0 then (νr1)(νr2)newnode![1, r1, r2, rep]
```

$$\begin{array}{l}
\quad\textbf{else } (\nu r_2)(\textbf{fix } r_1[left] = c_2?[right].newnode![0, left, right, rep] \\
\qquad\qquad\textbf{in } gentree![n-1, r_1] \,|\, gentree![n-1, r_2] \textbf{ end}) \\
\textbf{in fix } r[o] = o.sum![output] \textbf{ in } gentree[14, r] \textbf{ end end}
\end{array}$$