Type-based Analysis of Communication for Concurrent Programming Languages

Atsushi Igarashi* and Naoki Kobayashi

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

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.

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 [12, 2], checking usage of communication channels [14, 8],

* Research Fellow of the Japan Society for the Promotion of Science.

detecting deadlock [6], etc. Among them, the linear channel type system [8], 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:

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

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 [16]) 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 [8].); 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:

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

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 $(\nu r)\,(f![n, r] \mid r?[].\, r'![])$ with the more efficient process $f![n, r']$. As we have argued elsewhere [7, 8], this optimization corresponds to tail-call optimization of functions.

However, the previous linear type system [8] 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:

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

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:

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

The use constraint sets specify the values taken by use variables ($j, k, \ldots$). In the process $f![n,r] \mid r?[].\, r'![]$, 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 [8], 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 [16], Pict [15], and HACL [9] 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 [16, 19, 18, 15, 9]: 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]).

The rest of this paper is organized as follows. Section 2 introduces 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 briefly mentions 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. After discussing related work in Section 7, we conclude in Section 8. Proofs omitted in this paper can be found in an accompanying technical report [4].

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 [11], and it is close to the core languages of HACL [9] and Pict [15].

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.

Definition 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:

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

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 $bool^{(\omega,\omega)}$.

Several operations on use constants and types are defined below.

Definition 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 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 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 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

Definition 6. The set of *process expressions*, ranged over by P, is defined by:

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

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.[2] 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 P_1 the body of the process definition. **if** x **then** P_1 **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 $|$; for example, $x?[z].y![z] \,|\, (\nu w : \tau)P_1 \,|\, P_2$ means $(x?[z].y![z]) \,|\, ((\nu w : \tau)P_1) \,|\, P_2$. 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 **fix** $x[\tilde{y}] = P_1$ **in** P_2 **end** and, (2) variables $y_1, \ldots, y_n$ are bound in P of $x?[y_1, \ldots, y_n].P$ and **fix** $x[y_1, \ldots, y_n] = P$ **in** P_2 **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 $dom(\Gamma) = \{x_1, \ldots, x_n, true, false\}$ 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: $dom(\kappa \cdot \Gamma) = dom(\Gamma)$

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

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

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 7. A substitution S 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, $S\kappa_1 \geq S\kappa_2$ 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, $S\kappa_1 \geq S\kappa_2$. $\kappa_1 \geq_\Theta \kappa_2$ 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 $\Theta = \{j \geq k+1\}$.

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:

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

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:

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

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** $x[] = (x![] \mid x![])$ **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 explain 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 briefly discussed in Subsection 4.3, 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. The relation $\Theta_1 \models \Theta_2$ used below is defined by: $\Theta_1 \models \Theta_2$ if and only if $\forall(\kappa_1 \geq \kappa_2) \in \Theta_2, \kappa_1 \geq_{\Theta_1} \kappa_2$.

Definition 8. (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) $S\Gamma; S\Theta \vdash SP'$ for any substitution S, 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

A type reconstruction algorithm is obtained by (1) combining each rule with the rule (T-Weak) (so that the resulting typing rules are syntax-directed) and (2) reading the new typing rules in a bottom-up manner.

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 type reconstruction algorithm obtained in this manner is sound and complete: it takes a process expression P as an input and returns the principal typing of P if P is typable (otherwise it fails). For space restriction, we omit the whole description of the algorithm. Please refer to [4] for details. As already mentioned, a programmer need not put type annotations into a process expression: before passing a process expression to the algorithm, a system can automatically put fresh type and use variables $\alpha^{(j,k)}$ into a place where type annotations are omitted.

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 the type reconstruction algorithm 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 9 (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: $\kappa_i^{(0)} = 0, \kappa_i^{(m+1)} = [\kappa_1^{(m)}/j_1, \ldots, \kappa_n^{(m)}/j_n]\kappa_i$. Then, for some $M \geq 0$, $j_1 = \kappa_1^{(M)}, \ldots, j_n = \kappa_n^{(M)}$ is the least solution of Θ.*

Remark: If use variables occur in the type environment output by the algorithm, 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.

We informally explain that our analysis can be performed in time polynomial in the size n of a process expression. First, type reconstruction can be performed in time polynomial in n (the same argument applies as the one for simply-typed λ-calculus [5]). Secondly, the size of the use constraint set output by the type reconstruction algorithm is $O(n^2)$. Since each use variable can range only over $\{0, 1, \omega\}$, the number of iteration steps for solving the constraint is also $O(n^2)$, which implies that the total time required for solving the use constraint set is polynomial in the size of a process expression. Thus, the whole analysis can be done in polynomial time. More details are found in the full paper [4].

4.4 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):

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

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

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

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 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:

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

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 order for the type system to be sound 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. The soundness is proved in almost the same way as that of the original linear type system [8]. We only sketch the key ideas here: a full proof is found in the technical report [4].

The soundness of the type system is ensured by the following two properties:

(1) Lack of immediate misuse of channels.
It is easy to know by typing rules that a well-typed process doesn't immediately violate the channel usage specified by types; for example, if $x : [\tilde{\tau}]^{(0,1)}; \emptyset \vdash P$, then P cannot be of the form $(\nu y_1) \cdots (\nu y_n)\,(x![\tilde{v}] \mid x![\tilde{v}'])$.

(2) Subject reduction.
One-step reduction of a process is expressed by a 4-place relation: $\Gamma \vdash P \longrightarrow P' \dashv \Gamma'$. It is basically the same as the standard reduction relation of the form $P \rightarrow P'$ [11]; but in order to remember which channels have already been used, the consumed capability is removed from the type environment Γ or type annotations in P. For example, if the initial type environment is $(\Gamma, y : [\tau]^{(1,1)})$, the reduction of the process expression $(\nu x : [\tau]^{(1,1)})(y![u] \mid y?[z].x![z] \mid x?[w].v![w])$ by communication on the channel y is expressed by the relation:

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

The next reduction step, communication on the channel x, is expressed by:

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

We can show that if $\Gamma \vdash P$ and $\Gamma \vdash P \longrightarrow P' \dashv \Gamma'$, then $\Gamma' \vdash P'$.

By the above two properties, capabilities that have already been consumed cannot be reused; thus, we know that the type system is sound with respect to operational semantics.

6 Experimental Results

In this section, we show results of simple experiments with HACL compiler[4] to evaluate performance improvement obtained by our analysis. Application programs are Fibonacci functions `sfib25` and `pfib25` (which performs recursive calls sequentially/in parallel), a simulation of Conway's life game `life`, and concurrent objects `counter10000` (which creates a counter object and increments its value 10000 times) and `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 ones of optimized programs. 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[5]: therefore, `pfib25` is slower than `sfib25` because of overheads.

The result of `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 `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.

[4] 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.

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

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

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 Related Work

Our analysis has its origin in the linear channel type system developed by Kobayashi, Pierce, and Turner [8]. 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 [13] proposed another technique that can be used for finding some linear channels based on their effect-based analysis [12]. 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 method would

give rise to the same problem because an infinite number of channels (uniquely identified by control paths in the concrete semantics) would be mapped to the same abstract control path by the abstraction function.

Kobayashi, Nakade and Yonezawa [7] proposed a technique for finding linear channels (and *linearized* channels [8]). 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 [10] also proposed an algorithm for analyzing communication; however, its target language is very restricted (channels are not first-class values, and moreover, dynamic process creation is not allowed).

8 Conclusions

We have 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 Univ. Press, 1992.
2. Christopher Colby. Analyzing the communication topology of concurrent programs. In *ACM PEPM'95*, 1995.
3. Atsushi Igarashi. Type-based analysis of usage of values for concurrent programming languages. Master's thesis, University of Tokyo, February 1997.

4. Atsushi Igarashi and Naoki Kobayashi. Type-based analysis of communication for concurrent programming languages. Technical Report 97-03, Department of Information Science, University of Tokyo, June 1997.
5. P.C. Kanellakis, H.G. Mairson, and J.C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.
6. Naoki Kobayashi. A partially deadlock-free typed process calculus. In *IEEE LICS*, 1997.
7. Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *SAS'95*, volume 983 of *LNCS*, pages 225–242. Springer-Verlag, 1995.
8. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *ACM POPL'96*, pages 358–371, January 1996.
9. 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.
10. 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.
11. 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.
12. Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *ACM POPL'94*, pages 84–97, 1994.
13. 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.
14. Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *IEEE LICS'93*, pages 376–385, 1993.
15. 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.
16. John H. Reppy. CML: A higher-order concurrent language. In *ACM PLDI'91*, pages 293–305, 1991.
17. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM FPCA'95*, 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.

This article was processed using the LaTeX macro package with LLNCS style