TYPE-BASED ANALYSIS OF USAGE OF VALUES FOR CONCURRENT PROGRAMMING LANGUAGES

並列言語における型システムによる値の使用回数の解析

by

Atsushi Igarashi

五十嵐 淳

A Master's Thesis

Submitted to

the Graduate School of

the University of Tokyo

in Partial Fulfillment of the Requirements

for the Degree of Master of Science

in Information Science

Thesis Supervisor: Naoki Kobayashi 小林 直樹

Title: Lecturer of Information Science

February 5, 1997

ABSTRACT

We propose a type-based technique to analyze how many times each value, including communication channels, is used during execution of concurrent programs. This work is closely related with the recent work by Kobayashi, Pierce, and Turner on a linear channel system on a process calculus. They introduced a type system that ensures certain channels (called *linear channels*) to be used just once, and showed that how linear channels help reasoning about program behaviors. However, they only deal with a pure message passing calculus, and more importantly, the type reconstruction problem is left open. This thesis develops a type reconstruction algorithm of a variant of a linear channel type system for a concurrent language with data constructors such as records, and let-polymorphism. We can detect not only linear channels but also other used-once values (closures, records, etc.) by the type reconstruction algorithm. Computational cost of our analysis (excluding cost of ordinary type reconstruction – which is known exponential in the worst case –) is polynomial in the size of a program. Moreover, several experiments show that our analysis is enough efficient in practice. We also incorporate the proposed analysis into a compiler of a concurrent language *HACL* and present results of several benchmarks.

論 文 要 旨

並列プログラムの実行における、通信チャネルなどの値の使用回数を型に基づいて解析する手法を提案する。本研究は Kobayashi, Pierce, Turner らのリニアチャネルシステムに関する研究に密接に関連している。その研究では、あるチャネル (リニアチャネルという) は、型システムによって一度しか使われないことが保証でき、そのリニアチャネルはプログラムの振舞いに関する推論に役立つことが示されている。しかし、そこでは対象として純粋な並列計算しか扱われず、さらに重要なこととして、型推論の問題は論じられていない。本論文では、レコードなどのデータ構築子や let による多相型を導入した並列言語のためのリニアチャネルを判定する型システムおよびその型推論アルゴリズムを提案する。これはリニアチャネルだけでなく、それ以外の種類の一度しか使われない値 (関数、レコードなど) の検出を可能にする。この解析の計算コストは通常の型推論のコスト (最悪指数オーダーとして知られている) を除くと、プログラムサイズに対し多項式時間である。さらに、いくつかの実験により実際上は解析が十分効率的に行えることを示す。また、提案する解析手法を並列言語 *HACL* コンパイラ上に組み込みベンチマークの結果を示す。

Acknowledgement

Before everything else, I thank Naoki Kobayashi and Akinori Yonezawa, my current and former thesis supervisors. Naoki Kobayashi suggested me to study static analysis of concurrent programming languages, indicated the right direction of my work by many useful comments and discussions. Akinori Yonezawa also provided many insightful comments and encouragement for me.

Toshihiro Shimizu helped us experiment with his *HACL* compiler. I greatly appreciate his help. I also wish to express my gratitude to Kenjiro Taura for many useful comments from practical point of view and his encouragement.

I wish to thank David N. Turner. He carefully read an earlier draft of this thesis, pointed out a fault of the type system, and gave me many encouraging comments.

I especially thank Takayasu Ito for his kind treatment in my visit to Sendai and for carefully reading an earlier draft of this thesis and giving a number of useful suggestions.

I would like to express my gratitude to members of Yonezawa Laboratory and Kobayashi Laboratory. Discussions with Hidehiko Masuhara or Haruo Hosoya helped making understanding of my work clear. I wish to thank Toshihiro Shimizu, again, for serving delicious tea in our office. My special thanks are due to Yoshihiro Oyama and Toshiyuki Takahashi for struggling to make computer environment of Yonezawa Laboratory comfortable. I also wish to give my thanks to all members for their kindness and provision of joyful research life. I would like to thank Yoko

Kobayashi, secretary of Yonezawa Laboratory, for arrangement of many matters on my research.

Finally, I would like to express my gratitude to my classmates, including Motoki Nakade, Junji Tamatsukuri, Ken-ichiro Aoki, Satoshi Moriwaki and Haruo Hosoya for spending their precious time with me. They greatly diverted me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

With the recent development of parallel computers and high-speed networks, concurrent/distributed programming languages have been drawing great attentions. Concurrent programming languages are important even in sequential environments in describing interactive applications such as graphical user interfaces[4]. Indeed, many concurrent programming languages – including concurrent object-oriented languages[27, 1], concurrent extensions of functional languages[20], concurrent constraint/linear logic programming languages[22, 21, 10], and programming languages based on process calculi[17, 23] – have been proposed and implemented.

Of those researches, the frameworks of process calculi[13, 10] are drawing attentions as a vehicle for studying theoretical aspects of concurrent programming languages. The process calculi are based on the model that computation is performed by multiple concurrent processes communicating values along *communication channels*. Their computational models are simple but so expressive that various high-level constructs for concurrent object-oriented languages can be captured by process calculi extended with records[9, 11, 18]. Moreover, they provide us various useful methods to reason about behaviors of programs: criteria of behavioral equivalence of processes such as barbed bisimulation, or (static) type

systems. For these reasons, programming languages based on process calculi have the following advantages:

- Semantics of the core language is clearly defined through the basis calculus.
- High-level constructs, like concurrent objects, can be flexibly introduced/modified with clear semantics.
- Various aspects (theory, implementation, etc.) of such high-level constructs can be uniformly discussed.

On the other hand, once high-level constructs are encoded into a lower-level core language, their good properties are often thrown away and so safety use of the constructs is difficult to ensure on the core language. As another disadvantage, overhead of multi-threading and communication among threads, i.e., operations on channels, are large. To recover these disadvantages, analysis technique for languages based on process calculi, particularly *static* analysis which makes use of their clear semantics, is required. By developing analysis technique on such languages, it would be easy to develop similar technique for other concurrent programming languages.

1.2 Linear Channel Type System

As an instance of static analysis for process calculi, recently, Kobayashi, Pierce, and Turner[8] gave a static type system with which certain communication channels, called *linear channels*, can be ensured to be used just once. Linear channels have many good properties such as partial confluence with respect to communication on them.

Above all, linear channels enable to eliminate redundant communication. Consider the following expression **new** r **in** $(m(v, r) \mid r(x) \Rightarrow e)$ **end**, which represents a function call or a method invocation of concurrent objects. It creates a

new channel r, then activates $m(v,r)$ and $r(x) \Rightarrow e$ concurrently. $m(v,r)$ sends (v,r) to the channel m where m and v represent the location of the function or the method and its argument, respectively. The channel r is used for the callee to send a result. $r(x) \Rightarrow e$ is a message receiver which executes $[z/x]e$ when a value z is received at the channel r. If r is proved to be a linear channel, we can reduce the above expression into $m(v, \lambda x.e)$, in which the channel creation **new** r **in** ... **end** and the communication on the channel are eliminated. Moreover, consider a special case that e is just a $r'(x)$, which forwards the value from r to another channel r'. Then the resulting code is $m(v,r')$, which may be regarded as the *tail-call optimized form* of the expression **new** r **in** $(m(v,r) \mid r(x) \Rightarrow r'(x))$ **end**. Note that this kind of tail-call optimization is only possible by a non-trivial global analysis, unlike that in functional programming languages (because we need to check the behaviors of receivers on the channel m).

In addition to the elimination of redundant message passing, linear channel type system can be used to: (1) reduce the cost of operations on communication channels, (2) reduce the cost of garbage collection by the reuse of memory space for linear channels, (3) guarantee safety of programs by the properties of linear channels. There are many situations where linear channels are used. As mentioned above, one of two communications involved in function/method calls is performed on a linear channel. Moreover, most of dynamically created channels are linear ones in typical programs.

Their system is, however, on a pure message passing calculus, and the type reconstruction problem of the system is left open.

1.3 Our Approach

In this study, we improve their type system and develop a type reconstruction algorithm which can detect *affine channels*[8], which is used *at most once*. Note that in the type system of [8], linear channels and affine channels have nearly

identical properties because it is possible to throw away linear channels without using by placing them in a deadlocked subprocess. So, in this thesis, we do not particularly distinguish them.

The basic ideas of our analysis are a *type system with uses* and introduction of *use constraint*. *Uses* are attached to type constructors and denote how many times the values are used. For example, int^1 denotes the type of integers which can be used at most once where the superscripted 1 is a use. It will be ensured that values are used according to their uses, i.e., a value of int^1 is really used at most once, during execution.

Use constraint is introduced for ease of type reconstruction. In the sense that a whole program determines uses of values, a use is a global property. So, analysis in a simple-minded bottom-up manner does not work because uses of values cannot be determined from local information. To solve this problem, we introduce use variables, which represent such undetermined uses, and a use constraint set, which keeps local information as a system of inequalities on use variables. Our type reconstruction algorithm generates a use constraint set as a part of outputs. To detect linear channels, the constraint set is to be solved.

Our target language discussed here is a small core of concurrent programming languages, which can be considered as a process calculus extended with functions, data structures such as records, and the ML-style let-polymorphism. We choose such a language because behaviors of various concurrent programming languages can be captured as mentioned above and it is more realistic for practice than pure process calculi. Therefore, the result of this study would be easy to apply to other concurrent programming languages.

1.4 Contribution

Main contributions of this thesis are:

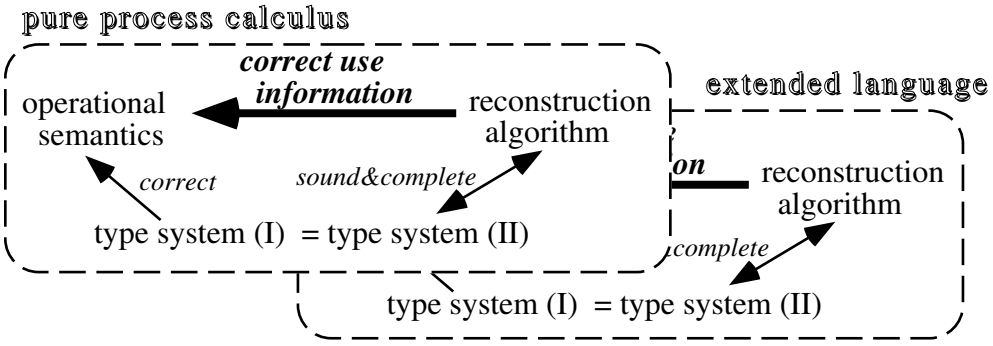

Figure 1.1: Overview of This Thesis

- Development of a type reconstruction algorithm for a type system with uses for a realistic core of concurrent programming languages.

- Development of a method for detection used-once values, including linear channels. Used-once values are detected with *no* declarations about types in programs by our analysis.

Computational cost of our analysis (excluding cost of ordinary type reconstruction – which is known exponential in the worst case –) is polynomial in the size of a program.

Note that the linearity analysis of communication channels is more difficult to detect than that in functional programming languages[24], because a linear channel has at least two syntactic occurrences – one for receiving and another for sending.

We also show that the cost of the analysis is efficient for practice and that the effects of optimization by linear channels are promising.

1.5 Overview of This Thesis

The goal of this thesis is to give an algorithm to obtain use information, and prove it to be correct with respect to operational semantics. The overview of this thesis is illustrated in Figure 1.1. First, we discuss the analysis for a pure process

calculus, and then we extend the target language. In both discussions, we introduce two kinds of type systems. One of them (type system (I)) is a type system to guarantee the correctness of uses of values. The judgements on uses obtained by the type system are proved to be sound with respect to operational semantics. Then, we modify it to the other type system, type system (II), which eases type reconstruction by introducing use constraint, present a type reconstruction algorithm and prove soundness and completeness of the algorithm. The two type systems are proved to be essentially equivalent. As a result, reconstructed type (including use information) will be sound with respect to operational semantics.

The structure of the rest of this thesis is as follows. In Chapter 2, we describe a type system with uses for a pure process calculus and prove it to be sound with respect to its operational semantics. The modified type system, its type reconstruction algorithm, and the method for detecting linear channels are presented in Chapter 3. In Chapter 4, we discuss our analysis for the extended language. By the analysis, we can detect not only linear channels but also other used-once values (closures, records, etc.) Chapter 5 gives some experimental results to show the effect of optimization through simple examples and that the cost of our analysis is enough efficient in practice in spite of its theoretical complexity. After discussing several issues and related work in Chapter 6 and Chapter 7, we conclude this thesis in Chapter 8.

Chapter 2

A Type System with Uses for a Pure Message Passing Calculus

In this chapter, we give a type system of an asynchronous (pure) message passing calculus, which judges how often each channel will be used, and prove that the type system is correct with respect to its operational semantics. The type system handles not only the usual types but also *uses*, which are attached to type constructors and denote how many times each channel is used. For example, $[\tau]^{(0,1)}$, where the superscripted 0 and 1 are called *uses*, denotes a type of channels used *at most once* to send a variable of τ type. By the correctness of the type system, we mean that the channels of such a type are used for sending at most once, indeed, during evaluation.

We first present the syntax of our target language, which can be considered an asynchronous fragment of the polyadic π-calculus[13], and explain its intuitive meaning. In the first section, without introducing uses, we give a type system and operational semantics for the target calculus. Second, the definition of uses and a type system with uses are described in Section 2.2. Third, operational semantics is extended with uses and the correctness of the type system with uses is proved in Section 2.3.

2.1 Target Language

The target language is considered as a fragment of the polyadic π-calculus allowing only asynchronous communication. Computation in this language is performed by multiple processes which communicate values asynchronously through communication channels, simply called *channels*. Channels are first-class citizens in the sense that they can be passed from a process to other processes.

2.1.1 Syntax

We give the definition of *preterms*, which are untyped process expressions. The metavariable $x(, y, z, \ldots)$ ranges over the finite set of *variables*. The syntax of preterms of the language is given below.

Definition 2.1 (preterms) The set of *preterms*, ranged over by e, is defined as follows.

$$
\begin{array}{rcll}
e & ::= & e_1 \,|\, e_2 & (\textit{parallel composition}) \\
 & | & x(y_1, \ldots, y_n) & (\textit{message send}) \\
 & | & r & (\textit{message receiver(s)}) \\
 & | & x^*(y_1, \ldots, y_n) \Rightarrow e & (\textit{replicated message receiver}) \\
 & | & \textbf{new } x \textbf{ in } e \textbf{ end} & (\textit{channel creation}) \\
r & ::= & x(y_1, \ldots, y_n) \Rightarrow e & (\textit{message receiver}) \\
 & | & r_1 \& r_2 & (\textit{guarded choice of receivers})
\end{array}
$$

A sequence of variables $y_1, \ldots, y_n$ is often written as $\tilde{y}$. Also, we abbreviate **new** x_1 **in** $\ldots$**new** x_n **in** e **end**$\ldots$ **end** to **new** $\tilde{x}$ **in** e **end**. We give $x(\tilde{y})$ (message send) a higher precedence than $\Rightarrow$, and $\Rightarrow$ a higher precedence than $|$ and $\&$. For example, $e_1 \,|\, x(z) \Rightarrow y(z) \,|\, e_2$ means $e_1 \,|\, (x(z) \Rightarrow (y(z))) \,|\, e_2$.

Their intuitive meanings are as follows. $e_1 \,|\, e_2$ executes e_1 and e_2 in parallel. $x(y_1, \ldots, y_n)$ sends the tuple of the values $(y_1, \ldots, y_n)$ along the channel x. **new** x **in** e **end** creates a new channel x and executes the process e. $x(y_1, \ldots, y_n) \Rightarrow e$ waits for $z_1, \ldots, z_n$ along the channel x, and executes $[z_1/y_1, \ldots, z_n/y_n]e$. $r_1 \& \cdots \& r_n$ executes either of r_i depending on available messages. For instance,

new x **in** $x(z) \mid ((v(y) \Rightarrow w(y))\&(x(y) \Rightarrow u(y)))$ **end** communicates on x and is reduced to **new** x **in** $u(z)$ **end**. A replicated message receiver $x^*(y_1, \ldots, y_n) \Rightarrow e$ also waits for $z_1, \ldots, z_n$ along x, spawns $[z_1/y_1, \ldots, z_n/y_n]e$ and executes itself repeatedly.

Bound variables of a preterm can be defined in a customary fashion, i.e., variables $y_1, \ldots, y_n$ are bound in e of $x(y_1, \ldots, y_n) \Rightarrow e$, $x^*(y_1, \ldots, y_n) \Rightarrow e$, and **new** $y_1, \ldots, y_n$ **in** e **end**. A variable which is not bound in a preterm will be called a free variable of the preterm. We define α-conversions of bound variables in a customary manner and assume that implicit α-conversions are allowed, henthforce.

2.1.2 Type System

The syntax of types is defined below:

Definition 2.2 (types) The metavariable $\alpha(, \beta, \ldots)$ ranges over the finite set of *type variables*. The set of *types*, ranged over by τ, is given by the following syntax.

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

$[\tau_1, \ldots, \tau_n]$, often abbreviated to $[\tilde{\tau}]$, denotes the type of channels which communicates a tuple of values, each of which has τ_i type.

A *type environment* Γ is a mapping from the set of variables to the set of types. The domain of a type environment is written as $dom(\Gamma)$. We write $x_1 : \tau_1, \ldots, x_n : \tau_n$, abbreviated to $\tilde{x} : \tilde{\tau}$, for a type environment Γ such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for each i. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ to mean a type environment Γ' such that $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ if $x \neq y$. A type judgement form is $\Gamma \vdash e$, read as *the preterm e is well-typed under* Γ. It means that if the free variables of e has the types obtained from Γ, respectively, then e is a well-typed process.

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 \mid e_2}\ (\text{T-Par}) \qquad \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 \& e_2}\ (\text{T-Choice})$$

$$\Gamma, x : [\tilde{\tau}], \tilde{y} : \tilde{\tau} \vdash x(\tilde{y})\ (\text{T-Send}) \qquad \frac{\Gamma, x : [\tilde{\tau}], \tilde{y} : \tilde{\tau} \vdash e}{\Gamma, x : [\tilde{\tau}] \vdash x(\tilde{y}) \Rightarrow e}\ (\text{T-Recv})$$

$$\frac{\Gamma, x : [\tilde{\tau}] \vdash e}{\Gamma \vdash \textbf{new}\ x\ \textbf{in}\ e\ \textbf{end}}\ (\text{T-New}) \qquad \frac{\Gamma, x : [\tilde{\tau}], \tilde{y} : \tilde{\tau} \vdash e}{\Gamma, x : [\tilde{\tau}] \vdash x^*(\tilde{y}) \Rightarrow e}\ (\text{T-RRecv})$$

Figure 2.1: Typing Rules for the Target Language

Typing rules are given in Figure 2.1. A well-typed preterm is simply called a *term*. To avoid confusion with type judgement derived the other typing rules described later, we write $\Gamma \vdash_{\mathcal{T}} e$ if it is derivable from these typing rules.

Intuitive meanings of typing rules are as follows:

(T-Par, T-Choice) If subexpression e_1 and e_2 is well-typed under Γ, then both $e_1 \mid e_2$ and $e_1 \& e_2$ are well-typed.

(T-Send) If x is a channel to a communicate a tuple of values, each of which has the type τ_i, and each y_i has the type τ_i, respectively, then message sending $x(\tilde{y})$ is well-typed.

(T-Recv, T-RRecv) If the body of a receiver e is well-typed under a type environment that each y_i has the type τ_i, respectively, and x has the channel type $[\tau_1, \ldots, \tau_n]$, then the whole receiver is well-typed under the type environment from which the assumption on $\tilde{y}$ is eliminated.

(T-New) If e is well-typed under a type environment that x has a channel type, then **new** x **in** e **end** is well-typed under the type environment from which the binding of x is eliminated.

2.1.3 Reduction Semantics

Following the standard presentation for process calculi[13], the reduction semantics for preterms is defined via two relations: a *structural congruence* and a *reduction relation*. First, we define a structural congruence relation $e_1 \cong e_2$, which is congruence based on the structures of preterms. It is introduced for the simplicity of the reduction semantics and represents the fact that the order of parallel composition in a preterm does not affect the behavior of the preterm and so forth. In other words, if two preterms are in a reduction relation given below, those structurally congruent to them are also in the relation. Second, the reduction relation $e \to e'$, which means that "e is reduced to e' in one step," is defined.

Definition 2.3 (structural congruence $e_1 \cong e_2$) *Structural congruence* $e_1 \cong e_2$ is the least congruence on preterms closed under the following rules.

$$
\begin{aligned}
e_1 \mid e_2 &\cong e_2 \mid e_1 \\
(e_1 \mid e_2) \mid e_3 &\cong e_1 \mid (e_2 \mid e_3) \\
e_1 \& e_2 &\cong e_2 \& e_1 \\
(e_1 \& e_2) \& e_3 &\cong e_1 \& (e_2 \& e_3) \\
\textbf{new } x \textbf{ in } (e_1 \mid e_2) \textbf{ end} &\cong e_1 \mid \textbf{new } x \textbf{ in } e_2 \textbf{ end} \text{ (if } x \text{ is not free in } e_1)
\end{aligned}
$$

Definition 2.4 (reduction relation $e \to e'$) The reduction relation $e \to e'$ is the least relation closed under the following rules:

$$((x(\tilde{y}) \Rightarrow e) \& e') \mid x(\tilde{z}) \to [\tilde{z}/\tilde{y}]e \text{ (R-Comm)}$$

$$(x^*(\tilde{y}) \Rightarrow e) \mid x(\tilde{z}) \to [\tilde{z}/\tilde{y}]e \mid (x^*(y) \Rightarrow e) \text{ (R-RComm)}$$

$$\frac{e_1 \cong e_2 \quad e_1 \to e_1' \quad e_1' \cong e_2'}{e_2 \to e_2'} \text{ (R-Cong)} \qquad \frac{e_1 \to e_1'}{e_1 \mid e_2 \to e_1' \mid e_2} \text{ (R-Par)}$$

$$\frac{e \to e'}{\textbf{new } x \textbf{ in } e \textbf{ end} \to \textbf{new } x \textbf{ in } e' \textbf{ end}} \text{ (R-New)}$$

The type system ensures that the well-typedness of a term is preserved during the reduction.

Theorem 2.1 (Subject Reduction (1)) If $\Gamma \vdash e$ and $e \to e'$, then $\Gamma \vdash e'$.

Proof Structural induction on the proof of $e \to e'$. $\square$

2.2 Type System with Uses

Now, we show how a type system judges how many times each channel is used. For this purpose, the set of *uses*, which are associated to the channel type constructor and denote how many times the channel is used, is introduced. The syntax of types and typing rules are redefined so that uses are properly handled.

2.2.1 Uses and Types with Uses

The set of uses, ranged over by κ, is $\{0, 1, \omega\}$. The intuitive meaning of each use is as follows. 0 means that channels will be never used, 1 means that channels will be used at most once, and ω means that channels will be used any number of times.

Note that a use is not just the number of syntax occurrences of the channel because each channel can be used for two different operations, i.e. sending and receiving. Consider the following preterm:

$$f(x) \Rightarrow x(v) \,|\, \textbf{new } y \textbf{ in } (f(y) \,|\, y(w) \Rightarrow e) \textbf{ end}$$

In the above preterm, y appears twice, but is used for different operations. It is used once for sending after y is sent to f, while for receiving in $y(w) \Rightarrow e$. Hence, a pair of two uses (κ_1, κ_2) is attached to the channel type constructor. The κ_1 and κ_2 denote how many times the channel *may* be used for receive operations and for send operations, respectively. After all, the total use of y is counted as $(1, 1)$ in the above example.

Now, we formally redefine the syntax of types.

Definition 2.5 (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.

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

As explained above, the attached κ_1, often called *receive use*, and κ_2, often called *send use*, denote how many times the channel *may* be used for receive operations and for send operations, respectively. For example, $[\tau]^{(0,1)}$ denotes a type of channels used *at most once* to send a value of τ type.

2.2.2 Typing Rules with Uses

A type judgement form $\Gamma \vdash e$ means not only that e is well typed in the sense until now, but also that each variable in e is used according to the uses of its type in Γ. For example, the type judgement $\Gamma, x : [\tau]^{(1,0)} \vdash e$ means that x is used *at most once for receiving a value of τ type, and never used for sending in e.*

Since type environments are concerned with uses of variables, we need to take special cares in merging type environments. For example, if $x : [\tau]^{(0,1)} \vdash e_1$ and $x : [\tau]^{(1,0)} \vdash e_2$, then x is totally used once for sending and once for receiving in $e_1 \mid e_2$. Therefore, the total use of a variable in $e_1 \mid e_2$ should be obtained by adding uses of two type environments. Thus, the rule for parallel composition is defined as:

$$\frac{\Gamma_1 \vdash e_1 \quad \Gamma_2 \vdash e_2}{\Gamma_1 + \Gamma_2 \vdash e_1 \mid e_2} \ (\textsc{Tu-Par})$$

where the operation $\Gamma_1 + \Gamma_2$, which will be defined later, represents a type environment obtained by adding uses of each variable in Γ_1 and Γ_2.

In the case of a guarded choice of receivers $(x_1(\tilde{y}_1) \Rightarrow e_1) \& \cdots \& (x_n(\tilde{y}_n) \Rightarrow e_n)$, on the other hand, the uses of a variable should be computed by taking an upper bound of uses in the type environments of the receivers, because only one of the

receivers is executed. Thus, the rule for & is defined as:

$$\frac{\Gamma_1 \vdash e_1 \quad \Gamma_2 \vdash e_2}{\Gamma_1 \sqcup \Gamma_2 \vdash e_1 \& e_2} \text{ (Tu-Choice)}$$

where $\Gamma_1 \sqcup \Gamma_2$ is the upper bound of type environments, which is also defined below.

Formal definitions of the summation and the upper bound of uses, types and type assignments are given below.

Definition 2.6 (summation) Summation of uses $\kappa_1 + \kappa_2$ is defined by:

$$\kappa_1 + \kappa_2 = \begin{cases} 0 & \text{if } (\kappa_1, \kappa_2) = (0,0) \\ 1 & \text{if } (\kappa_1, \kappa_2) \in \{(0,1),(1,0)\} \\ \omega & \text{otherwise} \end{cases}$$

If two types are identical except for their outermost uses, the summation of the types is obtained by adding the outermost uses:

$$\rho^{(\kappa_1,\kappa_2)} + \rho^{(\kappa_3,\kappa_4)} = \rho^{(\kappa_1+\kappa_3,\kappa_2+\kappa_4)}$$

The summation is extended pointwise to type environments as follows:

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

Definition 2.7 (upper bound) The upper bound of two uses $\kappa_1 \sqcup \kappa_2$ is defined as follows:

$$\kappa_1 \sqcup \kappa_2 = \begin{cases} 0 & \text{if } (\kappa_1, \kappa_2) = (0,0) \\ 1 & \text{if } (\kappa_1, \kappa_2) \in \{(0,1),(1,0),(1,1)\} \\ \omega & \text{otherwise} \end{cases}$$

The definition of the upper bound of types and that of type environments are obtained by replaced + with $\sqcup$ from Definition 2.6.

Another important operation is $\kappa \cdot \Gamma$, which represents the summation of κ copies of the type environment Γ. Using this operator, the typing rule for replicated message receivers is defined as follows:

$$\frac{\Gamma, y : \tilde{\tau} \vdash e}{(\omega \cdot \Gamma) + x : [\tilde{\tau}]^{(\omega,0)} \vdash x^*(y) \Rightarrow e} \text{ (Tu-RRecv)}$$

Because a replicated receiver $x^*(\tilde{y}) \Rightarrow e$ is considered as an arbitrary number of a parallel composition of $x(\tilde{y}) \Rightarrow e$, each binding in the type environment Γ used in the body may be used an arbitrary number of times, i.e., ω times. Therefore, we need totally the summation of ω copies as the type environment.

For example, in the following preterm,

$$(x^*(y) \Rightarrow z(y)) \,|\, x(v) \,|\, x(w)$$

the free variable z in the replicated receiver appears only once, but a message sending to x appears twice. So, the number of send operations on the channel z should be counted as $2 \cdot 1$.

$\kappa \cdot \Gamma$ is formally defined as follows.

Definition 2.8 (κ-product) $\kappa_1 \cdot \kappa_2$ is defined below.

$$\kappa_1 \cdot \kappa_2 = \begin{cases} 1 & \text{if } (\kappa_1, \kappa_2) = (1, 1) \\ \omega & \text{if } (\kappa_1, \kappa_2) \in \{(1, \omega), (\omega, 1), (\omega, \omega)\} \\ 0 & \text{otherwise} \end{cases}$$

$\kappa \cdot \tau$ is obtained by the multiplication of κ with the outermost use of τ, and $\kappa \cdot \Gamma$ is obtained by the pointwise extension:

$$\kappa \cdot \rho^{(\kappa_1, \kappa_2)} = \rho^{(\kappa \cdot \kappa_1, \kappa \cdot \kappa_2)}$$

$$(\kappa \cdot \Gamma)(x) = \kappa \cdot (\Gamma(x))$$

We allow to make the assumption on uses rough. For example, if $x : [\tau]^{(0,1)}, y : \tau \vdash x(y)$ holds, then we allow $x : [\tau]^{(0,\omega)}, y : \tau \vdash x(y)$, that is, if e is well typed under the assumption that x is used at most once, then e is well typed even under the assumption that x is used an arbitrary number of times. The below-defined relation $\Gamma_1 \geq \Gamma_2$ represents that the assumptions on uses of the variables in Γ_1 are larger (rougher) than that in Γ_2.

Definition 2.9 (relation $\geq$) The relation between uses $\geq$ is a partial order which satisfies $\omega \geq 1 \geq 0$. The relation $\geq$ is extended pointwise to pairs of uses.

$$\frac{\Gamma_1 \vdash e_1 \quad \Gamma_2 \vdash e_2}{\Gamma_1 + \Gamma_2 \vdash e_1 \mid e_2}\ (\textsc{Tu-Par}) \qquad \frac{\Gamma_1 \vdash e_1 \quad \Gamma_2 \vdash e_2}{\Gamma_1 \sqcup \Gamma_2 \vdash e_1 \& e_2}\ (\textsc{Tu-Choice})$$

$$\frac{\kappa_2 \geq 1 \quad \tau_1' \geq \tau_1 \quad \cdots \quad \tau_n' \geq \tau_n}{\Gamma, x : [\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)}, y_1 : \tau_1', \ldots, y_n : \tau_n' \vdash x(y_1, \ldots, y_n)}\ (\textsc{Tu-Send})$$

$$\frac{\Gamma, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash e \quad \kappa_1 \geq 1}{\Gamma + x : [\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)} \vdash x(y_1, \ldots, y_n) \Rightarrow e}\ (\textsc{Tu-Recv})$$

$$\frac{\Gamma', y_1 : \tau_1, \ldots, y_n : \tau_n \vdash e \quad \Gamma \geq \omega \cdot \Gamma'}{\Gamma + x : [\tau_1, \ldots, \tau_n]^{(\omega,\kappa)} \vdash x^*(y_1, \ldots, y_n) \Rightarrow e}\ (\textsc{Tu-RRecv})$$

$$\frac{\Gamma, x : [\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)} \vdash e}{\Gamma \vdash \mathbf{new}\ x^{(\kappa_1,\kappa_2)}\ \mathbf{in}\ e\ \mathbf{end}}\ (\textsc{Tu-New})$$

Figure 2.2: Typing Rules with Uses

We define $\geq$ between types as $\rho_1^{(\kappa_1,\kappa_2)} \geq \rho_2^{(\kappa_3,\kappa_4)}$ if and only if $\rho_1 = \rho_2$ and $(\kappa_1, \kappa_2) \geq (\kappa_3, \kappa_4)$. For type environments, we define $\Gamma \geq \Gamma'$ as follows:

$$\Gamma \geq \Gamma' \text{ iff } \forall x \in dom(\Gamma').(x \in dom(\Gamma) \wedge (\Gamma(x) \geq \Gamma'(x)))$$

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

The whole typing rules with uses are described in Figure 2.2. We write $\Gamma \vdash_{\mathcal{TU}} e : \tau$ if $\Gamma \vdash e : \tau$ is derivable by these rules. We explain other notable rules.

(Tu-Send) Because x in $x(\tilde{y})$ is used as a channel for sending a message, send use κ_2 of x must be greater than 0.

(Tu-New) The syntax for channel creation **new** x **in** e **end** is modified to **new** $x^{(\kappa_1,\kappa_2)}$ **in** e **end**[1] so that the use of x is declared explicitly. κ_1 and κ_2 are receive and send use of x, respectively. These indices of uses in preterms will be used for program transformation or for efficient code generation for channels, which are mentioned in Chapter 1. Note that there is no need for

[1] The abbreviation **new** $x_1^{(\kappa_{11},\kappa_{12})}, \ldots, x_n^{(\kappa_{n1},\kappa_{n2})}$ **in** e **end** is often used instead of **new** $x_1^{(\kappa_{11},\kappa_{12})}$ **in** ... **new** $x_n^{(\kappa_{n1},\kappa_{n2})}$ **in** e **end** ... **end**

programmers to declare them, indeed. The type reconstruction algorithm described in Chapter 3 infers and recovers these uses.

The types system with uses is sound with respect to the type system described in the previous section. By the soundness, we mean that if a preterm is well-typed under some type environment Γ by the typing rules in Figure 2.2, it is also well-typed by the typing rules in Figure 2.1 under the type environment from which use information is removed. $Erase(\Gamma, e)$ is a pair of Γ' and e' such that uses are removed from e and the bound types in Γ. The removal of uses from a type $EraseTy(\tau)$ is defined as $EraseTy([\tau_1, \ldots, \tau_n]^{(\kappa_1,\kappa_2)}) = [EraseTy(\tau_1), \ldots, EraseTy(\tau_2)]$ and $EraseTy(\alpha^{(\kappa_1,\kappa_2)}) = \alpha$.

Theorem 2.2 If $\Gamma \vdash_{\mathcal{TU}} e$, then there exist some Γ' and e' such that $\Gamma' \vdash_{\mathcal{T}} e'$, and $(\Gamma', e') = Erase(\Gamma, e)$.

Proof Structural induction on the proof of $\Gamma \vdash_{\mathcal{TU}} e$. □

2.3 Correctness of Type System with Uses

In this section, we show that the type system correctly judges the uses of channels. For example, we must ensure that if $x : [\tilde{\tau}]^{(0,1)} \vdash e$, then x can be used at most once for sending during evaluation of e. In order to check the correctness, the reduction relation is annotated with some use information. Then, we show that the correctness of the type system with uses with respect to this reduction semantics.

2.3.1 Reduction Semantics with Uses

In the original definition, the reduction relation is written as $e \rightarrow e'$. In this section, this relation is annotated with an environment Γ and a label l and written as $\Gamma \vdash e \xrightarrow{l} e'$. Intuitively, Γ is a type environment which keeps uses of variables, which can possibly be used during the evaluation of e, and l is either a variable x,

which implies that e is reduced to e' by communication on the free channel x, or a special label ε, which implies communication on a bound channel.

Definition 2.10 (reduction relation $\Gamma \vdash e \xrightarrow{l} e'$) The reduction relation $\Gamma \vdash e \xrightarrow{l} e'$ is the least relation closed under the following rules:

$$\frac{\kappa_1 \geq 1 \quad \kappa_2 \geq 1}{\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash ((x(\tilde{y}) \Rightarrow e)\&e') \,|\, x(\tilde{z}) \xrightarrow{x} [\tilde{z}/\tilde{y}]e} \text{ (Ru-Comm)}$$

$$\frac{\kappa \geq 1}{\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa)} \vdash (x^*(\tilde{y}) \Rightarrow e) \,|\, x(\tilde{z}) \xrightarrow{x} [\tilde{z}/\tilde{y}]e \,|\, (x^*(\tilde{y}) \Rightarrow e)} \text{ (Ru-RComm)}$$

$$\frac{e_1 \cong e_2 \quad \Gamma \vdash e_1 \xrightarrow{l} e_1' \quad e_1' \cong e_2'}{\Gamma \vdash e_2 \xrightarrow{l} e_2'} \text{ (Ru-Cong)} \qquad \frac{\Gamma_1 \vdash e_1 \xrightarrow{l} e_1'}{\Gamma_1 + \Gamma_2 \vdash e_1 \,|\, e_2 \xrightarrow{l} e_1' \,|\, e_2} \text{ (Ru-Par)}$$

$$\frac{\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash e \xrightarrow{x} e'}{\Gamma \vdash \mathbf{new}\ x^{(\kappa_1,\kappa_2)}\ \mathbf{in}\ e\ \mathbf{end} \xrightarrow{\varepsilon} \mathbf{new}\ x^{(\kappa_1^-,\kappa_2^-)}\ \mathbf{in}\ e'\ \mathbf{end}} \text{ (Ru-New1)}$$

$$\frac{\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash e \xrightarrow{l} e' \quad x \neq l}{\Gamma \vdash \mathbf{new}\ x^{(\kappa_1,\kappa_2)}\ \mathbf{in}\ e\ \mathbf{end} \xrightarrow{l} \mathbf{new}\ x^{(\kappa_1,\kappa_2)}\ \mathbf{in}\ e'\ \mathbf{end}} \text{ (Ru-New2)}$$

where $1^- = 0$ and $\omega^- = \omega$

We define Γ^{-l} by $(\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})^{-x} = \Gamma, x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)}$ and $\Gamma^{-\varepsilon} = \Gamma$. Intuitively, Γ^{-x} represents a type environment after a reduction occurs on the channel x. When $\Gamma \vdash e \xrightarrow{l} e'$ holds, we often write $\Gamma \vdash e \xrightarrow{l} e' \dashv \Gamma^{-l}$. It means that if the reduction comes from communication on a free channel of x, the uses (κ_1, κ_2) of the binding of x in Γ are decreased by 1. On the other hand, if the reduction comes from communication on a bound channel, then the uses are removed from the corresponding **new**-construct in the preterm. For example, the preterm

$$\mathbf{new}\ x^{(1,1)}\ \mathbf{in}\ (y(u) \,|\, y(z) \Rightarrow (x(z) \,|\, x(w) \Rightarrow v(w)))\ \mathbf{end}$$

is reduced under the type environment $\Gamma, y : [\tau]^{(1,1)}$ by communication on the free channel y:

$$\Gamma, y : [\tau]^{(1,1)} \vdash \mathbf{new}\ x^{(1,1)}\ \mathbf{in}\ (y(u) \,|\, y(z) \Rightarrow (x(z) \,|\, x(w) \Rightarrow v(w)))\ \mathbf{end}$$
$$\xrightarrow{y} \mathbf{new}\ x^{(1,1)}\ \mathbf{in}\ (x(u) \,|\, x(w) \Rightarrow v(w))\ \mathbf{end} \dashv \Gamma, y : [\tau]^{(0,0)}$$

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

$$\Gamma, y : [\tau]^{(0,0)} \vdash \mathbf{new}\ x^{(1,1)}\ \mathbf{in}\ (x(u) \mid x(w) \Rightarrow v(w))\ \mathbf{end}$$
$$\xrightarrow{\varepsilon} \mathbf{new}\ x^{(0,0)}\ \mathbf{in}\ v(u)\ \mathbf{end} \dashv \Gamma, y : [\tau]^{(0,0)}$$

On terms, the operational semantics with uses allows the same reduction as that of the previous section by removing use information.

Theorem 2.3 Suppose $\Gamma \vdash_{\mathcal{TU}} e_1$. Then, $\Gamma \vdash e_1 \xrightarrow{l} e_2$ holds if and only if $e_1' \to e_2'$ holds where e_1' and e_2' are obtained by removing uses from e_1 and e_2, respectively, and $\to$ is the reduction relation defined in Definition 2.4.

Proof

$\Rightarrow$ Structural induction on the proof of $\Gamma \vdash e \xrightarrow{l} e'$.

$\Leftarrow$ Structural induction on the proof of $e_1' \to e_2'$ with the well-typedness of e_1.

□

2.3.2 Correctness about Uses of Channels

As usual, correctness of the type system is shown through the subject reduction theorem (Theorem 2.4), which implies that well-typedness of a term is preserved during reduction, together with the lack of immediate misuse of uses by any term (Theorem 2.5). By the lack of immediate misuse of uses, we mean, for example, that there is no case where $y : [\tau]^{(0,1)} \vdash e$ but $e \cong \mathbf{new}\ x_1^{(\kappa_{11},\kappa_{12})}, \ldots, x_n^{(\kappa_{n1},\kappa_{n2})}\ \mathbf{in}\ (y(z_1) \mid y(z_2) \mid e')\ \mathbf{end}$ (which means that e is now trying to send *two* messages along the channel y though y is declared as a channel allowing only one sending).

Subject reduction theorem is stated again as follows.

Theorem 2.4 (Subject Reduction (2)) If $\Gamma \vdash e$ and $\Gamma \vdash e \xrightarrow{l} e'$, then Γ^{-l} is well-defined and $\Gamma^{-l} \vdash e'$ holds.

Proof Structural induction on the proof of $\Gamma \vdash e \xrightarrow{l} e'$. See Appendix A.1 for the detailed proof. □

Note that it implies not only that well-typedness is preserved in the usual sense, but also that the uses of the corresponding channel are removed after communication, so that, for example, the use $(1,1)$ of a channel becomes $(0,0)$ after it is used for communication.

Theorem 2.5 (Run-time safety) If $\Gamma \vdash e$ and $e \cong$ **new** $x_1^{(\kappa_{11},\kappa_{12})}, \ldots, x_n^{(\kappa_{n1},\kappa_{n2})}$ **in** $(e_1\{\mid e_2\})$ **end** then[2]:

1. If e_1 is $x(y_1, \ldots, y_n) \mid (x(z_1, \ldots, z_m) \Rightarrow e')\&r$ or $x(y_1, \ldots, y_n) \mid x^*(z_1, \ldots, z_m) \Rightarrow e'$, then $n = m$ and for the use pair (κ_1, κ_2) of the binding of x (in either Γ or **new**), $(\kappa_1, \kappa_2) \geq (1,1)$.

2. If e_1 is $x(\tilde{y})$, then the send use of the binding of x is greater than 0.

3. If e_1 is $(x(\tilde{y}) \Rightarrow e')\&r$, then the receive use of the binding of x is greater than 0.

4. If e_1 is $x(\tilde{y}) \mid x(\tilde{z})$, then the send use of the binding of x is ω.

5. If e_1 is $((x(\tilde{y}) \Rightarrow e')\&r_1) \mid ((x(\tilde{z}) \Rightarrow e'')\&r_2)$ then the receive use of the binding of x is ω.

6. If e_1 is $x^*(\tilde{y}) \Rightarrow e'$, then the receive use of the binding of x is ω.

Note that r denotes an arbitrary number of receivers $x(\tilde{y}) \Rightarrow e$ combined with $\&$.

Proof Trivial from the typing rules. □

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

2.4 Summary

First, we have described a type system for a pure asynchronous process calculus and its operational semantics. Then, a type system with uses, which are attached to the channel type constructor and denote how many times each channel is used, have been introduced. The type system with uses is sound with the original type system if use information is removed. The type system with uses has been proved to be sound with respect to operational semantics (with uses). The soundness ensures that there is no possibility of misuse of channels against its uses, i.e., channels, which are proved to be linear channels by the type system, are used at most once during reduction, indeed.

Chapter 3

Type Reconstruction and Detection of Linear Channels

The purpose of this chapter is to reconstruct, given a preterm e, the most general type environment Γ such that $\Gamma \vdash_{\mathcal{TU}} e$. The typing rules presented in Section 2.2 are not enough for this purpose. For example, consider the preterm $x(z) \,|\, y(z)$. In order to find the most general typing, the rule (Tu-Par) tells us to first compute the most general typings for $x(z)$ and $y(z)$, and then add the obtained type environments. However, since the use of z cannot be determined at this moment, the reconstruction step stops there. In order to avoid this, we introduce *use variables* to represent undetermined uses, and keep information on the use variables as *a constraint set on use variables*. Thus, the most general typing is represented as a pair consisting of a type environment and a constraint set on use variables. For example, the most general typings for $x(z)$ and $y(z)$ can be represented in the forms: $((x : \cdots, z : \alpha^{(j_1,j_2)}), \Theta_1)$ and $((y : \cdots, z : \alpha^{(k_1,k_2)}), \Theta_2)$ where j_1, j_2, k_1, k_2 are use variables and Θ_1, Θ_2 are constraint sets on use variables. From these typings, the typing for $x(z) \,|\, y(z)$ is represented as a pair $((x : \cdots, y : \cdots, z : \alpha^{(l_1,l_2)}), \Theta_1 \cup \Theta_2 \cup \{l_1 \geq j_1 + k_1, l_2 \geq j_2 + k_2\})$. This reconstruction step is expressed by the rule:

$$\frac{\Gamma_1, \Theta \vdash e_1 \quad \Gamma_2, \Theta \vdash e_2 \quad \Gamma \geq_\Theta \Gamma_1 + \Gamma_2}{\Gamma, \Theta \vdash e_1 \,|\, e_2} \text{ (Tru-Par)}$$

where the intended meaning of $\Gamma, \Theta \vdash e$ is that $S\Gamma \vdash_{\mathcal{TU}} Se$ holds for any substitution S of uses for use variables if S satisfies a system of inequations Θ. With these modifications, we can obtain, given a preterm e, the most general pair (Γ, Θ) such that $\Gamma, \Theta \vdash e$. After that, we can obtain $\Gamma' \vdash_{\mathcal{TU}} e'$ by solving the set Θ of inequations and substituting the minimal solution for use variables in Γ and e.

Note that by the introduction of use variables, programmers need not declare uses of channels in **new**-construct, because fresh use variables can automatically be inserted into a program before the reconstruction algorithm is applied.

First, we show a modified type system for reconstruction. After that, we demonstrate detection of linear channels. The procedure to detect linear channels consists of: (1) a type reconstruction phase to obtain the principal typing of a given preterm and (2) a phase to solve the inequalities on use variables. In the rest of this chapter, after presenting the modified type system for reconstruction and a type reconstruction algorithm (Section 3.1–3.3), we describe how to solve the inequalities on use variables (Section 3.4). We then give an example of detection of linear channels (Section 3.5).

3.1 Typing Rules with Use Constraint Set

The metavariable $j(, k, \ldots)$ ranges over the finite set of *use variables*. We replace uses with the following *use expressions* in order to handle use variables, summations and products of uses.

Definition 3.1 (use expressions) The set of *use expressions*, ranged over by κ, is given by the following syntax:

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

We often call $0, 1$, and ω *use constants*. We do not distinguish between a use expression with no use variable and its corresponding use. For example, we identify the use expression $1 + 0$ with the use 1. A *use constraint set* Θ is a system of

inequalities on use expressions, where each inequality is of the form $\kappa_1 \geq \kappa_2$ where κ_1 is restricted to be either a use constant or a use variable. The syntax of types is not changed except that uses are replaced with use expressions. We allow only either use constants or variables in **new** $x^{(\kappa_1,\kappa_2)}$ **in** e **end**.

The relation $\geq_\Theta$ is used in the typing rules for reconstruction, instead of $\geq$. $\kappa_1 \geq_\Theta \kappa_2$ means that for any ground substitutions for use variables (i.e., substitutions that instantiate all use variables in κ_1, κ_2, and Θ with use constants), if all inequalities in Θ are satisfied, then $\kappa_1 \geq \kappa_2$ is satisfied. For example, $j \geq_\Theta l+1$ if $\Theta = \{j \geq k+1, k \geq l\}$. The relation is extended pointwise to pairs of uses. Also, the relation $\geq_\Theta$ is extended to types and type environments in the same manner as in Section 2.2.

Definition 3.2 (relation $\geq_\Theta$) $\kappa_1 \geq_\Theta \kappa_2$ is defined if for all ground substitutions of use constants for use variables S, $\forall(\kappa_1' \geq \kappa_2' \in \Theta).S\kappa_1' \geq S\kappa_2' \Rightarrow S\kappa_1 \geq S\kappa_2$.

$\tau_1 \geq_\Theta \tau_2$ and $\Gamma_1 \geq_\Theta \Gamma_2$ are defined as follows:

$$\rho_1^{(\kappa_1,\kappa_2)} \geq_\Theta \rho_2^{(\kappa_3,\kappa_4)} \text{ iff } \rho_1 = \rho_2 \text{ and } (\kappa_1,\kappa_2) \geq_\Theta (\kappa_3,\kappa_4)$$

$$\Gamma \geq_\Theta \Gamma' \text{ iff } \forall x \in dom(\Gamma').(x \in dom(\Gamma) \wedge \Gamma(x) \geq_\Theta \Gamma'(x))$$

The typing rules for reconstruction are presented in Figure 3.1. We write $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$ if $\Gamma, \Theta \vdash e$ is derivable by the rules in Figure 3.1. 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:

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

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

The relation $\Theta_1 \models \Theta_2$ intuitively means that Θ_1 is stronger constraint than Θ_2.

$$\frac{\Gamma_1, \Theta \vdash e_1 \quad \Gamma_2, \Theta \vdash e_2 \quad \Gamma \geq_\Theta \Gamma_1 + \Gamma_2}{\Gamma, \Theta \vdash e_1 \,|\, e_2} \text{ (TRU-PAR)} \qquad \frac{\Gamma_1, \Theta \vdash e_1 \quad \Gamma_2, \Theta \vdash e_2 \quad \Gamma \geq_\Theta \Gamma_1 \sqcup \Gamma_2}{\Gamma, \Theta \vdash e_1 \& e_2} \text{ (TRU-CHOICE)}$$

$$\frac{\kappa_2 \geq_\Theta 1 \quad \tilde{\tau}' \geq_\Theta \tilde{\tau}}{\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}, \tilde{y} : \tilde{\tau}', \Theta \vdash x(\tilde{y})} \text{ (TRU-SEND)} \qquad \frac{\Gamma', \tilde{y} : \tilde{\tau}, \Theta \vdash e \quad \kappa_1 \geq_\Theta 1 \quad \Gamma \geq_\Theta \Gamma' + x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}}{\Gamma, \Theta \vdash x(\tilde{y}) \Rightarrow e} \text{ (TRU-RECV)}$$

$$\frac{\Gamma', x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}, \Theta \vdash e \quad \Gamma \geq_\Theta \Gamma'}{\Gamma, \Theta \vdash \textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end}} \text{ (TRU-NEW)} \qquad \frac{\Gamma', \tilde{y} : \tilde{\tau}, \Theta \vdash e \quad \Gamma \geq_\Theta (\omega \cdot \Gamma') + x : [\tilde{\tau}]^{(\omega,\kappa)}}{\Gamma, \Theta \vdash x^*(\tilde{y}) \Rightarrow e} \text{ (TRU-RRECV)}$$

Figure 3.1: Typing Rules for Type Reconstruction

Definition 3.3 (relation $\Theta_1 \models \Theta_2$)

$$\Theta_1 \models \Theta_2 \text{ iff } \forall \kappa_1 \geq \kappa_2 \in \Theta_2, \kappa_1 \geq_{\Theta_1} \kappa_2$$

The following two lemmas are trivial from the definition of $\geq_\Theta$.

Lemma 3.1 If $\tau \geq_\Theta \tau'$ and $\Theta' \models \Theta$, then $\tau \geq_{\Theta'} \tau'$.

Lemma 3.2 If $\Gamma \geq_\Theta \Gamma'$ and $\Theta' \models \Theta$, then $\Gamma \geq_{\Theta'} \Gamma'$.

Lemma 3.3 If $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$ and $\Theta' \models \Theta$, then $\Gamma, \Theta' \vdash_{\mathcal{TRU}} e$.

Proof Structural induction on the proof of $\Gamma, \Theta' \vdash_{\mathcal{TRU}} e$. □

The type system for reconstruction is essentially equivalent to the type system presented in Section 2.2 in the following sense.

Theorem 3.1 (Equivalence of $\mathcal{TU}$ and $\mathcal{TRU}$) If $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$, then $S_\kappa \Gamma \vdash_{\mathcal{TU}} S_\kappa e$ for any ground substitutions S_κ of use constants for use variables such that $\emptyset \models S_\kappa \Theta$ where $\emptyset$ is the empty constraint set (i.e., $S_\kappa \Theta$ is satisfied). Conversely, if $\Gamma \vdash_{\mathcal{TU}} e$, then $\Gamma, \emptyset \vdash_{\mathcal{TRU}} e$ where $\emptyset$ is the empty constraint set.

Proof The first half of the statement is proved as follows. For any substitutions S'_κ for use variables, $S'_\kappa \Gamma, S'_\kappa \Theta \vdash_{\mathcal{TRU}} S'_\kappa e$ holds by straightforward induction on the structure of the proof of $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$. By $\emptyset \models S_\kappa \Theta$ and Lemma 3.3, $S_\kappa \Gamma, \emptyset \vdash_{\mathcal{TRU}}$

$S_\kappa e$ holds. Since S_κ is a ground substitution, $S_\kappa\Gamma \vdash_{\mathcal{TU}} S_\kappa e$ (by straightforward induction).

The second half of the statement is proved by straightforward induction on the structure of the proof of $\Gamma \vdash_{\mathcal{TU}} e : \tau$. □

We give the definition of the principal typing after several preliminary definitions.

3.2 Principal Typing

First, we define a *type-use substitution* S, often simply called a *substitution* unless confusing, below. S maps not only a type variable to a bare type but also a use variable to a use constant.

Definition 3.4 (type-use substitution) A *type-use substitution* S is defined as a pair of S_ρ and S_κ, where S_ρ is a mapping from type variables to bare types and S_κ is a mapping from use variables to either use constants or use variables. $(S_\rho, S_\kappa)(\tau)$ is defined as $S_\kappa S_\rho \tau$. $(S_\rho, S_\kappa)(\kappa)$ is defined as $S_\kappa \kappa$.

Lemma 3.4 (Type-Use Substitution) If $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$, then $S\Gamma, S\Theta \vdash_{\mathcal{TRU}} Se$ for any type-use substitutions S.

Proof Straightforward induction on the structure of the proof of $\Gamma, \Theta \vdash_{\mathcal{TRU}} e : \tau$. □

Principal typing is defined as a pair consisting of a type environment and a use constraint set.

Definition 3.5 (principal typing) (Γ, Θ) is defined as the *principal typing* of e if the pair satisfies the following conditions:

1. $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$

2. If $\Gamma', \Theta' \vdash_{\mathcal{TU}} e$, there exists a substitution S such that

   (a) $\Theta' \models S\Theta$

   (b) $\Gamma' \supseteq S\Gamma$

3.3 Type Reconstruction Algorithm *PTU*

We show a type reconstruction algorithm for the modified type system. Before describing the algorithm, we introduce several subprocedures including unification.

Formally, the property of the unification algorithm U is stated as follows:

Definition 3.6 (unifier) A substitution S is a *unifier* of a set of pairs of types $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$ if and only if $S\tau_{i1} \equiv S\tau_{i2}$ for all i.

Definition 3.7 (most general unifier) A unifier S of $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$ is the *most general unifier* if and only if there exists S'' such that $S' = S''S$ for any other unifier S' of $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$

We do not present an algorithm of unification, but it is easy to obtain from the standard one by restricting use expressions in types to be either constants or variables.

Theorem 3.2 (Unification (1)) Given $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$ where every use in τ_{ij} is either a constant or a variable, there exists an algorithm U which computes the most general unifier of the set $\{(\tau_{11}, \tau_{12}), \ldots, (\tau_{n1}, \tau_{n2})\}$, or reports failure if it does not exist.

Then, we introduce subprocedures $\bigoplus$, $\bigodot$, and $\bigsqcup$, each of which corresponds to the conditions in the typing rules $\Gamma \geq_\Theta \Gamma_1 + \Gamma_2$, $\Gamma \geq_\Theta \kappa \cdot \Gamma'$ and $\Gamma \geq_\Theta \Gamma_1 \sqcup \Gamma_2$, respectively. From pairs of a type environment and a use constraint set, they generate a type environment and a use constraint set which satisfy the condition on the type environments like above.

We give the definitions of them by using the unification algorithm U.

Definition 3.8 $\bigoplus(\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\})$ is defined as the following procedure:

1. Define E, Θ, and Γ as the least sets (or mapping) which satisfy the following conditions. For each $x \in dom(\Gamma_1) \cup \cdots \cup dom(\Gamma_n)$,

$$\begin{cases} E \supseteq \{(\rho_{i_{x_1}}^{(j_{x1}, j_{x2})}, \rho_{i_{x_2}}^{(j_{x1}, j_{x2})}), \ldots, (\rho_{i_{x_{m-1}}}^{(j_{x1}, j_{x2})}, \rho_{i_{x_m}}^{(j_{x1}, j_{x2})})\} \\ \Theta \supseteq \{j_{x1} \geq \kappa_{i_{x_1}1} + \ldots + \kappa_{i_{x_m}1}, j_{x2} \geq \kappa_{i_{x_1}2} + \ldots + \kappa_{i_{x_m}2}\} \\ \Gamma(x) = \rho_{i_{x1}}^{(j_{x1}, j_{x2})} \end{cases}$$

   where $\{i_{x_1}, \ldots, i_{x_m}\} (= I_x)$ be the set of indices s.t. $i \in I_x \Leftrightarrow x \in dom(\Gamma_i)$, $\rho_{i_{x_k}}^{(\kappa_{i_{x_k}1}, \kappa_{i_{x_k}2})} = \Gamma_{i_{x_k}}(x)$ $(i_{x_k} \in I_x)$, and j_{x1}, j_{x2} are fresh use variables.

2. $S = U(E)$

3. Return $(S, S\Gamma, S(\Theta \cup \Theta_1 \cup \cdots \cup \Theta_n))$

Definition 3.9 $\bigodot(\Gamma, \Theta, \kappa)$ is defined as the following procedure:

1. Return a type environment Γ' and the least use constraint set Θ' which satisfy the following conditions: 1) $dom(\Gamma') = dom(\Gamma)$, 2) $\Theta' \supseteq \Theta$ and 3) $\Gamma(x) = \rho^{(\kappa_1, \kappa_2)} \Rightarrow (\Gamma'(x) = \rho^{(j_1, j_2)} \wedge j_1 \geq \kappa \cdot \kappa_1, j_2 \geq \kappa \cdot \kappa_2 \in \Theta')$ where j_1, j_2 are fresh use variables.

The definition of $\bigsqcup(\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\})$ is obtained from the definition of $\bigoplus$ by replacing all $+$ with $\sqcup$.

The algorithm PTU, shown in Figure 3.2 and Figure 3.3, takes e as an input and returns a triple (Γ, Θ, e). For the simplicity of the following discussions, we assume that the unification procedure does not generate a unifier which substitutes any other uses for use variables in e. This assumption is reasonable because it can be proved that uses to be unified with uses in e are only use variables. Hence, it is possible to avoid substituting some uses for use variables in e.

The following theorem ensures that our algorithm computes a principal typing.

Theorem 3.3 (PTU) If $\Gamma', \Theta' \vdash_{\mathcal{TRU}} e$, then $PTU(e)$ computes the principal typing of e without failure.

$$
\begin{array}{l}
PTU(x(y_1,\ldots,y_n)) = (\{x : [\alpha_1^{(j_{11},j_{12})},\ldots,\alpha_n^{(j_{n1},j_{n1})}]^{(j,k)}, y_1 : \alpha_1^{(k_{11},k_{12})},\ldots,y_n : \alpha_n^{(k_{n1},k_{n2})}\}, \\
\qquad\qquad \{k \geq 1\} \cup \{k_{ij} \geq j_{ij} | 1 \leq i \leq n, j = 1,2\}, x(y_1,\ldots,y_n)) \\
\qquad\qquad \text{where } \alpha_i\text{'s, } j,k,j_{ij}\text{'s, and } k_{ij}\text{'s are fresh type/use variables} \\
PTU(e_1 \,|\, e_2) = \text{let } (\Gamma_1,\Theta_1,e_1) = PTU(e_1) \\
\qquad\qquad (\Gamma_2,\Theta_2,e_2) = PTU(e_2) \\
\qquad\qquad (S,\Gamma,\Theta) = \bigoplus(\{(\Gamma_1,\Theta_1),(\Gamma_2,\Theta_2)\}) \\
\qquad \text{in } (\Gamma,\Theta,e_1 \,|\, e_2) \\
PTU(e_1 \& e_2) = \text{let } (\Gamma_1,\Theta_1,e_1) = PTU(e_1) \\
\qquad\qquad (\Gamma_2,\Theta_2,e_2) = PTU(e_2) \\
\qquad\qquad (S,\Gamma,\Theta) = \bigsqcup(\{(\Gamma_1,\Theta_1),(\Gamma_2,\Theta_2)\}) \\
\qquad \text{in } (\Gamma,\Theta,e_1 \& e_2) \\
PTU(x(y_1,\ldots,y_n) \Rightarrow e) = \\
\qquad \text{let } (\Gamma_1,\Theta_1,e) = PTU(e) \\
\qquad\qquad (S_1,\Gamma,\Theta) = \bigoplus(\{\ (x : [\alpha_1^{(j_1,k_1)},\ldots,\alpha_n^{(j_n,k_n)}]^{(j,k)}, \{j \geq 1\}), \\
\qquad\qquad\qquad\qquad (\Gamma_1 \setminus (y_i : \tau_i | y_i \in dom(\Gamma_1)), \Theta_1)\}) \\
\qquad\qquad S_2 = U(\{(S_1\alpha_i^{(j_i,k_i)}, S_1\tau_i) | y_i \in dom(\Gamma_1)\}) \\
\qquad \text{in } (S_2\Gamma, S_2\Theta, x(y_1,\ldots,y_n) \Rightarrow e) \\
\qquad\quad \text{where } \alpha_i\text{'s, } j_i\text{'s, and } k_i\text{'s are fresh type/use variables.}
\end{array}
$$

Figure 3.2: Type Reconstruction Algorithm PTU (part 1)

$PTU(x^*(y_1,\ldots,y_n) \Rightarrow e) =$

let $(\Gamma_1, \Theta_1, e) = PTU(e)$

$(\Gamma_1', \Theta_1') = \bigodot(\Gamma_1 \setminus (y_i : \tau_i | y_i \in dom(\Gamma_1)), \Theta_1, \omega)$

$(S_1, \Gamma, \Theta) = \bigoplus(\{(x : [\alpha_1^{(j_1,k_1)}, \ldots, \alpha_n^{(j_n,k_n)}]^{(\omega,k)}, \emptyset), (\Gamma_1', \Theta_1')\})$

$S_2 = U(\{(S_1\alpha_i^{(j_i,k_i)}, S_1\tau_i) | y_i \in dom(\Gamma_1)\})$

in $(S_2\Gamma, S_2\Theta, x^*(y_1,\ldots,y_n) \Rightarrow e)$

where α_i's, j_i's, and k_i's are fresh type/use variables.

$PTU(\textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end}) =$

let $(\Gamma, \Theta, e) = PTU(e)$

in if $x \in dom(\Gamma)$ then

$(S(\Gamma \setminus x : \tau_x), S\Theta, \textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end})$

where $S = U(\{(\tau_x, \alpha^{(\kappa_1,\kappa_2)})\})$

α is a fresh type variable

else $(\Gamma, \Theta, \textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end})$

Figure 3.3: Type Reconstruction Algorithm PTU (part 2)

Proof Structural induction on the proof of $\Gamma', \Theta' \vdash_{\mathcal{TRU}} e$. See Appendix A.2 for the detailed proof. □

3.4 Detection of Linear Channels by Solving Constraint

Given a pair (Γ, Θ) such that $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$, we obtain Γ such that $\Gamma' \vdash_{\mathcal{TU}} e'$ by solving Θ[1]. 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, (2) solving the first part of inequalities and (3) checking whether the solution satisfies the second part[2].

Since every operation on uses is monotonic, and $j_1, \ldots, j_m$ can range over finite space, the minimal solution of the first part is calculated by the simple iteration[3].

[1] e' is the preterm where use variables in e is replaced with the solution.

[2] If the check fails, Θ has no solutions.

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

Note that $j_1 = \omega, \ldots, j_n = \omega$ is always a solution of the first part.

Theorem 3.4 (Minimal Solution) Let $\Theta = \{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$. These inequalities are abbreviated as $\vec{j} \geq \vec{\kappa}(\vec{j})$. Note that each κ_i can be regarded as a n-ary function. Define the tuple of use constants $\vec{\kappa}^{(m)}(m \geq 0)$ by

$$\vec{\kappa}^{(0)} = (\overbrace{0, \ldots, 0}^{n})$$
$$\vec{\kappa}^{(m+1)} = \vec{\kappa}(\vec{\kappa}^{(m)}) = ([\kappa_1^{(m)}/j_1, \ldots, \kappa_n^{(m)}/j_n]\kappa_1, \ldots, [\kappa_1^{(m)}/j_1, \ldots, \kappa_n^{(m)}/j_n]\kappa_n).$$

Then, for some M, $\vec{j} = \vec{\kappa}^{(M)}$ is the least solution of Θ.

Proof Since a use constant can range over the finite space and the operations for uses are monotonic, there exists M such that $\vec{\kappa}^{(M)} = \vec{\kappa}(\vec{\kappa}^{(M)})$ holds. Suppose that $\vec{\kappa}' \geq \vec{\kappa}(\vec{\kappa}')$ holds for some $\vec{\kappa}'$. Then, by the monotonicity of each κ_i,

$$\vec{\kappa}' \geq \vec{\kappa}(\vec{\kappa}') \geq \ldots \geq (\vec{\kappa})^M(\vec{\kappa}')$$

Moreover, by $\vec{\kappa}' \geq (0, \ldots, 0)$,

$$(\vec{\kappa})^M(\vec{\kappa}') \geq (\vec{\kappa})^M((0, \ldots, 0))(= \vec{\kappa}^{(M)})$$

from which we obtain $\vec{\kappa}' \geq \vec{\kappa}^{(M)}$. $\square$

Remark: If use variables appear in Γ of the pair (Γ, Θ) obtained by the reconstruction, then those variables should be kept undetermined, because they may be constrained outside the term. When such a term is required to be compiled at that time (for separate compilation, etc.), we can allow programmers to declare their uses (in addition to type declarations), and the compiler can assign ω without such declarations.

3.5 An Example of Analysis and Optimization

We show an example of type reconstruction and optimization. Consider the following term, which computes the factorial of n, implemented with the target language extended with integers, boolean values and several primitive operators like **if**.

$fact^*(n, c) =$

if $n < 2$ **then** $c(1)$

else new $c1^{(j,k)}, c2$ **in**

$(pred(n, c1) \mid c_1(x) \Rightarrow (mul(x, n, c2) \mid c2(y) \Rightarrow c(y)))$ **end**

$pred$ is a (built-in) replicated receiver, which sends $n - 1$ to c if it receives (n, c). mul is also a built-in replicated receiver, which sends $x \times y$ to c if it receives (x, y, c). $pred$ and mul have the type $[int, [int]^{(0,1)}]^{(0,\omega)}$ and $[int, int, [int]^{(0,1)}]^{(0,\omega)}$, respectively[4]. For simplicity, we concern only uses related to $c1$. So uses about $c2$ is omitted.

The reconstruction algorithm outputs the triple $(fact : [int, [int]^{(j_1,k_1)}]^{(\omega,l)}, \Theta, e))$ where

$$\Theta = \{j \geq j_1 + j_2, k \geq k_1 + k_2, k_1 \geq 1, j_2 \geq 1, \ldots\}$$

Use variables j_1 and k_1 in Θ are the uses of $c1$ in the type environment for $pred(n, c1)$. Similarly, j_2 and k_2 are the ones for $c1(x) \Rightarrow e'$. The minimal solution of Θ is

$$j = j_2 = 1, j_1 = 0,$$
$$k = k_1 = 1, k_2 = 0, \ldots$$

As a result, we know that $c1$ is a linear channel. Similarly, we know that $c2$ is also linear.

From the information obtained above, we can optimize the program below.

$factopt^*(n, c) =$

if $n < 2$ then $c(1)$

else **new** $c1^{(1,1)}$ **in** $pred(n, c1) \mid c1(x) \Rightarrow mul(x, n, c)$ **end**

Note that the optimized program is "tail-call optimized" form of the original program.

[4]Strictly speaking, uses should be attached to also int, but we ignore them.

3.6 Summary

The type system with uses introduced in the previous chapter has been modified for type reconstruction. The modified system is essentially equivalent to the original one, but become suitable for extracting a type reconstruction algorithm by introducing use expressions and use constraint sets. Then, a type reconstruction algorithm *PTU* for the modified type system has been described. Given a preterm e, *PTU* outputs its principal typing, which includes a use constraint set. By solving the use constraint, with no type declarations in a preterm, we can detect which channels are linear. By linear channel information, we can apply a program transformation corresponding to tail-call optimization of concurrent programming languages.

Chapter 4

Analysis of Usage of Values

In this chapter, we extend our analysis to the language extended with basic values such as integers, functions, data constructors such as records, and ML-style let-polymorphism. The extended language is still small but realistic enough to apply our analysis to other concurrent programming languages. We show a typing system with uses and its type reconstruction algorithm for the extended language. Although it augments the expressiveness, the language extensions do not require our type system and analysis method to be modified so much.

To prove the correctness of the type system, we need handle uses of values other than channels in operational semantics. For this purpose, the reduction relation of the operational semantics is annotated with *heap*, which is an abstraction of memory space. To put it more concretely, a snapshot of execution is represented as a pair of a preterm and a heap, which keeps information on values of free variables of the preterm, and the reduction relation is defined as a relation between such pairs.

The structure of this chapter is as follows. In Section 4.1, the pure message passing calculus we have treated is extended. The extended syntax, a type system with uses, and its type reconstruction algorithm are described. In Section 4.2, the correctness of the type system is proved.

4.1 Extended Language

The extended target language is close to the core of HACL[10] and can be considered as an asynchronous process calculus equipped with integers, functions, the ML-style let-polymorphism, and records. It is small, but realistic enough to apply our analysis to other concurrent programming languages. Similarly to the language treated until the previous section, computation is performed by multiple processes which communicate values asynchronously through channels.

4.1.1 Syntax

The syntax of the extended language is given below.

Definition 4.1 (preterms) The set of *preterms*, ranged over by e, and the set of *values*, ranged over by v, are defined as follows.

$$
\begin{array}{rcll}
e & ::= & x(, y, z, \ldots) & (\textit{variable}) \\
 & | & v^{\kappa} & (\textit{values (other than channels) with a use}) \\
 & | & e_1 +_{\kappa} e_2 & (\textit{summation of integers}) \\
 & | & e_1\ e_2 & (\textit{application, message send}) \\
 & | & \textbf{match}\ \{l_1 = x_1, \ldots, l_n = x_n\} = e_1\ \textbf{in}\ e_2\ \textbf{end} & \\
 & & & (\textit{simultaneous field extractions from a record}) \\
 & | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} & (\textit{polymorphic definition}) \\
 & | & e_1 | e_2 & (\textit{parallel composition}) \\
 & | & x_1(y_1) \Rightarrow e_1 \& \cdots \& x_n(y_n) \Rightarrow e_n & (\textit{message receiver(s)}) \\
 & | & \textbf{new}\ x^{(\kappa_1, \kappa_2)}\ \textbf{in}\ e\ \textbf{end} & (\textit{channel creation}) \\
v & ::= & i \in \{\ldots, -1, 0, 1, 2, \ldots\} & (\textit{integers}) \\
 & | & \lambda x.e & (\lambda\textit{-abstraction}) \\
 & | & \textbf{rec}\ f(x) = e & (\textit{recursive function or process}) \\
 & | & \{l_1 = e_1, \ldots, l_n = e_n\} & (\textit{record creation})
\end{array}
$$

κ of v, $+$, and channel creation is a use (constant) defined before. Uses in a preterm are often omitted for readability unless they are important. We give $e_1\ e_2$ (applications, or message send) a higher precedence than $+_{\kappa}$, $+_{\kappa}$ a higher precedence than $\Rightarrow$, and $\Rightarrow$ a higher precedence than $|$ and $\&$.

Their intuitive meanings are as follows. $e_1 +_\kappa e_2$ evaluates e_1 and e_2 to two integers i_1 and i_2 and computes their mathematical summation whose use is κ. For example, $3^1 +_\omega 2^1$ is evaluated to 5^ω. $e_1\ e_2$ evaluates e_1 and e_2 to values x and y. If x is a communication channel, then it sends y along the channel x. (while if x is a function, it applies x to y as in ML.) **rec** $f(x) = e$ is a recursive function (or process) where f can be referred as $\lambda x.e$ in e. **match** first evaluates e_1 to a record $\{l_1 = y_1, \ldots, l_n = y_n\}^\kappa$, then binds $x_1, \ldots, x_n$ to $y_1, \ldots, y_n$, and executes e_2. For example, **match** $\{l = x, m = y\} = \{l = 3, m = \lambda x.x\}^\omega$ **in** $y\ x$ **end** is reduced to 3 via $\lambda x.x\ 3$. The intuitive meaning of **let** $x = e_1$ **in** e_2 **end** is the same as in ML. The concurrency primitives have the same meaning as described in Chapter 2. The communication primitives ($e_1\ e_2$ and $x(y) \Rightarrow e$) look as if they can pass only a single value at once. However, with records, polyadic communication can be expressed. Besides, replicated message receivers are removed from the syntax because they can be represented with **rec**. For example, $x^*(y) \Rightarrow e$ is represented as **rec** $x(y) = e$, while sending to a replicated message receiver is as an application of such a recursive process.

A variable x is bound in e of: 1) $\lambda x.e$, 2) **let** $x = e'$ **in** e **end**, 3) **new** $x^{(\kappa_1,\kappa_2)}$ **in** e **end**, and 4) $y(x) \Rightarrow e$. Similarly, f and x are bound in e of **rec** $f(x) = e$, and the variables $x_1, \cdots, x_n$ are bound in e_2 of **match** $\{l_1 = x_1, \ldots, l_n = x_n\} = e_1$ **in** e_2 **end**. We allow implicit α-conversion of bound variables, again.

4.1.2 Typing Rules

Uses are attached not only to the channel type constructor but also type constructors of integers, functions, and records. Similar to communication channels, the use of other values is also not just the number of syntactic occurrences. For the integer type, we count its use when the values are actually used, that is, by applying $+_\kappa$. For example, the use of 3 in a preterm $(\lambda x.(((\lambda y.1)x) +_\kappa x))\ 3$ (where uses are omitted) is 1, because x in the first occurrence of x is never used. Similarly,

the use of a record is counted only when values are extracted by **match** and that of a function is counted in application.

Now, we formally define the types and the type schemes for polymorphic values. For the convenience of type reconstruction, we attach two uses to every type constructor, although one of them is not important for most type constructors.

Definition 4.2 (bare types, types with uses, type schemes) The set of *bare types*, ranged over by ρ, and the set of *types*, ranged over by τ, the set of *type schemes*, ranged over by σ, are given by the following syntax.

$$
\begin{array}{rcll}
\rho & ::= & \mathit{int} & (\textit{integer type}) \\
 & | & \dot{\alpha}(,\dot{\beta},\ldots) & (\textit{(ordinary) type variables}) \\
 & | & \bar{\alpha}(,\bar{\beta},\ldots) & (\textit{non-}O\textit{ type variables}) \\
 & | & O & (\textit{process type}) \\
 & | & \tau_1 \to \tau_2 & (\textit{function types}) \\
 & | & \{l_1 : \tau_1, \ldots, l_n : \tau_n\} & (\textit{record types}) \\
\tau & ::= & \rho^{(\kappa_1,\kappa_2)} & \\
\sigma & ::= & \tau \mid \forall\dot{\alpha}.\sigma \mid \forall\bar{\alpha}.\sigma & (\textit{type schemes})
\end{array}
$$

O is the special type for processes, that is, if a preterm e is proved to be a well-typed process expression, it has O type. We introduce two kinds of type variables. One of them, $\dot{\alpha}$, is the usual one, for which any bare type can be substituted. $\bar{\alpha}$ is a type variable, which can be replaced with any bare type except for the process type and ordinary type variables. When we don't need to distinguish the two kinds of type variables, we use the word 'type variables' and α. The (bare) type of channels that carry values of a type τ is represented as $\tau \to O$ instead of $[\tau]$. Thus, a function that returns a process, called *process closure*, and a channel have the same form of a bare type.[1] They are distinguished by the use; since functions can only be used for application, the receive use is required to be 0. It is forced by the typing rule for λ-abstraction by giving a function a type $(\tau_1 \to \tau_2)^{(0,\kappa)}$. We often write $\tau_1 \stackrel{(\kappa_1,\kappa_2)}{\longrightarrow} \tau_2$ for $(\tau_1 \to \tau_2)^{(\kappa_1,\kappa_2)}$.

[1]This is not surprising because a channel m for sending a message can be considered as a process that sends a message $m(v)$ and does nothing.

We allow silent α-conversions of bound type variables by $\forall\alpha$. Two bare type schemes $\forall\alpha.\sigma$ and σ are identified if α is not bound in σ. Also, σ_1 and σ_2 are identified if they differ only in the order of their $\forall\alpha$'s. For instance, we identify $\forall\alpha_1.\forall\alpha_2.\forall\alpha_3.\alpha_1^{(j,k)} \to \alpha_2^{(l,m)}$ and $\forall\beta.\forall\alpha_1.\alpha_1^{(j,k)} \to \beta^{(l,m)}$.

Since they are not important, we ignore one of the uses of type constructors except for the function (channel) type constructor $\to$. Therefore, we often write int^κ for $int^{(-,\kappa)}$, O for $O^{(-,-)}$, and $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}^\kappa$ for $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}^{(-,\kappa)}$. The uses replaced with "$-$" have no real meaning, hence they are assumed to be ω by default in the type system.

A type environment will be a mapping from variable to type schemes in the type system of this chapter. Type judgement form is modified to $\Gamma \vdash e : \tau$, which means that "e has the type τ under the type environment Γ."

The definitions of $\tau_1 + \tau_2$, $\tau_1 \sqcup \tau_2$, $\kappa \cdot \tau$ and $\tau_1 \geq \tau_2$ are similar to that in Section 2.2. Also the definitions of such operations on type schemes are obtained as straightforward extension of those of types, i.e.,

$$\forall\alpha_1 \ldots \forall\alpha_n.\rho_1^{(\kappa_1,\kappa_2)} + \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_3,\kappa_4)} = \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1+\kappa_3,\kappa_2+\kappa_4)}$$

$$\forall\alpha_1 \ldots \forall\alpha_n.\rho_1^{(\kappa_1,\kappa_2)} \sqcup \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_3,\kappa_4)} = \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1\sqcup\kappa_3,\kappa_2\sqcup\kappa_4)}$$

$$\kappa \cdot \forall\alpha_1 \ldots \alpha_n.\rho^{(\kappa_1,\kappa_2)} = \forall\alpha_1 \ldots \alpha_n.\rho^{(\kappa\cdot\kappa_1,\kappa\cdot\kappa_2)}$$

$\forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1,\kappa_2)} \geq \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_3,\kappa_4)}$ is defined if $(\kappa_1, \kappa_2) \geq (\kappa_3, \kappa_4)$.

Before describing the typing rules, we give several preliminary definitions. For a subtle technical reason, we assume that e_2 in $e_1\ e_2$, e_1 in **let** $x = e_1$ **in** e_2 **end**, and each e_i in $\{l_1 = e_1, \ldots, l_n = e_n\}$ cannot have the type O. The condition is represented as $\tau \not\succeq O$ which means that τ is neither O nor $\dot{\alpha}^{(\kappa_1,\kappa_2)}$.

Substitutions for type variables are restricted so that O type (or ordinary type variables $\dot{\alpha}$) is not substituted for non-O type variables.

Definition 4.3 (well-defined substitution) A substitution for type variables

S_ρ is *well-defined* if and only if for all non-O type variables $\bar{\alpha}$, $S_\rho\bar{\alpha} \not\equiv O$ holds. A type-use substitution (S_ρ, S_κ) is well-defined if and only if S_ρ is well-defined.

We assume all substitutions are well-defined in this thesis.

The whole typing rules are described in Figure 4.1. $clos(\Gamma, \tau)$ denotes the type scheme $\forall\alpha_1 \ldots \forall\alpha_n.\tau$ where the α_i's are type variables that appear free in τ but not in Γ. We explain several notable rules.

(ETu-Plus) This rule requires both of two integers e_1 and e_2 to have uses greater than 0.

(ETu-Abs) Since the type $\tau_1 \xrightarrow{(0,\kappa)} \tau_2$ means that the function $\lambda x.e$ can be invoked κ times, each binding in the type environment Γ used in the body may be used κ times. Therefore, we need totally the summation of κ copies as the type environment at least.

(ETu-Recon) A record of type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}^\kappa$ allows us to use at most κ copies of the value of the type τ_i stored in the field l_i. Therefore, it must be that each element can be used as a value of type $\kappa \cdot \tau_i$.

4.1.3 Type Reconstruction Algorithm for the Extended Language

By introducing use expressions and use constraint sets, the typing rules presented in the previous subsection are easily modified for reconstruction in the same manner as in Chapter 3. We do not present the typing rules for reconstruction because they are trivial. The each name of the typing rule modified from (ETu-...) is (ETru-...), respectively. In this chapter, henthforce, $\Gamma \vdash_{\mathcal{ETU}} e : \tau$ means that $\Gamma \vdash e : \tau$ is derivable from the typing rules (ETu-...) and $\Gamma, \Theta \vdash_{\mathcal{ETRU}} e : \tau$ means $\Gamma, \Theta \vdash e : \tau$ is derivable from (ETru-...).

Equivalence of the two typing rules is stated in the similar manner to Theorem 3.1.

$$\frac{S_\rho \tau \geq \tau' \text{ for some substitution } S_\rho = [\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]}{\Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau \vdash x : \tau'} \text{ (ETu-Var)}$$

$$\Gamma \vdash i^\kappa : int^\kappa \text{ (ETu-Int)}$$

$$\frac{\Gamma_1 \vdash e_1 : int^{\kappa_1} \quad \Gamma_2 \vdash e_2 : int^{\kappa_2} \quad \kappa_1 \geq 1 \quad \kappa_2 \geq 1}{\Gamma_1 + \Gamma_2 \vdash e_1 +_\kappa e_2 : int^\kappa} \text{ (ETu-Plus)}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau_2 \quad \kappa_2 \geq 1 \quad \tau_1 \not\leq O \quad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau_2} \text{ (ETu-App/Send)}$$

$$\frac{\Gamma\{x : \tau_1\} \vdash e : \tau_2 \quad \Gamma' \geq \kappa \cdot \Gamma}{\Gamma' \vdash \lambda x.e : \tau_1 \xrightarrow{(0,\kappa)} \tau_2} \text{ (ETu-Abs)}$$

$$\frac{\Gamma\{f : \tau_1 \xrightarrow{(0,\omega)} \tau_2, x : \tau_1\} \vdash e : \tau_2 \quad \Gamma' \geq \omega \cdot \Gamma}{\Gamma' \vdash \textbf{rec } f(x) = e : \tau_1 \xrightarrow{(0,\omega)} \tau_2} \text{ (ETu-Fix)}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1' \quad \tau_1' \geq \kappa \cdot \tau_1 \quad \ldots \quad \Gamma_n \vdash e_n : \tau_n' \quad \tau_n' \geq \kappa \cdot \tau_n}{\Gamma_1 + \ldots + \Gamma_n \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}^\kappa} \text{ (ETu-Recon)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e' : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}^\kappa \qquad \kappa \geq_\Theta 1 \\ \Gamma_2\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash e : \tau \quad \tau_1 \not\leq O \ldots \tau_n \not\leq O\end{array}}{\Gamma_1 + \Gamma_2 \vdash \textbf{match } \{l_1 = x_1, \ldots, l_n = x_n\} = e' \textbf{ in } e \textbf{ end} : \tau} \text{ (ETu-Match)}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau' \quad \tau' \not\leq O \quad \Gamma_2\{x : clos(\Gamma_1, \tau')\} \vdash e_2 : \tau}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau} \text{ (ETu-Let)}$$

$$\frac{\Gamma_1 \vdash e_1 : O \quad \Gamma_2 \vdash e_2 : O}{\Gamma_1 + \Gamma_2 \vdash e_1 \mid e_2 : O} \text{ (ETu-Par)}$$

$$\frac{\Gamma_1 \vdash e_1 : O \quad \Gamma_2 \vdash e_2 : O}{\Gamma_1 \sqcup \Gamma_2 \vdash e_1 \ \& \ e_2 : O} \text{ (ETu-Choice)}$$

$$\frac{\kappa_1 \geq 1 \quad \Gamma_2\{y : \tau\} \vdash e : O}{\{x : \tau \xrightarrow{(\kappa_1,\kappa_2)} O\} + \Gamma_2 \vdash x(y) \Rightarrow e : O} \text{ (ETu-Recv)}$$

$$\frac{\Gamma\{x : (\tau \xrightarrow{(\kappa_1,\kappa_2)} O)\} \vdash e : O}{\Gamma \vdash \textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end} : O} \text{ (ETu-New)}$$

Figure 4.1: Typing Rules with Uses for the Extended Language

Theorem 4.1 (Equivalence of $\mathcal{ETU}$ and $\mathcal{ETRU}$) If $\Gamma, \Theta \vdash_{\mathcal{ETRU}} e : \tau$, then $S_\kappa\Gamma \vdash_{\mathcal{ETU}} S_\kappa e : S_\kappa\tau$ for any ground substitutions S_κ for use variables such that $\emptyset \models S_\kappa\Theta$ where $\emptyset$ is the empty constraint set (i.e., $S_\kappa\Theta$ is satisfied). Conversely, If $\Gamma \vdash_{\mathcal{ETU}} e : \tau$, then $\Gamma, \emptyset \vdash_{\mathcal{ETRU}} e : \tau$ where $\emptyset$ is the empty constraint set.

Proof Similar to the proof of Theorem 3.1. □

Principal typing of the type system for the extended language is defined as a triple consisting of a type environment, a use constraint set and a type.

Definition 4.4 (principal typing) (Γ, Θ, τ) is the *principal typing* of e if the triple satisfies the following conditions:

1. The range of Γ is only types (not including type schemes).
2. $\Gamma, \Theta \vdash_{\mathcal{ETRU}} e : \tau$
3. If $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} e : \tau'$ where the range of Γ' is only types, there exists a substitution S such that $\Theta' \models S\Theta$, $\Gamma' \supseteq S\Gamma$, and $\tau' \equiv S\tau$.

Before describing type reconstruction for the modified typing rules, we must mention unification. In our system, we deal with type schemes in type environments, so unification between type schemes is required. Besides ordinary unification , we must check whether bound type variables corresponds each other. Two type schemes $\forall\alpha_1 \ldots \forall\alpha_n.\tau_1$ and $\forall\beta_1 \ldots \forall\beta_m.\tau_2$ are unified as follows:

1. Let $\tau_1' = [\gamma_1/\alpha_1, \ldots, \gamma_n/\alpha_n]\tau_1$ where each γ_i is a fresh type variable, and $\tau_2' = [\delta_1/\beta_1, \ldots, \delta_m/\beta_m]\tau_2$ where each δ_i is a fresh *eigen* type variable, i.e., a type variable for which no other types can be substituted.
2. Apply first-order unification to the pair (τ_1', τ_2').
3. Let S be the most general unifier of the pair. Check whether $\{\gamma_i \mapsto S(\gamma_i) \mid \gamma_i \in dom(S)\}$ is a 1-to-1 mapping to $\{\delta_1, \ldots, \delta_m\}$.

Therefore, the existence of the unification algorithm for the set of pairs of type schemes can be stated in a similar manner to Theorem 3.2.

Theorem 4.2 (Unification (2)) Given $\{(\sigma_{11}, \sigma_{12}), \ldots, (\sigma_{n1}, \sigma_{n2})\}$ where every use in σ_{ij} is either a constant or a variable, there exists an algorithm U which computes the most general unifier of the set $\{(\sigma_{11}, \sigma_{12}), \ldots, (\sigma_{n1}, \sigma_{n2})\}$, or reports failure if it does not exist.

Then, the subprocedures $\bigoplus$, $\bigodot$ and $\bigsqcup$ defined in the previous section, are extended straightforwardly.

The type reconstruction algorithm, called $PTUV$, is shown in Figure 4.2, Figure 4.3, Figure 4.4, and Figure 4.5. $PTUV$ takes not only e but also a mapping from variables to type schemes As. The additional argument As keeps type schemes of variables declared by **let**. Since the algorithm may give incorrect results if both **let**-bound and λ-bound, we assume that the input to $PTUV$ is a preterm with all bound variables renamed to be distinct.

The soundness theorem of $PTUV$ is stated below.

Theorem 4.3 (PTUV) If $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} e : \tau'$ where the range of Γ' is only types, then $PTU(e, \emptyset)$ computes the principal typing of e where $\emptyset$ is the empty mapping.

Proof See Appendix A.3. □

4.1.4 Further Optimization by Analysis of Uses of Values

With the language extension, further optimization by linear channels is possible. Consider the following example[2] of Fibonacci function[3].

rec $fib(n, c) =$

[2] Several primitives like **if** are added to the language

[3] This function computes the two recursive calls sequentially.

$PTUV(x, As) =$ if $As(x) = \forall\alpha_1 \ldots \forall\alpha_n.\rho$

then $(x : \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(j,k)}, \{j \geq l, k \geq m\}, x, S\rho^{(l,m)})$

where $S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$, the β_i's are fresh type variables,

and j, k, l and m are fresh use variables

else $(x : \dot{\alpha}^{(j,k)}, \emptyset, \{j \geq l, k \geq m\}, x, \dot{\alpha}^{(l,m)})$

where $\dot{\alpha}, j, k, l, m$ are fresh type/use variables

$PTUV(i^\kappa, As) = (\emptyset, \emptyset, i^\kappa, int^\kappa)$

$PTUV(e_1 +_\kappa e_2, As) =$ let

$(\Gamma_1, \Theta_1, e_1, \tau_1) = PTUV(e_1, As)$

$(\Gamma_1, \Theta_2, e_2, \tau_2) = PTUV(e_2, As)$

$(S_1, \Gamma', \Theta') = \bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\})$

$S_2 = U(\{(\tau_1, int^k), (\tau_2, int^l)\})$

in $(S_2\Gamma', S_2(\Theta' \cup \{k \geq 1, l \geq 1\}), e_1 +_\kappa e_2, int^\kappa)$

where j, k and l are fresh use variables.

$PTUV(e_1\ e_2, As) =$ let

$(\Gamma_1, \Theta_1, e_1, \tau_1) = PTUV(e_1, As)$

$(\Gamma_2, \Theta_2, e_2, \tau_2) = PTUV(e_2, As)$

$(S_1, \Gamma', \Theta') = \bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\})$

$S_2 = U(\{(S_1\tau_1, (S_1\tau_2) \xrightarrow{(j,k)} \dot{\delta}^{(l,m)}), (S_1\tau_2, \bar{\gamma}^{(n,p)})\})$

in $(S_2\Gamma', S_2(\Theta' \cup \{k \geq 1\}), e_1\ e_2, S_2\dot{\delta}^{(l,m)})$

where $\bar{\gamma}, \dot{\delta}, j, k, l, m, n, p$ are fresh type/use variables

Figure 4.2: Type Reconstruction Algorithm PTUV (part 1)

$$
\begin{array}{l}
PTUV((\lambda x.e)^{\kappa}, As) = \text{let } (\Gamma, \Theta, e, \tau) = PTUV(e, As) \\
\qquad \text{in if } x \in dom(\Gamma) \\
\qquad\quad \text{then } (\Gamma', \Theta', (\lambda x.e)^{\kappa}, \tau' \xrightarrow{(0,\kappa)} \tau) \\
\qquad\qquad \text{where } (\Gamma', \Theta') = \bigodot(\Gamma \setminus x : \tau', \Theta, \kappa) \\
\qquad\quad \text{else } (\Gamma', \Theta', (\lambda x.e)^{\kappa}, \dot{\alpha}^{(j,k)} \xrightarrow{(0,\kappa)} \tau) \\
\qquad\qquad \text{where } \dot{\alpha}, j, k \text{ are fresh type/use variables and} \\
\qquad\qquad\qquad (\Gamma', \Theta') = \bigodot(\Gamma, \Theta, \kappa) \\
PTUV((\textbf{rec } f(x) = e)^{\omega}, As) = \\
\qquad \text{let } (\Gamma, \Theta, e, \tau) = PTUV(e, As) \\
\qquad \text{in if } x \in dom(\Gamma) \wedge f \in dom(\Gamma) \\
\qquad\quad \text{then let } S = U(\{(\tau_f, \tau_x \xrightarrow{(0,\omega)} \tau)\}) \\
\qquad\qquad\qquad \text{where } \tau_f = \Gamma(f), \tau_x = \Gamma(x) \\
\qquad\qquad\quad (\Gamma', \Theta') = \bigodot(S(\Gamma \setminus (x : \tau_x, f : \tau_f)), S\Theta, \omega) \\
\qquad\qquad \text{in } (\Gamma', \Theta', (\textbf{rec } f(x) = e)^{\omega}, S\tau_f) \\
\qquad\quad \text{else if } x \in dom(\Gamma) \wedge f \notin dom(\Gamma) \\
\qquad\quad \text{then } (\Gamma', \Theta', (\textbf{rec } f(x) = e)^{\omega}, \tau_x \xrightarrow{(0,\omega)} \tau) \\
\qquad\qquad \text{where } (\Gamma', \Theta') = \bigodot(\Gamma \setminus x : \tau_x, \Theta, \omega) \\
\qquad\quad \text{else if } x \notin dom(\Gamma) \wedge f \in dom(\Gamma) \\
\qquad\quad \text{then let } S = U(\{(\tau_f, \dot{\alpha}^{(j,k)} \xrightarrow{(0,\omega)} \tau)\}) \\
\qquad\qquad\qquad \text{where } \dot{\alpha}, j, k \text{ are fresh and } \tau_f = \Gamma(f) \\
\qquad\qquad\quad (\Gamma', \Theta') = \bigodot(S(\Gamma \setminus f : \tau_f), S\Theta, \omega) \\
\qquad\qquad \text{in } (\Gamma', \Theta', (\textbf{rec } f(x) = e)^{\omega}, S\tau_f) \\
\qquad\quad \text{else } (\Gamma', \Theta', (\textbf{rec } f(x) = e)^{\omega}, \dot{\alpha}^{(j,k)} \xrightarrow{(0,\omega)} \tau) \\
\qquad\qquad \text{where } (\Gamma', \Theta') = \bigodot(\Gamma, \Theta, \omega) \text{ and } \dot{\alpha}, j, k \text{ are fresh}
\end{array}
$$

Figure 4.3: Type Reconstruction Algorithm PTUV (part 2)

$$
\begin{array}{l}
PTUV(\{l_1 = e_1, \ldots, l_n = e_n\}^{\kappa}, As) = \\
\quad \text{let } (\Gamma_1, \Theta_1, e_1, \rho_1^{(\kappa_{11}, \kappa_{12})}) = PTUV(e_1, As) \\
\qquad \vdots \\
\quad (\Gamma_n, \Theta_1, e_n, \rho_n^{(\kappa_{n1}, \kappa_{n2})}) = PTUV(e_n, As) \\
\quad (S, \Gamma', \Theta') = \bigoplus(\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\}) \\
\quad \Theta'' = \Theta' \cup S(\{\kappa_{11} \geq \kappa \cdot l_{11}, \ldots, \kappa_{n1} \geq \kappa \cdot l_{n1}\} \\
\qquad\qquad \cup \{\kappa_{12} \geq \kappa \cdot l_{12}, \ldots, \kappa_{n2} \geq \kappa \cdot l_{n2}\}) \\
\qquad \text{where } l_{i1}\text{'s and } l_{i2}\text{'s are fresh} \\
\quad \text{in } (\Gamma', \Theta'', \{l_1 = e_1, \ldots, l_n = e_n\}^{\kappa}, \{l_1 : S\rho_1^{(l_{11}, l_{12})}, \ldots, l_n : S\rho_n^{(l_{n1}, l_{n2})}\}^{\kappa}) \\
PTUV(\textbf{match } \{l_1 = x_1, \ldots, l_n = x_n\} = e_1 \textbf{ in } e_2 \textbf{ end}, As) = \\
\quad \text{let } (\Gamma_1, \Theta_1, e_1, \tau') = PTUV(e_1, As) \\
\quad (\Gamma_2, \Theta_2, e_2, \tau) = PTUV(e_2, As) \\
\quad (S_1, \Gamma', \Theta') = \bigoplus(\Gamma_1, \Theta_1, \Gamma_2 \setminus (x_i : \tau_i \text{ s.t. } x_i \in dom(\Gamma_2)), \Theta_2) \\
\quad S_2 = U(\{(\{l_1 : \bar{\alpha}_1^{(k_1, l_1)}, \ldots, l_n : \bar{\alpha}_n^{(k_n, l_n)}\}^j, S_1\tau')\} \\
\qquad\qquad \cup \{(S_1\tau_i, \bar{\alpha}_i^{(k_i, l_i)}) | x_i \in dom(\Gamma_2)\}) \\
\qquad \text{where } j, \bar{\alpha}_i\text{'s, } k_i\text{'s and } l_i\text{'s are fresh} \\
\quad \text{in } (S_2\Gamma', S_2(\Theta' \cup \{j \geq 1\}), \\
\qquad \textbf{match } \{l_1 = x_1, \ldots, l_n = x_n\} = e_1 \textbf{ in } e_2 \textbf{ end}, S_2S_1\tau) \\
PTUV(\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}, As) = \\
\quad \text{let } (\Gamma_1, \Theta_1, e_1, \tau_1) = PTU(e_1, As) \\
\quad As' = As \cup \{x : clos(\Gamma_1, \tau_1)\} \\
\quad (\Gamma_2, \Theta_2, e_2, \tau_2) = PTU(e_2, As') \\
\quad \text{in if } x \in dom(\Gamma_2) \text{ then} \\
\qquad \text{then } (S_2\Gamma', S_2\Theta', \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}, S_2S_1\tau_2) \\
\qquad\quad \text{where } (S_1, \Gamma', \Theta') = \bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2 \setminus x : \sigma_x, \Theta_2)\}) \\
\qquad\qquad S_2 = U(\{(S_1 clos(\Gamma_1, \tau_1), S_1\sigma_x), (\bar{\alpha}^{(j,k)}, S_1\tau_1)\}) \\
\qquad\qquad \bar{\alpha}, j, k \text{ are fresh (type/use) variables} \\
\qquad \text{else } (S_2\Gamma', S_2\Theta', \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end}, S_2S_1\tau_2) \\
\qquad\quad \text{where } (S_1, \Gamma', \Theta') = \bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\}) \\
\qquad\qquad S_2 = U(\{(\bar{\alpha}^{(j,k)}, S_1\tau_1)\}) \\
\qquad\qquad \bar{\alpha}, j, k \text{ are fresh}
\end{array}
$$

Figure 4.4: Type Reconstruction Algorithm PTUV (part 3)

$$
\begin{array}{l}
PTUV(e_1 \mid e_2, As) = \text{let} \\
\qquad (\Gamma_1, \Theta_1, e_1, \tau_1) = PTUV(e_1, As) \\
\qquad (\Gamma_2, \Theta_2, e_2, \tau_2) = PTUV(e_2, As) \\
\qquad (S_1, \Gamma', \Theta') = \bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\}) \\
\qquad S_2 = U(\{(S_1\tau_1, O), (S_1\tau_2, O)\}) \\
\quad \text{in } (S_2\Gamma', S_2\Theta', e_1 \mid e_2, O) \\
PTUV(e_1 \& e_2, As) = \text{let} \\
\qquad (\Gamma_1, \Theta_1, e_1, \tau_1) = PTUV(e_1, As) \\
\qquad (\Gamma_2, \Theta_2, e_2, \tau_2) = PTUV(e_2, As) \\
\qquad (S_1, \Gamma', \Theta') = \bigsqcup(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\}) \\
\qquad S_2 = U(\{(S_1\tau_1, O), (S_1\tau_2, O)\}) \\
\quad \text{in } (S_2\Gamma', S_2\Theta, e_1 \ \& \ e_2, O) \\
PTUV(x(y) \Rightarrow e, As) = \text{let} \\
\qquad\quad (\Gamma_2, \Theta_2, e, \tau_2) = PTUV(e, As) \\
\qquad \text{in if } y \in dom(\Gamma_2) \text{ then} \\
\qquad\quad (S_2\Gamma', S_2\Theta', x(y) \Rightarrow e, O) \\
\qquad\qquad \text{where } (S_1, \Gamma', \Theta') = \bigoplus(\{(x : \dot\alpha^{(l,m)} \xrightarrow{(j,k)} O, \{j \geq 1\}), \\
\qquad\qquad\qquad\qquad\qquad\qquad (\Gamma_2 \setminus y : \tau', \Theta_2)\}) \\
\qquad\qquad\qquad S_2 = U(\{(S_1(\dot\alpha^{(l,m)}), S_1\tau'), (S_1\tau_2, O)\}) \\
\qquad\qquad\qquad \dot\alpha, j, k, l, m \text{ are fresh} \\
\qquad\quad \text{else } (S_2\Gamma', S_2\Theta', x(y) \Rightarrow e, O) \\
\qquad\qquad \text{where } (S_1, \Gamma', \Theta') = \bigoplus(\{(x : \dot\alpha^{(l,m)} \xrightarrow{(j,k)} O, \{j \geq 1\}), (\Gamma_2, \Theta_2)\}) \\
\qquad\qquad\qquad S_2 = U(\{(S_2\tau_2, O)\}) \\
\qquad\qquad\qquad \dot\alpha, j, k, l, m \text{ are fresh} \\
PTUV(\textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end}, As) = \\
\qquad \text{let } (\Gamma, \Theta, e, \tau) = PTUV(e, As) \\
\qquad \text{in if } x \in dom(\Gamma) \text{ then} \\
\qquad\quad (S(\Gamma \setminus x : \tau_x), S\Theta, \textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end}, O) \\
\qquad\qquad \text{where } S = U(\{(\tau_x, \dot\alpha^{(l,m)} \xrightarrow{(\kappa_1,\kappa_2)} O), (\tau, O)\}) \\
\qquad\qquad\qquad \dot\alpha, l, m \text{ are fresh} \\
\qquad \text{else } (S\Gamma, S\Theta, \textbf{new } x^{(\kappa_1,\kappa_2)} \textbf{ in } e \textbf{ end}, O) \\
\qquad\qquad\qquad \text{where } S = U(\{(\tau, O)\})
\end{array}
$$

Figure 4.5: Type Reconstruction Algorithm PTUV (part 4)

if $n < 2$ **then** $c(1)$

else new $c1, c2$ **in**

$(fib(n-1, c1) \mid c1(x) \Rightarrow (fib(n-2, c2) \mid c2(y) \Rightarrow c(x+y)))$

end

By solving the use constraint set of the principal typing obtained by $PTUV$, our analysis concludes $c1$ and $c2$ are linear. Then, the second argument of the recursive calls can be replaced with functions which behave as same as after receiving from linear channels, that is, $c2$ in $fib(n-1, c2)$ is replaced with $\lambda y.c(x+y)$. As a result, the optimized term will be

rec $fibopt(n, c) =$

if $n < 2$ then $c(1)$

else $fibopt(n-1, \lambda x.(fibopt(n-2, \lambda y.c(x+y))))$

The optimized term corresponds to the continuation passing style[2] representation of the functional Fibonacci program.

4.2 Correctness of Uses of Values

In this section, we prove the correctness of uses of values. To handle the uses of values in operational semantics, the notion of *heap*, which is an abstraction of memory space, is introduced. Formally, heap is a mapping from variables to pairs of a *heap value*, which is a value of a restricted form, and its use. A snapshot of execution is represented as a pair of a heap and a preterm, and the reduction relation is defined as a relation between such two pairs. As uses of free channels are decreased in the type environment, uses of other values are decreased in the heap during reduction. As in Section 2.3, correctness is shown through Subject Reduction Theorem (3) (Theorem 4.4) and Run-time Safety Theorem (2) (Theorem 4.5).

4.2.1 Heap

The formal definitions of heap values and heap are given as follows:

Definition 4.5 (heap values, heap) The set of *heap values*, ranged over by h, is a subset of the set of values v. A heap value is either a integer, a λ-abstraction, a recursive function, or a record whose elements are all variables. A *heap* H is a mapping from variables to pairs of a heap value and a use.

$$\begin{array}{rcl} h & ::= & i|\lambda x.e|\mathbf{rec}\ f(x) = e|\{l_1 = x_1, \ldots, l_n = x_n\} \\ H & ::= & x_1 = h_1^{\kappa_1}, \ldots, x_n = h_n^{\kappa_n} \end{array}$$

Intuitively, heap values are values (other than channels) which can be allocated to memory spaces. Hence, a record is restricted to consist of variables, which can be considered as pointers to other values. The definitions of $H, x = h$ and $dom(H)$ are obtained in the similar manner to type environments.

4.2.2 Operational Semantics with Heap

Structural congruence relation is defined same as before (See Definition 2.3). Before describing the whole reduction relation, the reduction relation corresponding actions other than communications, which includes function application, extraction from a record, etc., is described. The relation has the form $\texttt{letrec}\ H\ \texttt{in}\ e \longrightarrow_\lambda \texttt{letrec}\ H'\ \texttt{in}\ e'$. $\texttt{letrec}\ H\ \texttt{in}\ e$ represents the snapshot of an execution where a free variable x of e can be used κ times as the heap value h if $H(x) = h^\kappa$.

The below definition of *evaluation contexts* (E) represents that such execution steps are done from left to right, i.e., in the call-by-value manner. An instruction (I), which is a redex of the reduction, is either (heap-allocated) function application to a value, summation of two (heap-allocated) integers, heap values (with its use), **let** declaration which completes the evaluation of the locally-bound value, or extraction from a (heap-allocated) record.

Definition 4.6 (Evaluation Context, Instruction)

$$
\begin{array}{rcl}
E & ::= & [\,] \mid E + e \mid x + E \mid Ee \mid xE \mid \textbf{let } x = E \textbf{ in } e \textbf{ end} \\
& \mid & \{l_1 = x_1, \ldots l_{i-1} = x_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\} \\
& \mid & \textbf{match } \{l_1 = x_1, \ldots, l_n = x_n\} = E \textbf{ in } e \textbf{ end} \\
I & ::= & x\ y \mid x + y \mid h^\kappa \mid \textbf{let } x = y \textbf{ in } e \textbf{ end} \\
& \mid & \textbf{match } \{l_1 = x_1, \ldots, l_n = x_n\} = y \textbf{ in } e \textbf{ end}
\end{array}
$$

The reduction relation for sequential execution is defined as follows where we write $E[e]$ for the preterm obtained by substituting e for [] in E.

Definition 4.7 (reduction relation $\texttt{letrec } H \texttt{ in } e \xrightarrow{\tilde{x}}_\lambda \texttt{letrec } H' \texttt{ in } e'$) The reduction relation $\texttt{letrec } H \texttt{ in } e \xrightarrow{\tilde{x}}_\lambda \texttt{letrec } H' \texttt{ in } e'$ is the least relation closed under the following rules.

(L-Alloc) $\texttt{letrec } H \texttt{ in } E[h^\kappa]$

$\quad \longrightarrow_\lambda \texttt{letrec } H, x = h^\kappa \texttt{ in } E[x]$

(L-Plus) $\texttt{letrec } H, x = i_1^{\kappa_1}, y = i_2^{\kappa_2} \texttt{ in } E[x +_\kappa y]$

$\quad \xrightarrow{x,y}_\lambda \texttt{letrec } H, x = i_1^{\kappa_1}, y = i_2^{\kappa_2}, z = (\underline{i_1 + i_2})^\kappa \texttt{ in } E[z]$

where $\underline{i_1 + i_2}$ denotes the mathematical summation of two integers i_1 and i_2.

(L-App) $\texttt{letrec } H, x = (\lambda z.e)^\kappa \texttt{ in } E[xy]$

$\quad \xrightarrow{x}_\lambda \texttt{letrec } H, x = (\lambda z.e)^\kappa \texttt{ in } E[[y/z]e]$

(L-Fix) $\texttt{letrec } H, x = (\textbf{rec } f(z) = e)^\kappa \texttt{ in } E[xy]$

$\quad \xrightarrow{x}_\lambda \texttt{letrec } H, x = (\textbf{rec } f(z) = e)^\kappa \texttt{ in } E[[y/z, x/f]e]$

(L-Let) $\texttt{letrec } H \texttt{ in } E[\textbf{let } x = y \textbf{ in } e \textbf{ end}]$

$\quad \longrightarrow_\lambda \texttt{letrec } H \texttt{ in } E[[y/x]e]$

(L-Match) $\texttt{letrec } H, y = \{l_1 = z_1, \ldots, l_n = z_n\}^\kappa \texttt{ in } E[\textbf{match } \{l_1 = x_1, \ldots, l_n = x_n\} = y \textbf{ in } e \textbf{ end}]$

$\quad \xrightarrow{y}_\lambda \texttt{letrec } H, y = \{l_1 = z_1, \ldots, l_n = z_n\}^\kappa \texttt{ in } E[[z_1/x_1, \ldots, z_n/x_n]e]$

The rule (L-Alloc) models the allocation of a (heap) value by binding the value to a new variable. The variables on the arrow denotes the used heap values in the reduction. The rule (L-Plus) represents that two integers x and y are used. Similarly, a function and a record are used by applications and extraction, respectively.

The summation of two heaps $H_1 + H_2$ used in the definition of the reduction relation below is defined as follows:

$$(H_1 + H_2)(x) = \begin{cases} h^{\kappa_1+\kappa_2} & \text{if } x \in (dom(\Gamma_1) \cap dom(\Gamma_2)) \\ & \text{and } H_1(x) = h^{\kappa_1}, H_2(x) = h^{\kappa_2} \\ H_1(x) & \text{if } x \in (dom(H_1) \setminus dom(H_2)) \\ H_2(x) & \text{if } x \in (dom(H_2) \setminus dom(H_1)) \end{cases}$$

The whole reduction relation is defined as the straightforward extension of Definition 2.10. The label $\varepsilon : \tilde{x}$ means not only communication on a bound channel, but also a reduction other than communications derived from $\xrightarrow{\tilde{x}}_\lambda$.

Definition 4.8 (reduction relation $\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ e'$)

$$\frac{\kappa_1 \geq 1 \quad \kappa_2 \geq 1}{\Gamma, x : \tau \overset{(\kappa_1,\kappa_2)}{\longrightarrow} O \vdash \texttt{letrec}\ H\ \texttt{in}\ ((x(y) \Rightarrow e)\&e') \,|\, x(z) \xrightarrow{x} \texttt{letrec}\ H\ \texttt{in}\ [z/y]e} \text{ (ERu-Comm)}$$

$$\frac{e_1 \cong e_2 \quad \Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e_1 \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ e_1' \quad e_1' \cong e_2'}{\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e_2 \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ e_2'} \text{ (ERu-Cong)}$$

$$\frac{\texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{\tilde{x}}_\lambda \texttt{letrec}\ H'\ \texttt{in}\ e'}{\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{\varepsilon:\tilde{x}} \texttt{letrec}\ H'\ \texttt{in}\ e'} \text{ (ERu-Lamb)}$$

$$\frac{\Gamma_1 \vdash \texttt{letrec}\ H_1\ \texttt{in}\ e_1 \xrightarrow{l} \texttt{letrec}\ H_1'\ \texttt{in}\ e_1'}{\Gamma_1 + \Gamma_2 \vdash \texttt{letrec}\ H_1 + H_2\ \texttt{in}\ e_1 \,|\, e_2 \xrightarrow{l} \texttt{letrec}\ H_1' + H_2\ \texttt{in}\ e_1' \,|\, e_2} \text{ (ERu-Par)}$$

$$\frac{\Gamma, x : \tau \overset{(\kappa_1,\kappa_2)}{\longrightarrow} O \vdash \texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{x} \texttt{letrec}\ H\ \texttt{in}\ e'}{\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ \textbf{new}\ x^{(\kappa_1,\kappa_2)}\ \textbf{in}\ e\ \textbf{end} \xrightarrow{\varepsilon} \texttt{letrec}\ H\ \texttt{in}\ \textbf{new}\ x^{(\kappa_1^-,\kappa_2^-)}\ \textbf{in}\ e'\ \textbf{end}} \text{ (ERu-New1)}$$

$$\frac{\Gamma, x : \tau \overset{(\kappa_1,\kappa_2)}{\longrightarrow} O \vdash \texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ e' \quad x \neq l}{\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ \textbf{new}\ x^{(\kappa_1,\kappa_2)}\ \textbf{in}\ e\ \textbf{end} \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ \textbf{new}\ x^{(\kappa_1,\kappa_2)}\ \textbf{in}\ e'\ \textbf{end}} \text{ (ERu-New2)}$$

4.2.3 Correctness

To state the subject reduction theorem about this operational semantics, the definition of well-typedness of $\texttt{letrec}\ H\ \texttt{in}\ e$ is given after the following preliminary definition.

Definition 4.9 Type environment Γ is *split into* Γ_1 *and* Γ_2 *with respect to a heap* H if and only if $\Gamma = \Gamma_1 + \Gamma_2$, $dom(\Gamma_1) \cap dom(H) = \emptyset$ and $dom(\Gamma_2) \subseteq dom(H)$, for

which we write $\Gamma = \Gamma_1 \uplus_H \Gamma_2$.

The following lemma is trivial.

Lemma 4.1 Given H and Γ, there uniquely exist Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 \uplus_H \Gamma_2$.

The well-typedness of `letrec` H `in` e is defined by the following rule.

$$\frac{\begin{array}{l}\Gamma_e \vdash e : \tau \qquad \Gamma_e = \Gamma_{e_1} \uplus_H \Gamma_{e_2} \\ \forall x \in dom(H). \left(\begin{array}{c} \Gamma_x \vdash H(x) : \tau_x \quad \Gamma(x) = clos(\Gamma_x, \tau_x) \\ \Gamma_x = \Gamma_{x1} \uplus_H \Gamma_{x2} \end{array} \right) \\ \Gamma = \Gamma_{x_1 2} + \cdots + \Gamma_{x_n 2} + \Gamma_{e2} \qquad dom(H) = \{x_1, \ldots, x_n\}\end{array}}{\Gamma_{x_1 1} + \cdots + \Gamma_{x_n 1} + \Gamma_{e_1} \vdash \texttt{letrec}\ H\ \texttt{in}\ e : \tau} \ (\text{ETU-HEAP})$$

Intuitive meanings of the conditions are:

1. e must be well-typed.
2. All heap values bound in H must be well-typed.
3. Γ keeps total use information on (heap-allocated) values used in other heap values and e. Therefore, uses of $\Gamma(x)$ and $H(x)$ must be the same.

$\Gamma_{x_i 1}$'s and Γ_{e1} are type environments whose domain have no intersection with the heap, that is, the variables in these type environments can be possibly used as free channels.

H^{-l}, defined below, denotes the heap after the reduction.

Definition 4.10 H^{-l} is defined by: $H^{-l} = H$, $H^{-\varepsilon} = H$, $(H, x = h^{\kappa})^{-\varepsilon:x,\tilde{y}} = (H, x = h^{\kappa^-})^{-\varepsilon:\tilde{y}}$.

Now, we state subject reduction theorem and run-time safety theorem.

Theorem 4.4 (Subject Reduction (3)) If $\Gamma \vdash$ `letrec` H `in` $e : \tau$ and $\Gamma \vdash$ `letrec` H `in` $e \xrightarrow{l}$ `letrec` H' `in` e', then $\Gamma^{-l} \vdash$ `letrec` H'^{-l} `in` $e' : \tau$.

Proof Structural induction on the proof of $\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ e'$. See Appendix A.4 for the detailed proof. □

Theorem 4.5 (Run-time Safety (2)) A term has no immediate possibilities of misuse of channels or values. More formally, if $\vdash \texttt{letrec}\ H\ \texttt{in}\ e : \tau$ and $e \cong \mathbf{new}\ x_1^{(\kappa_{11},\kappa_{12})}, \ldots, x_n^{(\kappa_{n1},\kappa_{n2})}\ \mathbf{in}\ (e_1\{|\ e_2\})\ \mathbf{end}$ then:

1. If e_1 is $x(y)\,|\,((x(z) \Rightarrow e'')\&r)$, then for the use pair (κ_1, κ_2) of the binding of x in either Γ or **new**, $(\kappa_1, \kappa_2) \geq (1, 1)$.

2. If e_1 is $x\ y$ and $x \notin dom(H)$, then the send use of the binding of x is greater than 0.

3. If e_1 is $(x(y) \Rightarrow e')\&r$, then the receive use of the binding of x is greater than 0.

4. If e_1 is $x\ y\,|\,x\ z$ and $x \notin dom(H)$, then the send use of the binding of x is ω.

5. If e_1 is $((x(y) \Rightarrow e')\&r_1)\,|\,((x(z) \Rightarrow e'')\&r_2)$ then the receive use of the binding of x is ω.

6. If $e_1 = E[xy]$ for some E and $H(x) = (\lambda z.e)^{\kappa}$, then $\kappa \geq 1$.

7. If $e_1 = E[xy]$ for some E and $H(x) = (\mathbf{rec}\ f(z) = e)^{\kappa}$, then $\kappa = \omega$.

8. If $e_1 = E[x + y]$ for some E, then $H(x) = i_1^{\kappa_x}$, $H(y) = i_2^{\kappa_y}$, $\kappa_x \geq 1$, and $\kappa_y \geq 1$.

9. If $e_1 = E[\mathbf{match}\ \{l_1 = x_1, \ldots, l_n = x_n\} = y\ \mathbf{in}\ e\ \mathbf{end}]$ for some E, then the use of $H(y)$ is greater than 0.

Proof Trivial from typing rules. □

4.3 Summary

The target language has been extended from a pure process calculus to a process calculus equipped with integers, functions, data constructors such as records, and let-polymorphism. A type system with uses for the extended language and a type reconstruction algorithm has been described. It enables detection of not only linear channels but also other used-once values and further optimizations for concurrent programming languages. Also, the soundness of the type system has been proved. To handle the uses of values, the notion of heap has been introduced into operational semantics.

Chapter 5

Experimental Results

In this chapter, the experimental results about the effects of optimization and the cost of the analysis are shown.

5.1 Evaluation of Optimization

In this section, we show results of simple experiments with *HACL* compiler[1] to evaluate how much the results of our analysis can improve performance of concurrent programs. Application programs are the example of Fibonacci function described in the previous chapter, and concurrent objects expressed by *HACL*. Before showing the results, we mention encoding of concurrent objects first.

5.1.1 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

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

result, and puts back the new state into the channel. So, the following fragment of a preterm corresponds to a typical method definition:

$$\textbf{let } m = \textbf{rec } \; m(arg, r) = \\ \qquad\qquad state(s) \Rightarrow (\ldots \mid r(result) \mid state(news)) \\ \textbf{in } \ldots$$

A caller of the method is typically of the form:

$$\textbf{new } r^{(j,k)} \textbf{ in } (m(v, r) \mid r(x) \Rightarrow e) \textbf{ end}$$

With our analysis, it can be translated to $m(v, \lambda x.e)$ in most cases.

5.1.2 Experiment Results and Evaluation

We evaluate effect of optimization through four kinds of programs: a sequential Fibonacci program (the one shown in the optimization example where $n = 25$), a parallel Fibonacci program (which performs recursive calls in parallel), a counter increment program (which creates a counter object and increments its value 10000 times), and a tree summation program (which creates a binary tree of which each node is a concurrent object, and computes a summation of values of its leaf nodes). Each row in Table 5.1 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 optimized programs by our analysis. The rightmost column ("speedup") shows speedup per one function call or method invocation, which is calculated by (naive − optimized)/(# of communications on linear channels). In addition, we list in the third column the running time of a program written with function primitives for the sequential Fibonacci. Note that all programs are executed on a single processor machine.[2] Therefore parallel fibs are slower than sequential fibs because of overheads.

[2]SS20 (Hyper SPARC 150Mhz)

	naive (sec)	optimized (sec)	function (sec)	speedup (μsec)
sequential fib	1.45	0.57	0.41	3.6
parallel fib	1.76	0.93	—	3.4
`counter`	0.26	0.17	—	9.0
`tree14`	1.55	1.36	—	5.8

Table 5.1: Running time and speedup for Fibonacci, `counter` and `tree14`

The result of the sequential Fibonacci program indicates that even if programmers encode functional computations with concurrency primitives, the compiler can generate an optimized code which is comparable to those written directly using function primitives. The speedup rate of the parallel Fibonacci program is relatively smaller because of overheads of multi-threading, but it is still large. Note that the speedup rate (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). For more general programs, the speedup rate will be much smaller.

The last two programs (`counter` and `tree14`) are measured to estimate effect of our optimization for typical concurrent object-oriented programs. In `counter`, method invocations are much more frequent than creations of concurrent objects, while in `tree14`, creations of concurrent objects happen as frequent as method invocations do. The figures of the table show that the speedup per one communication on a linear channel is larger than for Fibonacci programs.

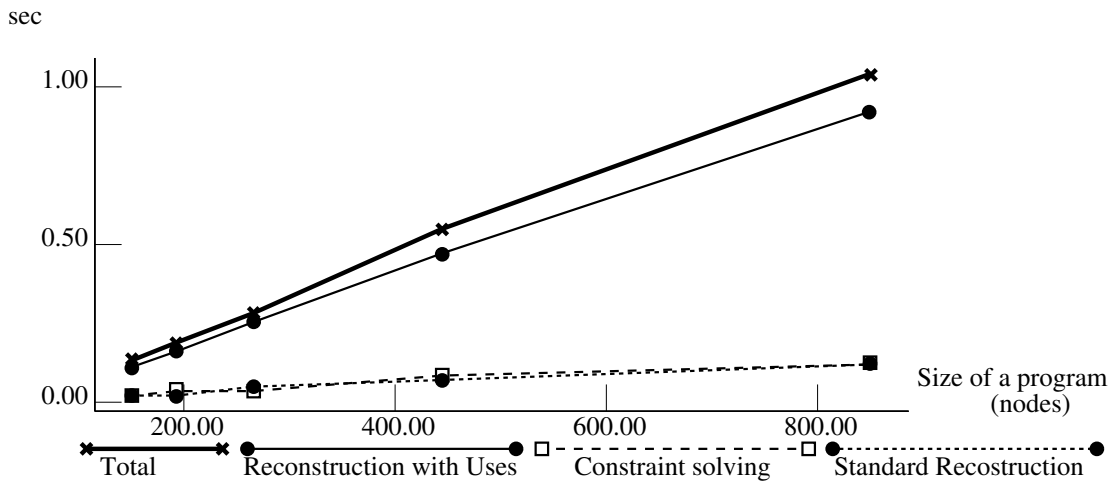

Figure 5.1: Elapsed time for Our Analysis and Ordinary Type Reconstruction

5.2 Cost of Analysis

In this section, we compare the cost of our analysis with standard type reconstruction (not including use information). The cost of our analysis consists of that of type reconstruction and that for solving a use constraint set. We show the elapsed (system) time for several programs in the Figure 5.1. The analysis program is implemented with Standard ML of New Jersey 0.93 and the elapsed time is measured on SS20 (Hyper SPARC 150Mhz).

The horizontal and vertical axis represents the size of the syntax tree of a term and the elapsed time, respectively. The solid line connected with $\bullet$ denotes the elapsed time for type reconstruction with uses while the broken line with $\bullet$ does for standard type reconstruction. The broken line connected with $\Box$ denotes one for solving the obtained use constraint sets. The total time of our analysis is represented as the bold line connected with $\times$. This results tells us that type reconstruction with uses is efficient enough for practice although the complexity of type reconstruction is theoretically exponential. Moreover, the cost of solving a use constraint set, to be seen later that its complexity is polynomial, is much less

than that of type reconstruction.

Chapter 6

Discussion

In this chapter, we give several discussions. In Section 6.1, to refine the analysis, subtyping and polymorphism on uses are discussed as extensions of the type system. In Section 6.2, we discuss several implementation issues about linear values. In Section 6.3, we see the computational complexity of our analysis is polynomial in the size of preterms.

6.1 Subtyping and Polymorphism

Our simple type system often suffers from worse result of the analysis than expected. It is mainly derived from (ETu-App/Send) rule, which tells us the type of the value to be sent must be the same as the domain type of the channel including use information. It often makes too many channels have the same (bare) type. For example, consider the following term:

$$\begin{array}{l}\textbf{let } x = \textbf{rec } x(y) = y\ e \textbf{ in} \\ \quad \textbf{let } z = \textbf{rec } z(y) = (y\ e_1 \mid y\ e_2) \textbf{ in} \\ \quad\quad m\ x \mid m\ z \mid\ \textbf{new } u \textbf{ in } (x\ u \mid u(y) \Rightarrow e_3) \textbf{ end} \\ \textbf{end end}\end{array}$$

Since y in **rec** $z(y) = \ldots$ is used for sending twice, the type of z is, for example, $(\tau \xrightarrow{(0,\omega)} O) \xrightarrow{(0,\omega)} O$. Our typing rules force x to have the same type as z. As

a result, our type system concludes u, which is passed along x, is not a linear channel, though it will be used only once for sending and receiving, respectively. This example shows the possibility that only one misuse (or maybe intentional use) of channels leads many channels to be non-linear channels. As another example, uniform lists, which consists of elements of a uniform type, can cause the same kind of problems.

Subtyping To avoid these problems, we could introduce a subtyping relation and modify the typing rule for application. For instance, the rules for the subtyping relation $\tau_1 \preceq \tau_2$, and the typing rule for application would be below.

$$\tau \preceq \tau \ (\text{S-Refl}) \qquad \frac{\tau_1 \preceq \tau_2 \quad \tau_2 \preceq \tau_3}{\tau_1 \preceq \tau_3} \ (\text{S-Tran}) \qquad \frac{(\kappa_1, \kappa_2) \geq (\kappa_3, \kappa_4)}{\rho^{(\kappa_1,\kappa_2)} \preceq \rho^{(\kappa_3,\kappa_4)}} \ (\text{S-Use})$$

$$\frac{\tau_3 \preceq \tau_1 \quad \tau_2 \preceq \tau_4}{\tau_1 \xrightarrow{(0,\kappa)} \tau_2 \preceq \tau_3 \xrightarrow{(0,\kappa)} \tau_4} \ (\text{S-Fun1}) \qquad \frac{\tau_1 \preceq \tau_2}{\tau_1 \xrightarrow{(\kappa,0)} O \preceq \tau_2 \xrightarrow{(\kappa,0)} O} \ (\text{S-Fun2})$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau_2 \quad \kappa_2 \geq 1 \quad \tau_3 \not\simeq O \quad \Gamma_2 \vdash e_2 : \tau_3 \quad \tau_3 \preceq \tau_1}{\Gamma_1 + \Gamma_2 \vdash e_1 \ e_2 : \tau_2} \ (\text{ETu-App/Send})$$

The first two rules represent reflectivity and transitivity of the subtyping relation. (S-Use) rule allows us to regard $\rho^{(\kappa_1,\kappa_2)}$ as the type whose uses are smaller than (κ_1, κ_2), that is, it tells us, for example, "the value which can be used an arbitrary number of times is regarded as used at most once." (S-Fun1) rule is the same as the standard subtyping relation between function types. (S-Fun2) rule represents that the channel to receive τ type values can be used as the channel to receive values of its supertype.

Since $(\tau \xrightarrow{(0,1)} O) \xrightarrow{(0,\omega)} O \preceq (\tau \xrightarrow{(0,\omega)} O) \xrightarrow{(0,\omega)} O$ could be derived from these rules, even if z has the type $(\tau \xrightarrow{(0,\omega)} O) \xrightarrow{(0,\omega)} O$, we could give the type $(\tau \xrightarrow{(0,1)} O) \xrightarrow{(\omega,0)} O$ to x and conclude u is linear.

By introducing subtyping, a type reconstruction algorithm would output $(\Gamma, \Theta, \preceq, e, \tau)$ where $\preceq$ is subtyping constraint which should be satisfied by τ and

the types in the Γ. Although the type reconstruction algorithm itself would not require so much modification, it would be complex to solve the subtyping constraint. The complexity is derived from subtyping rules for function types. For example, consider the following subtyping relation $\tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau_2 \preceq \tau_3 \xrightarrow{(\kappa_3,\kappa_4)} \tau_4$. Applicable subtyping rules depends on the uses of the types, that is, (S-FUN1) rule can be applied only when $\kappa_1 = \kappa_3 = 0$ and so forth.

Polymorphism on Use Variables Another solution for the problems is to allow polymorphism not only on type variables but also on use variables. For instance, the type scheme of x in the above example would be expressed as:

$$\tau \xrightarrow{(j,k)} O \xrightarrow{(0,\omega)} O_{\text{where}}\{k \geq 1\}$$

It means that x is used as a function whose type is $(\tau \xrightarrow{(\kappa_1,\kappa_2)} O) \xrightarrow{(0,\omega)} O$ for any κ_i such that $\kappa_2 \geq 1$. Polymorphism on use variables would allow x and z, which are sent to m, to have different types and u to be a linear channel.

Theoretically, however, introduction of use polymorphism makes the size of a use constraint set in type judgements exponential for the same reason in the case of polymorphism on type variables. Besides the compiler needs to generate different versions of a polymorphic function for different instantiations of the use variables. Further experiments will be necessary to evaluate the trade-off about polymorphism on uses.

6.2 How to Utilize Use Information

We discuss how the compiler utilizes uses of values for optimization. As is shown through the examples until now, uses of channels enable program transformation. Besides, we reduce the cost of garbage collection by releasing the memory space for linear values immediately after they are used.

It is not straightforward, however, to implement such optimization because our

type system allows for non-linear values to be mingled with linear values. In other words, even if a function has a type $int^1 \xrightarrow{(\kappa_1,\kappa_2)} \tau$, it is ensured only that the passed integer will be used at most once in the body. The integer may be used in other places. Fortunately, such mixture does not matter so much for implementation of channels (closures). In the implementation of *HACL*, the data structure for channels includes the pointer to the code for communication where such explicit release of memory can be embedded. Therefore, the difference between linear channels (closure) and non-linear ones can be encapsulated in their data structure and they can be treated in a uniform manner in the generated code. In order to distinguish records, however, a run-time tag, which represents their uses, is required to the data structure for records.

To avoid such mixture by a type system, we would modify the typing rules. For example, (ETu-Var) would be:

$$\frac{S_\rho \text{ is a substitution } [\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]}{\Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau \vdash x : S_\rho\tau} \text{ (ETu-Var)}$$

According to this typing rule, the type judgement like $x : int^\omega \vdash x : int^1$ is not allowed, that is, the value of ω cannot be regarded as a value of 1. Also, the definition of $+$ must be modified. $1 + 1 = \omega$ represents the fact that "a non-linear value can be used as a linear value in subexpressions." So, for the intended purpose, summation of such uses, i.e., $1+1$, $1+\omega$ and $\omega+1$, should be undefined. For the similar reason, $1 \sqcup \omega$ and $\omega \sqcup 1$ also should not be defined.

With this modification, use constraint would be a system of *equations* on use variables. It would be more complex to solve than that of our system.

6.3 Computational Complexity of Analysis

It is well-known that the complexity of type reconstruction for languages, including our target language, with lambda abstraction, function application, and let-polymorphism is exponential in the size of a program[12]. In this section, we see

that the additional cost for the analysis, that is, the time complexity for solving the use constraint set of the principal typing is polynomial in the preterm.

First, we give several preliminary definitions. The size of a use expression $size(\kappa)$ is defined as 1 if κ is either a constant or a variable and $1 + size(\kappa_1) + size(\kappa_2)$ if κ is $\kappa_1 + \kappa_2$, $\kappa_1 \sqcup \kappa_2$, or $\kappa_1 \cdot \kappa_2$. $size(\Theta)$, $\#(s)$, and $\#_{FV}(e)$ denote the number of inequalities in Θ, the number of variables in s, and the number of free variables of e, respectively.

Let $\{j_1 \geq \kappa_1, \ldots, j_n \geq \kappa_n\}$ be the use constraint set to be solved[1]. Since $j_1 = \cdots = j_n = \omega$ is always the solution and $\vec{\kappa}^{(i)}$is monotonously increased from $(0, \ldots, 0)$, the number of steps of iterations is at most $2 \cdot n$. In each iteration step, the cost for computing $\vec{\kappa}^{(m+1)}$ is $O(\sum_i size(\kappa_i))(\leq O(n \cdot \max_i(size(\kappa_i))))$.

Then, we estimate n and $\max_i(size(\kappa_i))$ of a use constraint set generated by $PTUV$. For example, consider the case of $e = e_1 +_\kappa e_2$. Let $(\Gamma_1, \Theta_1, \ldots)$ and $(\Gamma_2, \Theta_2, \ldots)$ be obtained from $PTUV(e_1, As)$ and $PTUV(e_2, As)$, respectively. The size of the use constraint set generated by $\bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\})$ is equal to $2 \cdot \#(dom(\Gamma_1) \cup dom(\Gamma_2)) + size(\Theta_1) + size(\Theta_2)$ by the definition of $\bigoplus$. Note that $\#(dom(\Gamma_1) \cup dom(\Gamma_2))$ is equal to $\#_{FV}(e)$. As a result, the size of Θ obtained from $PTUV(e_1 +_\kappa e_2, As)$ will be:

$$size(\Theta) = 2 \cdot \#_{FV}(e) + size(\Theta_1) + size(\Theta_2) + 2$$

By similar case analyses, we conclude that $size(\Theta)$ obtained from $PTUV(e, As)$ is $O(n^2)$ where n is the size of a preterm e. The maximal size of the use expressions in Θ is equal to the maximal number of children of the nodes of the syntax tree of e.

As a result, the computational complexity for solving the constraint set is polynomial in the size of a preterm. Note that this estimation of the order is very rough and there is better upper bound. Indeed, the experimental results in the Chapter 5 shows the cost for solving is almost linear.

[1]We use the notations in Theorem 3.4.

Chapter 7

Related Work

The most closely related work are Turner, Wadler, and Mossin's work on analysis of linearity for (lazy) functional programming languages[24], and recent work on analysis of communication for concurrent programming languages[16, 7]. In fact, the type system presented in this study was partially inspired by [24] and [7]. In [24], the use can only be either 1 or ω. As a result, in their system, if a variable has more than one syntactic occurrence, its use is inferred to be ω. For example, the use of y in **let** $f = \lambda x.1$ **in** $(fy) + y$ **end** is ω in their system, although y is actually used just once. This roughness of the analysis causes serious damage in the analysis of uses of communication channels, because channels have normally at least two syntactic occurrences, as already mentioned.

Nielson and Nielson[16] proposed another technique that can find some of linear channels using results of their effect-based analysis[15]. However, their technique behave poorly in finding linear channels, because it counts operations on channels *region-wise*, where a region is a set of (possibly infinite) communication channels, rather than channel-wise. For example, in the term **let** $f = (\mathbf{rec}\ f() = m(x) \Rightarrow (x() \mid f()))$ **in** $(f() \mid m(n))$ **end**, the number of send operations performed to the channel x is counted as ω in their analysis while counted as 1 in our type-based analysis. Kobayashi, Nakade and Yonezawa[7] solved this problem partially, and proposed a better (but more costly) analysis to

find linear channels (as well as an analysis to find *linearized*[8] channels). However, as far as linear channels are concerned, our type-based analysis presented here gives *more accurate* results for typical programs with *much less costs*.

As another approach to determine the lifetimes of values, Baker's Linear Lisp[3] forces programmers to use each variable exactly once. But this restriction on programmers is too severe. Above all, the restriction cannot be applied to usage of channels for the same reason as above.

Recently, Kobayashi advanced the linear channel type system of [8] to a type system, which ensures partial deadlock-freedom of processes[6]. In that work, a class of channels, called *reliable channels*, are introduced. If a process is proved to be using only reliable channels by the type system, it never causes deadlock. Conversely, if deadlock occurs, it happens on unreliable channels, which are the other channels than reliable ones. Moreover, a process using a subclass of reliable channels has no non-determinism. In addition to uses, the key idea of that type system is to keep ordering of channel uses as a part of type information. Our technique would be applicable to the type reconstruction for this type system.

Chapter 8

Conclusions

In this thesis, we have proposed a type-based analysis to find use-once values, including linear channels, for concurrent programming languages. This analysis enables several optimizations such as program transformation and reduction of run-time costs for communications. The program transformation includes tail-call optimization of concurrent programs, which is enabled only by nontrivial global analysis unlike that in functional programming.

8.1 Contributions

- We have improved the linear channel type system of [8] and developed a type system which judges how many times each value is used for a realistic core of concurrent programming languages. The soundness of the type system has been proved with respect to the operational semantics.

- A type reconstruction algorithm for the type system and a method for detecting used-once values, including linear channels, have been developed. By the analysis, we can detect used-once values with *no* declarations about types in programs. Although we presented the analysis through a core of concurrent programming languages, the technique proposed here could be applied to other concurrent programming languages, including functional program-

ming languages extended with concurrency primitives[20], concurrent object-oriented programming languages[27, 26], and concurrent languages based on process calculi[19].

- We have implemented the proposed analysis method for a concurrent programming language *HACL* and showed that our analysis is practically efficient through several experiments, though the cost is theoretically exponential in the size of program.

- We have showed that the results of preliminary experiments on the reduced costs of operations on channels are promising.

8.2 Future Work

- Extension of the type system. As discussed in Chapter 6, we often suffer from roughness of the analysis due to the simplicity of our type system. Thus, in order to refine the analysis, extensions of the type system, including subtyping and polymorphism on use, should be considered.

- Further performance evaluation (especially in distributed environments). In distributed environments, the tail-call optimization presented in this thesis often changes behaviors of programs because the computation site may change by passing (continuation) closures. For example, in *HACL* for distributed environments, **new** r **in** $m(e_1, r) \,|\, r(x) \Rightarrow e_2$ **end** computes e_2 at the site where r is created, while its tail-call optimized form $m(e_1, \lambda x.e_2)$ computes e_2 at the site of m. Thus, performance of optimization should be evaluated through more realistic applications.

- Implementation issues on how to utilize linearity information on closures or other values. As discussed in Chapter 6, it is not straightforward to

use linearity information on values for optimization. Thus, implementation technique for linear values should be studied.

- Type system with uses for dynamically-typed languages. We consider our analysis technique can be applied to dynamically-typed languages such as Scheme by improving type systems for such languages[25]. So, future work also includes development of a type system with uses for a dynamically-typed language.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] H. G. Baker. The boyer benchmark meets linear logic. *Lisp Pointers*, 6(4):3–10, October–December 1993.

[4] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. *Software Trends*, 1:61–80, 1993.

[5] A. Igarashi, N. Kobayashi, and A. Yonezawa. Type-based analysis of usage of communication channels for concurrent programming languages. In *JSSST WOOC96 proceedings*, 1996. (in Japanese).

[6] N. Kobayashi. A partially deadlock-free process calculus (II). Technical report, Department of Information Science, University of Tokyo, 1996.

[7] N. Kobayashi, M. Nakade, and A. 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.

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

[9] N. Kobayashi and A. Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pages 31–45, 1994.

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

[11] N. Kobayashi and A. Yonezawa. Towards foundations for concurrent object-oriented programming – types and language design –. *Theory and Practice of Object Systems*, 1(4):243–268, 1995.

[12] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of ACM SIGACT / SIGPLAN Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

[13] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[14] J. C. Mitchell. Type systems for programming languages. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B*, chapter 8, pages 365–458. The MIT press/Elsevier, 1990.

[15] H. R. Nielson and F. 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.

[16] H. R. Nielson and F. 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.

[17] B. C. Pierce. Programming in the pi-calculus: An experiment in programming language design. Lecture notes for a course at the LFCS, University of Edinburgh., 1993.

[18] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, volume 907 of *LNCS*, pages 187–215. Springer-Verlag, 1995.

[19] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1995.

[20] J. 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.

[21] V. Saraswat. *Concurrent Constraint Programming.* The MIT Press, 1992.

[22] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[23] K. Taura and A. Yonezawa. Schematic: A concurrent object-oriented extension to scheme. Technical Report 95-11, Department of Information Science, University of Tokyo, December 1995.

[24] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, San Diego, California, 1995.

[25] A. K. Wright and R. Cartwright. A practical soft type system for scheme. In *Proceedings of ACM Conference on Lisp and Functional Programming*, 1994.

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

[27] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* The MIT Press, 1987. 282pages.

Appendix A

Proofs

A.1 Proof of Subject Reduction Theorem (2) (Theorem 2.4)

We need several lemmas to prove the subject reduction theorem. These lemmas are proved by straightforward induction on type derivations.

Lemma A.1 (Weakening on Uses (1)) If $\Gamma \vdash e$, then for Γ' such that $\Gamma' \geq \Gamma$, $\Gamma' \vdash e$ holds.

Lemma A.2 If $\Gamma \vdash e$ and $e \cong e'$, then $\Gamma \vdash e'$ holds.

Proof of Subject Reduction Theorem (2) We prove by induction on the proof of $\Gamma \vdash e \xrightarrow{l} e'$ with case analysis on the last rule used.

(Ru-Comm) Suppose $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash ((x(\tilde{y}) \Rightarrow e)\&e') \,|\, x(\tilde{z})$ and the following type derivation.

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \tilde{y} : \tilde{\tau} \vdash e \quad \kappa_{11} \geq 1}{\Gamma_1 + x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{21})} \vdash x(\tilde{y}) \Rightarrow e} \quad \Gamma_2 \vdash e'}{(\Gamma_1 + x : [\tilde{\tau}]^{(\kappa_{11},\kappa_{21})}) \sqcup \Gamma_2 \vdash (x(\tilde{y}) \Rightarrow e)\&e'} \quad \dfrac{\kappa_{22} \geq 1 \quad \tilde{\tau}' \geq \tilde{\tau}}{\Gamma_3 + x : [\tilde{\tau}]^{(\kappa_{12},\kappa_{22})}, \tilde{z} : \tilde{\tau}' \vdash x(\tilde{z})}}{\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash ((x(\tilde{y}) \Rightarrow e)\&e') \,|\, x(\tilde{z})}$$

where $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} = ((\Gamma_1 + x : [\tilde{\tau}_1]^{(\kappa_{11},\kappa_{21})}) \sqcup \Gamma_2) + \Gamma_3 + x : [\tilde{\tau}_1]^{(\kappa_{12},\kappa_{22})}, \tilde{z} : \tilde{\tau}'$. From $\kappa_{11} \geq 1$ and $\kappa_{22} \geq 1$, $(\kappa_1, \kappa_2) \geq (1, 1)$ holds.

Consider the case of $x \in dom(\Gamma_1)$. Let the use pair of $\Gamma_1(x)$ be (κ_1', κ_2'). First, we consider only receive uses. If $(\kappa_1, \kappa_2) = (1, 1)$, then the uses of $\Gamma_1(x)$ is equal to $(0, 0)$, that is, $(\kappa_1^-, \kappa_2^-) \geq (\kappa_1', \kappa_2')$ holds. In the other cases, $(\kappa_1^-, \kappa_2^-) \geq (\kappa_1', \kappa_2')$ is easily obtained. Therefore, $\Gamma, x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)} \geq \Gamma_1\{\tilde{z} : \tilde{\tau}'\}$ holds even if $x \notin dom(\Gamma_1)$. By Lemma A.1 and renaming of free variables $y_1, \ldots, y_n$ in e to $z_1, \ldots, z_n$, respectively, $\Gamma, x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)} \vdash [\tilde{z}/\tilde{y}]e$ holds.

(Ru-RComm) Suppose $\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa)} \vdash x^*(y_1, \ldots, y_n) \Rightarrow e \mid x(\tilde{z})$ and the following type derivation.

$$\dfrac{\dfrac{\Gamma_1', \tilde{y} : \tilde{\tau} \vdash e \quad \Gamma_1 \geq \omega \cdot \Gamma_1'}{\Gamma_1 + x : [\tilde{\tau}]^{(\omega,\kappa_1)} \vdash x^*(\tilde{y}) \Rightarrow e} \quad \dfrac{\kappa_3 \geq 1 \quad \tilde{\tau}' \geq \tilde{\tau}}{\Gamma_2 + x : [\tilde{\tau}]^{(\kappa_2,\kappa_3)}, \tilde{z} : \tilde{\tau}' \vdash x(\tilde{z})}}{\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa)} \vdash x^*(\tilde{y}) \Rightarrow e \mid x(\tilde{z})}$$

where $\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa)} = \Gamma_1 + x : [\tilde{\tau}]^{(\omega,\kappa_1)} + \Gamma_2 + x : [\tilde{\tau}]^{(\kappa_2,\kappa_3)}, \tilde{z} : \tilde{\tau}$. From $\kappa_3 \geq 1$, $\kappa \geq 1$ holds. Therefore, $\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa^-)}$ is well-defined and $\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa^-)} \geq \Gamma_1 + \Gamma_1', \tilde{z} : \tilde{\tau}' + x : [\tilde{\tau}]^{(\omega,\kappa_1)}$ holds whether $\kappa = 1$ or ω. (Note that $\Gamma_1 = \Gamma_1'' + \omega \cdot \Gamma_1'$ for some Γ_1'' and $(\omega \cdot \Gamma_1') + \Gamma_1' = \omega \cdot \Gamma_1'$.) By Lemma A.1 and renaming, we have $\Gamma_1', \tilde{z} : \tilde{\tau}' \vdash [\tilde{z}/\tilde{y}]e$. From (Tu-Par) and Lemma A.1, $\Gamma, x : [\tilde{\tau}]^{(\omega,\kappa^-)} \vdash [\tilde{z}/\tilde{y}]e \mid x^*(\tilde{y}) \Rightarrow e$ is derived.

(Ru-Cong) Follow from Lemma A.2.

(Ru-Par) Suppose $\Gamma \vdash e_1 \mid e_2$. From (Tu-Par) rule, for some Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 + \Gamma_2$, $\Gamma_1 \vdash e_1$ and $\Gamma_2 \vdash e_2$ By induction hypothesis, Γ_1^{-x} is well-defined and $\Gamma_1^{-x} \vdash e_1'$ holds. It is trivial that $(\Gamma_1 + \Gamma_2)^{-x} \geq \Gamma_1^{-x} + \Gamma_2$ holds. Hence $\Gamma^{-x} \vdash e_1' \mid e_2$ is derived from Lemma A.1.

(Ru-New1) Suppose $\Gamma \vdash \mathbf{new}\ x^{(\kappa_1,\kappa_2)}\ \mathbf{in}\ e\ \mathbf{end}$. From (Tu-New) rule, $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash e$. By induction hypothesis, $\Gamma, x : [\tilde{\tau}]^{(\kappa_1^-,\kappa_2^-)} \vdash e'$ holds. Therefore, $\Gamma (= \Gamma^{-\varepsilon}) \vdash \mathbf{new}\ x^{(\kappa_1^-,\kappa_2^-)}\ \mathbf{in}\ e'\ \mathbf{end}$.

(Ru-New2) Suppose $\Gamma \vdash$ **new** $x^{(\kappa_1,\kappa_2)}$ **in** e **end**. From (Tu-New) rule, $\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)} \vdash e$. By induction hypothesis, $(\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})^{-l} \vdash e'$ holds. Because $l \neq x$, $(\Gamma, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)})^{-l} = \Gamma^{-l}, x : [\tilde{\tau}]^{(\kappa_1,\kappa_2)}$ and $\Gamma^{-l} \vdash$ **new** $x^{(\kappa_1,\kappa_2)}$ **in** e' **end** holds.

□

A.2 Proof of Theorem 3.3

To prove Theorem 3.3, we need the following lemmas on the subprocedures $\bigoplus, \bigodot$, and $\bigsqcup$.

Lemma A.3 (function $\bigoplus$ (1)) Given $\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\}$, suppose that $(S', \Gamma', \Gamma'_1, \ldots, \Gamma'_n, \Theta')$ satisfies $\Gamma' \geq_{\Theta'} \Gamma'_1 + \ldots + \Gamma'_n$ and that for all i: (1) $\Gamma'_i \supseteq S'\Gamma_i$, and (2) $\Theta' \models S'\Theta_i$. Then, $\bigoplus(\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\})$ succeeds without failure and outputs (S, Γ, Θ) which satisfies the following conditions: (1) $\Gamma \geq_\Theta S\Gamma_1 + \ldots + S\Gamma_n$, (2) $\forall i.\Theta \models S\Theta_i$, (3) there exists a substitution S'' such that $S' = S''S$, $\Gamma' \supseteq S''\Gamma$, and $\Theta' \models S''\Theta$.

Lemma A.4 (function $\bigodot$ (1)) Given (Γ, Θ, κ), suppose that $(S, \Gamma', \Gamma'', \Theta')$ satisfies: (1) $\Gamma'' \geq_{\Theta'} \kappa \cdot \Gamma'$, (2) $\Gamma' \supseteq S\Gamma(x)$, and (3) $\Theta' \models S'\Theta$. Then, $\bigodot(\Gamma, \Theta, \kappa)$ succeeds without failure and outputs (Γ_1, Θ_1) which satisfies the following conditions: (1) $\Gamma_1 \geq_{\Theta_1} \kappa \cdot \Gamma$, (2) $\Theta_1 \models \Theta$, and (3) $\Gamma'' \supseteq S\Gamma_1(x)$.

Lemma A.5 (function $\bigsqcup$ (1)) Given $\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\}$, suppose that $(S', \Gamma', \Gamma'_1, \ldots, \Gamma'_n, \Theta')$ satisfies $\Gamma' \geq_{\Theta'} \Gamma'_1 \sqcup \ldots \sqcup \Gamma'_n$ and that for i: (1) $\Gamma'_i \supseteq S'\Gamma_i$, and (2) $\Theta' \models S'\Theta_i$. Then, $\bigsqcup(\{(\Gamma_1, \Theta_1), \ldots (\Gamma_n, \Theta_n)\}$ succeeds without failure and outputs (S, Γ, Θ) which satisfies the following conditions: (1) $\Gamma \geq_\Theta S\Gamma_1 \sqcup \ldots \sqcup S\Gamma_n$, (2) $\forall i.\Theta \models S\Theta_i$, and (3) there exists a substitution S'' such that $S' = S''S$, $\Gamma' \supseteq S''\Gamma(x)$, and $\Theta' \models S''\Theta$.

These lemmas are trivial from their definitions and Theorem 3.2.

Proof of Theorem 3.3 Structural induction on the proof of $\Gamma, \Theta \vdash_{\mathcal{TRU}} e$ with case anlysis on the last rule used.

(TRU-SEND) Suppose that $\Gamma', \Theta' \vdash_{\mathcal{TRU}} x(y_1, \ldots, y_n)$. Let $(\Gamma, \Theta, x(y_1, \ldots, y_n))$ be $PTU(x(y_1, \ldots, y_n))$. It is trivial that $\Gamma, \Theta \vdash_{\mathcal{TRU}} x(y_1, \ldots, y_n)$. From (TRU-SEND) rule, we have $\Gamma'(x) = [\rho_1^{(\kappa_{11}, \kappa_{12})}, \ldots, \rho_n^{(\kappa_{n1}, \kappa_{n2})}]$, $\Gamma'(y_i) = \rho_i^{(\kappa'_{i1}, \kappa'_{i2})}$ such that $(\kappa'_{i1}, \kappa'_{i2}) \geq_\Theta (\kappa_{i1}, \kappa_{i2})$ 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 $S_\kappa = [\kappa_{11}/j_{11}, \ldots, \kappa_{n2}/j_{n2}, \kappa'_{11}/k_{11}, \ldots, \kappa'_{n2}/k_{n2}]$. It satisfies the second condition of the principal typing.

(TRU-PAR) Suppose that $\Gamma', \Theta' \vdash_{\mathcal{TRU}} e_1 \,|\, e_2$. From (TRU-PAR) rule, we have $\Gamma'_1, \Theta' \vdash_{\mathcal{TRU}} e_1$, $\Gamma'_2, \Theta' \vdash_{\mathcal{TRU}} e_2$ and $\Gamma' \geq_\Theta \Gamma'_1 + \Gamma'_2$. By induction hypothesis, $PTU(e_1)$ and $PTU(e_2)$ succeeds and outputs $(\Gamma_1, \Theta_1, e_1)$ and $(\Gamma_2, \Theta_2, e_2)$, respectively. Moreover, for each results, there exist substitutions S_1 and S_2 which satisfy the second condition of Definition 3.5. Let $S' = S_1 S_2$. S' satisfies the assumption of Lemma A.3. Therefore, $\bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\})$ succeeds and outputs (S, Γ, Θ) such that $\Gamma \geq_\Theta S\Gamma_1 + S\Gamma_2$, $\Theta \models S\Theta_i$ for $i = 1, 2$. By Lemma 3.4, 3.3, for $i = 1, 2$, we obtain $S\Gamma_i, \Theta \vdash_{\mathcal{TRU}} e_i$. Therefore, $\Gamma, \Theta \vdash_{\mathcal{TRU}} e_1 \,|\, e_2$ is derived from (TRU-PAR) rule.

By Lemma A.3, there exists S'' such that $S' = S''S$, $\Gamma' \supseteq S''\Gamma$, and $\Theta' \models S''\Theta$.

(TRU-CHOICE) Similar to the case of (TRU-PAR) rule.

(TRU-RECV) Suppose that $\Gamma' + x : [\rho_1^{(\kappa'_{11}, \kappa'_{12})}, \ldots, \rho_n^{(\kappa'_{n1}, \kappa'_{n2})}]^{(\kappa'_1, \kappa'_2)}, \Theta' \vdash_{\mathcal{TRU}} x(y_1, \ldots, y_n) \Rightarrow e$. By (TRU-RECV) rule, we obtain

1. $\Gamma'', y_1 : \rho_1^{(\kappa'_{11}, \kappa'_{12})}, ldots, y_n : \rho_n^{(\kappa_{n1}, \kappa_{n2})}, \Theta' \vdash_{\mathcal{TRU}} e$
2. $\kappa'_{x1} \geq_{\Theta'} 1$
3. $\Gamma' \geq_{\Theta'} \Gamma''$

By induction hypothesis, $PTU(e)$ succeeds without failure and outputs (Γ, Θ, e). Let S' be the substitution referred in the second condition of Definition 3.5 and $S = S'([\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n], [\kappa'_1/j, \kappa'_{11}/j_1, \ldots, \kappa'_{1n}/j_n, \kappa'_{12}/k_1, \ldots, \kappa'_{n2}/k_n])$.

By Lemma A.3, $\bigoplus(\{(x : [\alpha_1^{(j_1,k_1)}, \ldots, \alpha_n^{(j_n,k_n)}]^{(j,k)}, \{j \geq 1\}), (\Gamma \setminus (y_i : \tau'_i | y_i \in dom(\Gamma)), \Theta)\})$ succeeds and outputs $(S_1, \Gamma_1, \Theta_1)$ such that: 1) $\Gamma_1 \geq_{\Theta_1} S_1(x : [\alpha_1^{(j_1,k_1)}, \ldots, \alpha_n^{(j_n,k_n)}]^{(j,k)}) + S_1(\Gamma \setminus (y_i : \tau'_i | y_i \in dom(\Gamma)))$, 2) there exists S_2 such that $S = S_2 S_1$, $\Gamma' + x : [\tilde{\tau}]^{(\kappa'_1,\kappa'_2)} \supseteq S_2\Gamma_1$, and $\Theta' \models S_2\Theta_1$.

By Lemma 3.2, $U(\{(S_1\alpha_i^{(j_i,k_i)}, S_1\tau_i) | y_i \in dom(\Gamma)\})$ outputs its most general unifier S_3 and there exists S_4 such that $S_2 = S_4 S_3$.

Now, we have $S_3\Gamma_1, S_3\Theta_1 \vdash_{\mathcal{TRU}} e$ by Lemma 3.4, 3.3, $S_3S_1 j \geq_{\Theta_1} 1$, and $S_3\Gamma \geq_{\Theta_1} S_3\Gamma$. Therefore, $S_3\Gamma_1, S_3\Theta_1 \vdash_{\mathcal{TRU}} x(\tilde{y}) \Rightarrow e$. S_4 is the substitution which satisfies the second condition of the principal typing.

(Tru-RRecv) Similar to the case of (Tru-Recv).

(Tru-New) Suppose that $\Gamma', \Theta' \vdash_{\mathcal{TRU}}$ **new** $x^{(\kappa_1,\kappa_2)}$ **in** e **end**. From (Tru-New) rule, we have $\Gamma'', x : [\tilde{\tau}']^{(\kappa_1,\kappa_2)}, \Theta' \vdash_{\mathcal{TRU}} e$. By induction hypothesis, $PTU(e)$ succeeds without failure and outputs (Γ, Θ, e). Let S be the substitution in Definition 3.5.

Consider the case of $x \in dom(\Gamma)$. By Theorem 4.2, $U(\Gamma(x), \alpha^{(\kappa_1,\kappa_2)})$ succeeds and outputs S'. Therefore, $S'(\Gamma \setminus x : \tau_x), S'\Theta \vdash_{\mathcal{TRU}}$ **new** $x^{(\kappa_1,\kappa_2)}$ **in** e **end** is derived. By the property of the most general unifier, $S = S''S'$, $\Gamma' \supseteq S''S'(\Gamma \setminus x : \tau_x)$ and $\Theta' \models S''(S'\Theta)$.

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

□

A.3 Proof of Theorem 4.3

To prove Theorem 4.3, we need to strengthen the theorem so that induction works. Theorem 4.3 is then obtained as a special case of the strengthened theorem (Theorem A.1).

We give several preliminary definitions before stating the strengthened theorem.

Definition A.1 A type scheme $\forall\alpha_1 \ldots \forall\alpha_n.\tau_1$ is *more general* than $\forall\beta_1 \ldots \forall\beta_m.\tau_2$ if $[\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]\tau_1 = \tau_2$ for some $\rho_1, \ldots, \rho_n$.

Intuitively, a type scheme σ is more general than σ' if every simple type obtained from σ' is also obtained from σ.

The following condition "Γ respects As" means that all the non-simple types in Γ are less general than those in As.

Definition A.2 Γ *respects* Δ if there exists a substitution S such that for all $x \in dom(\Gamma)$, if $\Gamma(x)$ is a type scheme $\forall\alpha_1 \ldots \forall\alpha_n.\rho_1^{(\kappa_1,\kappa_2)} (n \geq 1)$, then:

1. $x \in dom(\Delta)$
2. if $\Delta(x) = \forall\beta_1 \ldots \forall\beta_m.\rho_2^{(\kappa_3,\kappa_4)}$, $S\forall\beta_1 \ldots \forall\beta_m.\rho_2^{(\kappa_1,\kappa_2)}$ is more general than $\Gamma(x)$.

Now, we state Theorem A.1. When $PTU(e, \emptyset)$ is invoked, it recursively calls itself for each subterm of e, with the second argument As being augmented with bindings of let-variables to type schemes. So, in order to make induction steps work, intuitively we want to prove that, if As gives most general possible type schemes for variables bound by outer **let** expressions, then $PTU(e, As)$ gives a most general typing among all the valid typings. By the condition "As gives most general possible type schemes for variables bound by outer **let** expressions," it suffices to show that $PTU(e, As)$ is most general only among typings whose type environment "respects" As. Note that Theorem 4.3 is obtained as a special case since "Γ respects $\emptyset$" if and only if "Γ contains only simple types."

Theorem A.1 Suppose that $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} e : \tau'$ and Γ' respects As. Then, $PTU(e, As)$ succeeds. Moreover, let $(\Gamma, \Theta, e, \tau) = PTU(e, As)$. The quadruple satisfies the following three conditions:

1. $\Gamma, \Theta \vdash_{\mathcal{ETRU}} e : \tau$
2. If $\Gamma(x)$ is $\forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1,\kappa_2)}$ $(n \geq 1)$, then $x \in dom(As)$ and there exists a substitution S' whose domain does not includes any free (type/use) variable in $(\Gamma, \Theta, e, \tau)$ such that $S'\forall\alpha_1 \ldots \forall\alpha_n.\rho'^{(\kappa_1,\kappa_2)} = \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1,\kappa_2)}$ where $As(x) = \forall\alpha_1 \ldots \forall\alpha_n.\rho'^{(\kappa'_1,\kappa'_2)}$.
3. There exists a substitution S such that
   (a) $S\Gamma(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma)$.
   (b) $\Theta' \models S\Theta$
   (c) $\tau' \equiv S\tau$

The following lemmas for the subprocedures are obtained from Lemmas A.3–A.5 by replacing the statement like $\Gamma' \supseteq S\Gamma$ with "$S\Gamma(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma)$."

Lemma A.6 (function $\oplus$ (2)) Given $\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\}$, suppose that $(S', \Gamma', \Gamma'_1, \ldots, \Gamma'_n, \Theta')$ satisfies $\Gamma' \geq_{\Theta'} \Gamma'_1 + \ldots + \Gamma'_n$ and that for all i: (1) $S'\Gamma_i(x)$ is more general than $\Gamma'_i(x)$ for all $x \in dom(\Gamma_i)$, (2) $\Theta' \models S'\Theta_i$. Then, $\bigoplus(\{(\Gamma_1, \Theta_1), \ldots, (\Gamma_n, \Theta_n)\})$ succeeds without failure and outputs (S, Γ, Θ) which satisfies the following conditions: (1) $\Gamma \geq_\Theta S\Gamma_1 + \ldots + S\Gamma_n$, (2) $\forall i.\Theta \models S\Theta_i$, (3) there exists a substitution S'' such that

1. $S' = S''S$
2. $S''\Gamma(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma)$.
3. $\Theta' \models S''\Theta$

Lemma A.7 (function $\odot$ (2)) Given (Γ, Θ, κ), suppose that $(S, \Gamma', \Gamma'', \Theta')$ satisfies: (1) $\Gamma'' \geq_{\Theta'} \kappa \cdot \Gamma'$, (2) $S\Gamma(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma)$, and (3) $\Theta' \models S'\Theta$. Then, $\odot(\Gamma, \Theta, \kappa)$ succeeds without failure and outputs (Γ_1, Θ_1) which satisfies the following conditions: (1) $\Gamma_1 \geq_{\Theta_1} \kappa \cdot \Gamma$, (2) $\Theta_1 \models \Theta$, and (3) $S\Gamma_1(x)$ is more general than $\Gamma''(x)$ for all $x \in dom(\Gamma_1)$.

Lemma A.8 (function $\sqcup$ (2)) Given $(\Gamma_1, \Theta_1, \Gamma_2, \Theta_2)$, suppose that $(S', \Gamma', \Gamma'_1, \Gamma'_2, \Theta')$ satisfies $\Gamma' \geq_{\Theta'} \Gamma'_1 \sqcup \Gamma'_2$ and that for $i = 1, 2$: (1) $S'\Gamma_i(x)$ is more general than $\Gamma'_i(x)$ for all $x \in dom(\Gamma_i)$, (2) $\Theta' \models S'\Theta_i$. Then, $\bigsqcup(\Gamma_1, \Theta_1, \Gamma_2, \Theta_2)$ succeeds without failure and outputs (S, Γ, Θ) which satisfies the following conditions: (1) $\Gamma \geq_\Theta S\Gamma_1 \sqcup S\Gamma_2$, (2) $\forall i.\Theta \models S\Theta_i$, and (3) there exists a substitution S'' such that

1. $S' = S''S$
2. $S''\Gamma(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma)$.
3. $\Theta' \models S''\Theta$

Proof of Theorem A.1 We prove by induction on the structure of the proof of $\Gamma, \Theta \vdash_{\mathcal{ETRU}} e : \tau$ with case analysis on the last rule used. We show only the several main cases.

(ETru-Var) Suppose that $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} x : \rho'^{(\kappa_3, \kappa_4)}$ and that Γ' respects As. We have two cases.

- $x \in dom(As)$. By (ETru-Var) rule, we have $[\rho'_1/\beta_1, \ldots, \rho'_m/\beta_m]\rho''^{(\kappa_1, \kappa_2)} \geq_{\Theta'} \rho'^{(\kappa_3, \kappa_4)}$ where $\Gamma'(x) = \forall\beta_1 \ldots \forall\beta_m.\rho''^{(\kappa_1, \kappa_2)}$.

  Let S_1 be $[\rho'_1/\beta_1, \ldots, \rho'_m/\beta_m]$. Because Γ' respects As, there exists S_2 such that $S_2\forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1, \kappa_2)}$ is more general than $\Gamma'(x)$, that is, for some $\rho_1, \ldots, \rho_n$, $[\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]S_2\rho^{(\kappa_1, \kappa_2)} = \rho''^{(\kappa_1, \kappa_2)}$ where $As(x) = \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa'_1, \kappa'_2)}$. Let $S_3 = [\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]$, $\Gamma = x : \forall\alpha_1 \ldots \forall\alpha_n.\rho^{(j,k)}$, $\Theta = \{j \geq l, k \geq m\}$ and $S_4 = [\gamma_1/\alpha_1, \ldots, \gamma_n/\alpha_n]$

where j, k, l, m are fresh use variables, and the γ_i's are fresh type variables whose kinds (usual or non-O) are the same as those of the α_i's, respectively. For $(\Gamma, \Theta, x, S_4\rho^{(l,m)})$, we have $\Gamma, \Theta \vdash_{\mathcal{ETRU}} x : S_4\rho^{(l,m)}$. It is trivial that the second condition in the theorem is satisfied. Moreover, $S(= S_1S_3S_2[\alpha_1/\gamma_1, \ldots, \alpha_n/\gamma_n, \kappa_1/j, \kappa_2/k, \kappa_3/l, \kappa_4/m])$ satisfies the following conditions:

1. $S\forall\alpha_1 \ldots \forall\alpha_n.\rho^{(j,k)}$ is more general than $\Gamma'(x)$ because $S\forall\alpha_1 \ldots \forall\alpha_n.\rho^{(j,k)} = S_2\forall\alpha_1 \ldots \forall\alpha_n.\rho^{(\kappa_1,\kappa_2)}$.
2. $\Theta' \models S\Theta$
3. $SS_4\rho^{(l,m)} \equiv S_1S_3S_2\rho^{(\kappa_3,\kappa_4)} \equiv \rho'^{(\kappa_3,\kappa_4)}$

- $x \notin dom(As)$. In this case, $\Gamma'(x)$ must be a simple type and the type judgement can be written as $(\Gamma'', x : \rho'^{(\kappa''_1,\kappa''_2)}), \Theta' \vdash_{\mathcal{ETRU}} x : \rho'^{(\kappa'_1,\kappa'_2)}$ where $(\kappa''_1, \kappa''_2) \geq_{\Theta'} (\kappa'_1, \kappa'_2)$. $PTU(x, As)$ always succeeds and outputs $(\Gamma, \Theta, x, \dot{\alpha}^{(l,m)})$ where $\Gamma = x : \dot{\alpha}^{(j,k)}$ and $\Theta = \{j \geq l, k \geq m\}$ with use fresh variables j, k, l, m. We have $x : \dot{\alpha}^{(j,k)}, \Theta \vdash_{\mathcal{ETRU}} \dot{\alpha}^{(l,m)}$ and there exists $S = [\kappa''_1/j, \kappa''_2/k, \kappa'_1/l, \kappa'_2/m, \rho'_1/\dot{\alpha}]$ such that
  1. $\rho'^{(\kappa'_1,\kappa'_2)} \equiv S\dot{\alpha}^{(j,k)}$ i.e., $S\Gamma(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma)$.
  2. $\Theta' \models S\Theta$

(ETru-App/Send) Suppose that $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} e_1\ e_2 : \rho^{(\kappa_3,\kappa_4)}$ and that Γ' respects As. By (ETru-App/Send), we have $\Gamma'_1, \Theta' \vdash_{\mathcal{ETRU}} e_1 : \rho_1^{(\kappa_5,\kappa_6)} \xrightarrow{(\kappa_1,\kappa_2)} \rho^{(\kappa_3,\kappa_4)}$, $\Gamma'_2, \Theta' \vdash_{\mathcal{ETRU}} e_2 : \rho_1^{(\kappa_5,\kappa_6)}$, $\kappa_2 \geq_{\Theta'} 1$, $\rho_1^{(\kappa_5,\kappa_6)} \not\simeq O$ and $\Gamma' \geq_{\Theta'} \Gamma'_1 + \Gamma'_2$. Both Γ'_1 and Γ'_2 respect As. By induction hypothesis, $PTU(e_1, As)$ succeeds and the outputted quadruple $(\Gamma_1, \Theta_1, e_1, \tau_1)$ satisfies the three conditions of the theorem. In particular, $\Gamma_1, \Theta_1 \vdash_{\mathcal{ETRU}} e_1 : \tau_1$ holds and there exists a substitution S_1 such that:

1. $S_1\Gamma_1(x)$ is more general than $\Gamma'_1(x)$ for all $x \in dom(\Gamma_1)$.

2. $\Theta' \models S_1\Theta_1$
3. $\rho_1^{(\kappa_5,\kappa_6)} \xrightarrow{(\kappa_1,\kappa_2)} \rho^{(\kappa_3,\kappa_4)} \equiv S_1\tau_1$

Similary, $PTU(e_2, As)$ succeeds and the outputted quadruple $(\Gamma_2, \Theta_2, e_2, \tau_2)$ satisfies the three conditions of the theorem. In particular $\Gamma_2, \Theta_2 \vdash_{\mathcal{ETRU}} e_2 : \tau_2$ holds and there exists a substitution S_2 such that:

1. $S_2\Gamma_2(x)$ is more general than $\Gamma_2'(x)$ for all $x \in dom(\Gamma_2)$.
2. $\Theta' \models S_2\Theta_2$
3. $\rho_1^{(\kappa_5,\kappa_6)} \equiv S_2\tau_2$

Let $S = S_1S_2[\kappa_1/j, \kappa_2/k, \kappa_3/l, \kappa_4/m, \kappa_5/n, \kappa_6/p, \rho/\dot{\delta}, \rho_1/\bar{\gamma}]$ where j, k, l, m, n, p are fresh use variables and $\dot{\delta}, \bar{\gamma}$ are fresh type variables. By Lemma A.6, $\bigoplus(\{(\Gamma_1, \Theta_1), (\Gamma_2, \Theta_2)\})$ outputs $(S_3, \Gamma_3, \Theta_3)$ without failure. The triple satisfies the following conditions:

1. $\Gamma_3 \geq_{\Theta_3} S_3\Gamma_1 + S_3\Gamma_2$, $\Theta_3 \models S_3\Theta_1$ and $\Theta_3 \models S_3\Theta_2$ hold.
2. There exists a substitution S_4 such that
   (a) $S = S_4S_3$
   (b) $S_4\Gamma_3(x)$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma_3)$.
   (c) $\Theta' \models S_4\Theta_3$

By Theorem 4.2, $U(\{(S_3\tau_1, (S_3\tau_2) \xrightarrow{(j,k)} \dot{\delta}^{(l,m)}), (S_3\tau_2, \bar{\gamma}^{(n,p)})\})$ succeeds without failure and outputs S_5 which satisfies $S_5(S_3\tau_1) \equiv S_5((S_3\tau_2) \xrightarrow{(j,k)} \dot{\delta}^{(l,m)}), S_5(S_3\tau_2) \equiv S_5\bar{\gamma}^{(n,p)}$ and $\exists S_6.(S_4 = S_6S_5)$.

For $\Theta_5 = S_5(\Theta_3 \cup \{k \geq 1\})$, now we have $S_5S_3\Gamma_1, \Theta_5 \vdash_{\mathcal{ETRU}} e_1 : S_5S_3\tau_1$, $S_5S_3\Gamma_2, \Theta_5 \vdash_{\mathcal{ETRU}} e_2 : S_5S_3\tau_2$, $S_5S_3\tau_2 (= S_5\bar{\gamma}^{(n,p)}) \not\equiv O$, $S_5k \geq_{\Theta_5} 1$, $S_5\Gamma_3 \geq_{\Theta_5} S_5S_3\Gamma_1 + S_5S_3\Gamma_2$ and $S_5(S_3\tau_1) \equiv S_5((S_3\tau_2) \xrightarrow{(j,k)} \dot{\delta}^{(l,m)})$. By (ETRU-APP/SEND), we have $S_5\Gamma_3, \Theta_5 \vdash_{\mathcal{ETRU}} e_1e_2 : S_5\dot{\delta}^{(l,m)}$. It is trivial that the

second condition of the theorem is satisfied for $S_5\Gamma_3$ and As. We have also S_6 such that

1. $S_6S_5\Gamma_3(x)(= S_4\Gamma_3(x))$ is more general than $\Gamma'(x)$ for all $x \in dom(\Gamma_3)$.
2. $\Theta' \models S_6\Theta_5$
3. $\rho^{(\kappa_3,\kappa_4)} \equiv S_6S_5\dot{\delta}^{(l,m)}$

(ETru-Abs) Suppose that $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} (\lambda x.e')^{\kappa'} : \rho_1^{\prime(\kappa_1',\kappa_2')} \xrightarrow{(0,\kappa')} \tau_2'$ and that Γ' respects As. By (ETru-Abs), we have $\Gamma'', x : \rho_1^{\prime(\kappa_1',\kappa_2')}, \Theta' \vdash_{\mathcal{ETRU}} e' : \tau_2'$ and $\Gamma' \geq_{\Theta'} \kappa' \cdot \Gamma''$. It is triviall that $\Gamma'', x : \rho_1^{\prime(\kappa_1',\kappa_2')}$ also respects As. By induction hypothesis, $PTU(e', As)$ succeeds and the outputted quadruple $(\Gamma_1, \Theta_1, e', \tau_2)$ satisfies the three conditions. In particular, $\Gamma_1, \Theta_1 \vdash_{\mathcal{ETRU}} e' : \tau_2$ holds and there exists a substitution S_1 such that:

1. $S_1\Gamma_1(y)$ is more general than $(\Gamma'', x : \rho_1^{\prime(\kappa_1',\kappa_2')})(y)$ for all $y \in dom(\Gamma_2)$.
2. $\Theta' \models S_1\Theta_1$
3. $\tau_2' \equiv S_1\tau_2$

We have two cases here.

- $x \in dom(\Gamma_1)$. Let $S = S_1$. By Lemma A.7, $\bigodot(\Gamma_1 \setminus x : \tau_1, \Theta_1, \kappa')$ outputs (Γ_2, Θ_2) with no failure. This pair satisfies the following conditions:

  1. $\Gamma_2 \geq_{\Theta_2} j \cdot (\Gamma_1 \setminus x : \tau_1)$
  2. $\Theta_2 \models \Theta_1$
  3. $S\Gamma_2(y)$ is more general than $\Gamma'(y)$ for all $y \in dom(\Gamma_2)$.
  4. $\Theta' \models S\Theta_2$

  By (ETru-Abs), we have $\Gamma_2, \Theta_2 \vdash_{\mathcal{ETRU}} (\lambda x.e')^{\kappa'} : \tau_1 \xrightarrow{(0,\kappa')} \tau_2$. Γ_2 and As satisfies the second condition of the theorem. We have, S which satisfies the third condition of the theorem.

- $x \notin dom(\Gamma_1)$. Let $S = S_1[\kappa_1'/j, \kappa_2'/k, \rho_1'/\dot{\alpha}]$ with a fresh $\dot{\alpha}, j$, and k. By Lemma A.7, $\odot(\Gamma_1, \Theta_1, \kappa')$ outputs (Γ_2, Θ_2) with no failure. This pair satisfies the following conditions:

  1. $\Gamma_2 \geq_{\Theta_2} \kappa' \cdot \Gamma_1$
  2. $\Theta_2 \models \Theta_1$
  3. $S\Gamma_2(y)$ is more general than $\Gamma'(y)$ for all $y \in dom(\Gamma_2)$.
  4. $\Theta' \models S\Theta_2$

  By (ETru-Abs), we have $\Gamma_2, \Theta_2 \vdash_{\mathcal{ETRU}} (\lambda x.e')^{\kappa'} : \dot{\alpha}^{(j,k)} \xrightarrow{(0,l)} \tau_2$. Γ_2 and As satisfies the second condition of the theorem. We have S, which satisfies the third condition of the theorem.

(ETru-Let) Given e and As, suppose that $\Gamma', \Theta' \vdash_{\mathcal{ETRU}} \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau'$ and that Γ' respects As. By (ETru-Let), we have $\Gamma_1', \Theta' \vdash_{\mathcal{ETRU}} e_1 : \rho_1^{\prime(\kappa_1', \kappa_2')}$, $(\Gamma_2', x : clos(\Gamma_1', \rho_1^{\prime(\kappa_1', \kappa_2')}))$, $\Theta' \vdash_{\mathcal{ETRU}} e_2 : \tau'$, $\rho_1^{\prime(\kappa_1', \kappa_2')} \not\equiv O$ and $\Gamma' \geq_{\Theta'} \Gamma_1' + \Gamma_2'$. Γ_1' also respects As. By induction hypothesis, $PTU(e_1, As)$ succeeds and outputs a quadruple $(\Gamma_1, \Theta_1, e_1, \rho_1^{(\kappa_1, \kappa_2)})$ satisfies the three conditions of the theorem. In particular, $\Gamma_1, \Theta_1 \vdash_{\mathcal{ETRU}} e_1 : \rho_1^{(\kappa_1, \kappa_2)}$ holds and there exists a substitution S_1 such that:

1. $S_1\Gamma_1(y)$ is more general than $\Gamma_1'(y)$ for all $y \in dom(\Gamma_1)$.
2. $\Theta' \models S_1\Theta_1$
3. $\rho_1^{\prime(\kappa_1', \kappa_2')} \equiv S_1\rho_1^{(\kappa_1, \kappa_2)}$

Because $\rho_1^{\prime(\kappa_1', \kappa_2')} \equiv S_1\rho_1^{(\kappa_1, \kappa_2)}$, $\Gamma_2', x : clos(\Gamma_1', \rho_1^{\prime(\kappa_1', \kappa_2')})$ respects $As' (= As \cup \{x : clos(\Gamma_1, \rho_1^{(\kappa_1, \kappa_2)})\})$. By induction hypothesis, $PTU(e_2, As')$ succeeds and outputs a quadruple $(\Gamma_2, \Theta_2, e_2, \tau_2)$ which satisfies the three conditions of the theorem. In particular, $\Gamma_2, \Theta_2 \vdash_{\mathcal{ETRU}} e_2 : \tau_2$ holds, and there exists a substitution S_2 such that:

1. $S_2\Gamma_2(y)$ is more general than $(\Gamma'_2, x : clos(\Gamma'_1, \rho_1^{\prime(\kappa'_1,\kappa'_2)}))(y)$ for all $y \in dom(\Gamma_2)$.
2. $\Theta' \models S_2\Theta_2$
3. $\tau' \equiv S_2\tau_2$

We have two cases here.

- $x \in dom(\Gamma_2)$. For $(\Gamma_2, \Theta_2, e_2, \tau_2)$, from the second conditon of the theorem, there exists a substitution S' such that $\Gamma_2(x) \equiv= S'(clos(\Gamma'_1, \rho_1^{\prime(\kappa_1,\kappa_2)}))$ Let $S = S_1S_2S'[\rho'_1/\bar{\alpha}, \kappa'_1/j, \kappa'_2/k]$. By Lemma A.6, $\bigoplus(\{(\Gamma_1, \Theta_1), ((\Gamma_2 \setminus x : \forall\alpha_1 \ldots \forall\alpha_n.\rho_x^{(\kappa_{x1},\kappa_{x2})}), \Theta_2)\})$ outputs $(S_3, \Gamma_3, \Theta_3)$ without failure. The triple satisfies the following conditions:

  1. $\Gamma_3 \geq_{\Theta_3} S_3\Gamma_1 + S_3(\Gamma_2 \setminus x : \forall\alpha_1 \ldots \forall\alpha_n.\rho_x^{(\kappa_{x1},\kappa_{x2})})$, $\Theta_3 \models S_3\Theta_1$ and $\Theta_3 \models S_3\Theta_2$ hold.
  2. There exists a substitution S_4 such that
     (a) $S = S_4S_3$
     (b) $S_4\Gamma_3(y)$ is more general than $\Gamma'(y)$ for all $y \in dom(\Gamma_3)$.
     (c) $\Theta' \models S_4\Theta_3$

  By Theorem 4.2, $U(\{(\bar{\alpha}^{(j,k)}, S_3\rho_1^{\prime(\kappa'_1,\kappa'_2)}), (S_3clos(\Gamma'_1, \rho_1^{(\kappa_1,\kappa_2)}), S_3\Gamma_2(x))\})$ succeeds without failure and outputs (S_5, Θ_5) which satisfies $S_5(\bar{\alpha}^{(j,k)}) \equiv S_5(S_3\rho_1^{\prime(\kappa'_1,\kappa'_2)})$, $S_5(S_3clos(\Gamma'_1, \rho_1^{(\kappa_1,\kappa_2)})) \equiv S_5(S_3\Gamma_2(x))$, and $\exists S_6.(S_4 = S_6S_5)$.

  Now we have $S_5S_3\Gamma_1, S_5\Theta_3 \vdash_{\mathcal{ETRU}} e_1 : S_5S_3\rho_1^{(\kappa_1,\kappa_2)}, S_5S_3\Gamma_2, S_5\Theta_3 \vdash_{\mathcal{ETRU}} e_2 : S_5S_3\tau_2$. $S_5S_3\rho_1^{(\kappa_1,\kappa_2)} (= S_5\bar{\alpha}^{(S_3\kappa_1,S_3\kappa_2)}) \not\simeq O$ and $S_5\Gamma_3 \geq_{S_5\Theta_3} S_5S_3\Gamma_1 + S_5S_3\Gamma_2$. By (ETru-Let), we have $S_5\Gamma_3, S_5\Theta_3 \vdash_{\mathcal{ETRU}}$ **let** $x = e_1$ **in** e_2 **end** : $S_5S_3\tau_2$. It is trivial that the second condition of the theorem is satisfied for $S_5\Gamma_3$ and As. We have S_6 such that

1. $S_6S_5\Gamma_3(y)(= S_4\Gamma_3(y))$ is more general than $\Gamma'(y)$ for all $y \in dom(\Gamma_3)$.
2. $\Theta' \models S_6\Theta_5$
3. $\tau'_2 \equiv S_6(S_5S_3\tau_2)$

- $x \notin dom(\Gamma_2)$. We can discuss similarly by replacing $\Gamma_2 \setminus x : \forall\alpha_1 \ldots \forall\alpha_n.\rho_x^{(\kappa_{x1},\kappa_{x2})}$ with Γ_2.

□

Proof of Theorem 4.3 Because the range of Γ' is only types, Γ' respects $\emptyset$. So, $PTU(e,\emptyset)$ succeeds. Let $(\Gamma,\Theta,e,\tau) = PTU(e,\emptyset)$. We have no x such that $\Gamma(x) = \forall\alpha.\sigma$ because $As = \emptyset$. So, the range of Γ is only types. $\Gamma,\Theta \vdash_{\mathcal{ETRU}} e : \tau$ holds and there exists a substitution S such that $\Theta' \models S\Theta$, $\Gamma' \supseteq S\Gamma$ and $\tau' \equiv S\tau$ from Theorem A.1. □

A.4 Proof of Subject Reduction Theorem (3) (Theorem 4.4)

We need several lemmas to prove the subject reduction theorem. Lemma A.13 is especially important because it ensures that the well-typedness of `letrec` H `in` e is preserved if a reduction is derived from $\longrightarrow_\lambda$.

Lemma A.9 (Weakening on Uses (2)) If $\Gamma \vdash_{\mathcal{ETU}} e : \tau$, then for Γ' such that $\Gamma' \geq \Gamma$, $\Gamma' \vdash_{\mathcal{ETU}} e : \tau$.

Proof Structural induction on the proof of $\Gamma \vdash_{\mathcal{ETU}} e : \tau$. □

Lemma A.10 (Term Substitution) Given τ_y, if (1) $\Gamma + y : \forall\beta_1 \ldots \forall\beta_n.\tau_y$ is well-defined, (2) $\Gamma, z : \tau_z \vdash_{\mathcal{ETU}} e : \tau$ holds, and (3) there exists a well-defined substitution $S_y(= [\rho_1/\beta_1, \ldots, \rho_n/\beta_n])$ s.t. $S_y\tau_y \geq \tau_z$, then $\Gamma + y : \forall\beta_1 \ldots \forall\beta_n.\tau_y \vdash_{\mathcal{ETU}} [y/z]e : \tau$ holds.

Proof We prove by induction on the structure of the proof of $\Gamma, z : \tau_z \vdash_{\mathcal{ETU}} e : \tau$ with case analysis on the last rule used.

(ETu-Var) There are two cases. If $e = z$, the last proof step is

$$\frac{\tau_z \geq \tau}{\Gamma, z : \tau_z \vdash z : \tau}$$

By the assumption (1), we have a well-defined substitution S_y such that $S_y \tau'_y \geq \tau$ where $\tau'_y = (\Gamma + y : \forall\beta_1 \ldots \forall\beta_n.\tau_y)(y)$. Therefore, $\Gamma + y : \forall\beta_1 \ldots \forall\beta_n.\tau_y \vdash y : \tau$ holds.

If $e = x \neq z$, Trivial from $[y/z]x = x$.

(ETu-App/Send) The last proof step is

$$\frac{\Gamma_1, z : \tau_{z_1} \vdash e_1 : \tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau_2 \quad \kappa_2 \geq 1 \quad \tau_1 \not\simeq O \quad \Gamma_2, z : \tau_{z_2} \vdash e_2 : \tau_1}{(\Gamma_1 + \Gamma_2), z : (\tau_{z_1} + \tau_{z_2}) \vdash e_1\ e_2 : \tau_2}$$

We can easily obtain τ_{y_1} and τ_{y_2} such that $\tau_y = \tau_{y_1} + \tau_{y_2}$, $S_y\tau_{y_1} \geq \tau_{z_1}$, and $S_y\tau_{y_2} \geq \tau_{z_2}$ hold. By the induction hypothesis, we have $\Gamma_1 + y : \forall\beta_1 \ldots \forall\beta_n.\tau_{y_1} \vdash [y/z]e_1 : \tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau_2$ and $\Gamma_2 + y : \forall\beta_1 \ldots \forall\beta_n.\tau_{y_2} \vdash [y/z]e_2 : \tau_1$. By (ETu-App/Send) rule, $\Gamma_1 + \Gamma_2 + y : \forall\beta_1 \ldots \forall\beta_n.\tau_{y_2} \vdash e_1\ e_2 : \tau_2$ holds.

The other induction steps are similar to the case of (ETu-App/Send). □

Lemma A.11 (Heap Allocation) Suppose that if $\Gamma_h \vdash h^\kappa : \tau_h$, then $\Gamma \vdash E[h^\kappa] : \tau$. Then, there exists some Γ' such that $\Gamma = \Gamma' + \Gamma_h$ and $\Gamma', x : clos(\Gamma_h, \tau_h) \vdash E[x] : \tau$.

Proof We prove by straightforward induction on the structure of the context E. We show several main cases.

($E = []$) Trivial. ($\Gamma' = \emptyset$).

($E = E'e$) From $\Gamma \vdash E[h^\kappa] : \tau$ and (ETu-App/Send) rule, we have: 1) $\Gamma_1 \vdash E'[h^\kappa] : \tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau$, 2) $\Gamma_2 \vdash e : \tau_1$, 3) $\kappa_2 \geq 1$, 4) $\tau_1 \not\approx O$, and 5) $\Gamma = \Gamma_1 + \Gamma_2$. By induction hypothesis, we have Γ'_1 such that $\Gamma_1 = \Gamma'_1 + \Gamma_h$ and $\Gamma'_1, x : clos(\Gamma_h, \tau_h) \vdash E'[x] : \tau_1 \xrightarrow{(\kappa_1,\kappa_2)} \tau$. By associativity of $+$, $\Gamma = (\Gamma'_1 + \Gamma_2) + \Gamma_h$ holds. By (ETu-App/Send) rule, we have $\Gamma'_1 + \Gamma_2$ such that $(\Gamma'_1 + \Gamma_2), x : clos(\Gamma_h, \tau_h) \vdash E[x] : \tau$.

The other induction steps are similar. □

Lemma A.12 Suppose that if $\Gamma + \Gamma_e \vdash e : \tau_e$, then $\Gamma_E \vdash E[e] : \tau$ for some context E, and that $\Gamma + \Gamma_{e'} \vdash e' : \tau_e$ holds. Then, there exists some type environment Γ'_E such that $\Gamma'_E + \Gamma_{e'} \vdash E[e'] : \tau$ and $\Gamma'_E + \Gamma_e = \Gamma_E$ holds.

Proof Straightforward induction on the structure of the context E. □

Lemma A.13 If $\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e : \tau$ and $\texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{\tilde{x}}_\lambda \texttt{letrec}\ H'\ \texttt{in}\ e'$, then $\Gamma \vdash \texttt{letrec}\ H'^{-\varepsilon:\tilde{x}}\ \texttt{in}\ e' : \tau$.

Proof We prove with case analysis on the reduction rule used.

(L-Alloc) From (ETu-Heap) rule, we have $\Gamma_e \vdash E[h^\kappa] : \tau$. We obtain $\Gamma_x \vdash h^\kappa : \tau_x$ from its type derivation tree. By Lemma A.11, there exists Γ' such that $\Gamma_e = \Gamma' + \Gamma_x$ and $\Gamma', x : clos(\Gamma_x, \tau_x) \vdash E[x] : \tau$ hold. By Lemma 4.1, we can obtain (Γ'_1, Γ'_2) and $(\Gamma_{x1}, \Gamma_{x2})$ such that $\Gamma' = \Gamma'_1 \uplus_H \Gamma'_2$ and $\Gamma_x = \Gamma_{x1} \uplus_H \Gamma_{x2}$, respectively. Now, we have $(\Gamma'_1 + \Gamma'_2), x : clos(\Gamma_{x1} + \Gamma_{x2}, \tau_x) \vdash E[x] : \tau$. Let $\Gamma'' = \Gamma_{x_1} + \cdots + \Gamma_{x_n} + \Gamma_x + \Gamma'_1 + (\Gamma'_2, x : clos(\Gamma_h, \tau_h))$. For all $y \in dom(H, x = h^\kappa)$, we have $\Gamma_y \vdash (H, x = h^\kappa)(y) : \tau_y$, and $\Gamma''(y) = clos(\Gamma_y, \tau_y)$. Therefore, we have $\Gamma_{x_1 1} + \cdots \Gamma_{x_n 1} + \Gamma_{x1} + \Gamma'_1 \vdash \texttt{letrec}\ H, x = h^\kappa\ \texttt{in}\ E[x] : \tau$ because $\Gamma_{h1} + \Gamma'_1 = \Gamma_{e1}$.

(L-App) By (ETu-Heap) rule, we have $\Gamma_e, x : clos(\Gamma_x, \tau_1 \xrightarrow{(0,\kappa)} \tau_2) \vdash E[xy] : \tau$ and $\Gamma_x \vdash (\lambda z.e)^\kappa : \tau_1 \xrightarrow{(0,\kappa)} \tau_2$. From the derivation tree of the first type

judgement, we have $\Gamma_{xy} \vdash xy : \tau_{xy}$. By (ETu-App/Send) rule and (ETu-Var) rule, there exist substitutions $S_x = [\rho_{x1}/\alpha_1, \ldots, \rho_{xn}/\alpha_n]$ and $S_y = [\rho_{y1}/\beta_1, \ldots, \rho_{ym}/\beta_m]$ such that:

1. $\Gamma_1, x : \forall\alpha_1 \ldots \forall\alpha_n.\tau_1 \xrightarrow{(0,\kappa_1)} \tau_2 \vdash x : S_x(\tau_1 \xrightarrow{(0,\kappa'_1)} \tau_2)$
2. $\Gamma_2, y : \forall\beta_1 \ldots \forall\beta_m.\tau_3 \vdash y : \tau'_3$
3. $S_y\tau_3 \geq \tau'_3 \equiv S_x\tau_1$
4. $\kappa_1 \geq \kappa'_1 \geq 1$
5. $\Gamma_{xy} = (\Gamma_1 x : \forall\alpha_1 \ldots \forall\alpha_n.\tau_1 \xrightarrow{(0,\kappa_1)} \tau_2) + (\Gamma_2, y : \forall\beta_1 \ldots \forall\beta_m.\tau_3)$
6. $S_x\tau_2 \equiv \tau_{xy}$

Meanwhile, from $\Gamma_x \vdash (\lambda z.e)^{\kappa} : \tau_1 \xrightarrow{(0,\kappa)} \tau_2$, (ETu-Abs) rule and Lemma 3.4, we have $S_x(\Gamma'_x, z : \tau_1) \vdash e : S_x\tau_2$ and $\Gamma_x \geq \kappa \cdot \Gamma'_x$. By Lemma A.10, $S_x\Gamma'_x + y : \forall\beta_1 \ldots \forall\beta_m.\tau_3 \vdash [y/z]e : \tau_{xy}$ holds. Let κ''_1 be κ_1 if $\kappa = \omega$, or 0 otherwise. By Lemma A.9, we have $(\Gamma_1, x : \forall\alpha_1 \ldots \forall\alpha_n.\tau_1 \xrightarrow{(0,\kappa''_1)} \tau_2) + (\Gamma_2, y : \forall\beta_1 \ldots \forall\beta_m.\tau_3) + S_x\Gamma'_x \vdash [y/z]e : \tau_{xy}$. By Lemma A.12, $S_x\Gamma'_x + (\Gamma_e, x : clos(\Gamma_x, \tau_1 \xrightarrow{(0,\kappa^-)} \tau_2)) \vdash E[[y/z]e] : \tau$ holds. We have $S_x\Gamma'_x = \Gamma'_x$ because no bound type variables $\alpha_1, \ldots, \alpha_n$ of $clos(\Gamma_x, \tau_1 \xrightarrow{(0,\kappa)} \tau_2)$ is free in Γ'_x by the definition of $clos$. We can easily obtain $\tilde{\Gamma}_x$,$\tilde{\Gamma}_{x1}$ and $\tilde{\Gamma}_{x2}$ such that $\tilde{\Gamma}_x = \tilde{\Gamma}_{x1} \uplus_H \tilde{\Gamma}_{x2}$ and $\tilde{\Gamma}_x + \Gamma'_x = \Gamma_x$. By (ETu-Abs) rule, $\tilde{\Gamma}_x \vdash (\lambda z.e)^{\kappa^-} : \tau_1 \xrightarrow{(0,\kappa^-)} \tau_2$ because $\tilde{\Gamma}_x \geq \kappa^- \cdot \Gamma'_x$ holds. Let $\Gamma_x = \Gamma_{x1} \uplus_H \Gamma_{x2}$ and $\Gamma'_x = \Gamma'_{x1} \uplus_H \Gamma'_{x2}$. It suffices that $\tilde{\Gamma}_{x_1} + \Gamma'_{x1} = \Gamma_{x1}$. Therefore, $\Gamma \vdash \texttt{letrec}\ H, x = (\lambda z.e)^{\kappa^-}\ \texttt{in}\ E[[y/z]e] : \tau$ is derived from (ETu-Heap) rule.

About the other rules, proved similarly. □

Proof of Subject Reduction Theorem (3) We prove by induction on the proof of $\Gamma \vdash \texttt{letrec}\ H\ \texttt{in}\ e \xrightarrow{l} \texttt{letrec}\ H'\ \texttt{in}\ e'$ with case analysis on the last rule used.

(ERu-Lamb) Follow from Lemma A.13.

For the cases of the other rules obtain $\Gamma_e \vdash_{\mathcal{ETU}} e : O$ by (ETu-Heap) and we can discuss similarly to the proof of Subject Reduction Theorem (2) (Appendix A.1). □