TECHNICAL REPORT 95-04

Static Analysis on Communication for Asynchronous Concurrent Programming Languages

Naoki Kobayashi, Motoki Nakade and Akinori Yonezawa

April 1995

Department of Information Science
Faculty of Science, University of Tokyo

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

TECHNICAL REPORT 95-04

TITLE
Static Analysis on Communication for Asynchronous Concurrent Programming Languages

AUTHORS
Naoki Kobayashi, Motoki Nakade and Akinori Yonezawa

KEY WORDS AND PHRASES
static analysis, effect system, asynchronous concurrent programming languages

ABSTRACT
We propose an effect-based static analysis technique on communication for *asynchronous* concurrent programming languages. Our analysis gives an upper-bound of the number of enqueued messages and receivers for each communication channel, which can be used for compile-time optimizations for implementation of message passing. The main targets of our analysis are concurrent object-oriented languages, for which no formal static analysis method has been established.

REPORT DATE	**WRITTEN LANGUAGE**
April 1995	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
24	20

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT
First issue 35 copies.

SUPPLEMENTARY NOTES

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

Static Analysis on Communication for Asynchronous Concurrent Programming Languages

Naoki Kobayashi
koba@is.s.u-tokyo.ac.jp

Motoki Nakade
nakade@is.s.u-tokyo.ac.jp

Akinori Yonezawa
yonezawa@is.s.u-tokyo.ac.jp
Department of Information Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan

Abstract

We propose an effect-based static analysis technique on communication for *asynchronous* concurrent programming languages. Our analysis gives an upper-bound of the number of enqueued messages and receivers for each communication channel, which can be used for compile-time optimizations for implementation of message passing. The main targets of our analysis are concurrent object-oriented languages, for which no formal static analysis method has been established.

1 Introduction

Recent high-level concurrent programming languages, including concurrent object-oriented programming languages[1][20][19], concurrent logic programming languages[18][14], and Concurrent ML(CML)[13], are based on computational models where multiple processes (or *concurrent objects*) perform local computation, exchanging values via asynchronous or synchronous communication. Although such computational models provide potentially more powerful abstractions for programs than sequential languages such as functional programming languages and sequential object-oriented languages, extensive use of communication between processes often causes a large overhead due to cost of manipulating messages and processes queue. Message passing is generally more costly than function call. Recent work on implementation of a concurrent object-oriented language[17] tried to reduce the gap between cost of message passing and that of function call. It reduced cost of message passing very close to that of function call, when a receiver is ready to receive the message. However, in general, message passing is still more costly than function call. The aim of this paper is to reduce the cost of message passing by *static analysis* on communication that might occur when processes are executed.

Based on a kind of effect analysis[15], we propose a static analysis to infer the maximum number of enqueued messages and enqueued receivers for each communication channel during computation. By slightly modifying the typing/effect rules, we can also approximate how many messages are sent and received along each communication channel. If we know that at most one message is enqueued for the communication channel c, we need only allocate a memory for saving a single message. Hence both cost and memory space for manipulating a message queue is reduced. If only a constant number of messages are enqueued, we need only allocate an array of the fixed size for storing messages, and no boundary check is required. The same argument also holds for receiver queue; several concurrent programming languages, including CML[13], PICT[12] and HACL[6], allow multiple processes to compete for the same communication channel, hence receivers must be

maintained by a queue. Using the information obtained from our analysis technique, the cost for manipulating receiver queues is also reduced. Moreover, we can also completely eliminate message passing under a stronger condition.

The motivation for the analysis of communication behavior is not solely to improve performance of message passing. If we can satisfactorily analyze communication behavior, then we will be able to apply various techniques of analysis and optimization developed for sequential languages. In order to illustrate the difficulty in optimizing concurrent programs, consider the following fragment of a program:

$$m(v) \mid m(x) => e$$

where $m(v)$ is a process for sending a value v along a channel m, $m(x) => e$ is a process that receives some value v along a channel m and executes $e[v/x]$, and $\mid$ is an operator for parallel composition. In functional languages, $(\lambda x.e)v$ can be replaced by $e[v/x]$, and then further optimization may be made by partial evaluation of $e[v/x]$. However, in concurrent languages, replacement of $m(v) \mid m(x) => e$ by $e[v/x]$ may be wrong, because other processes may send or receive values along m, hence $m(v)$ and $m(x) => e$ may not communicate each other. Thus, communication operations often leave no room for further optimizations which were possible in sequential languages. Our analysis of the required queue length solves this problem; if at most one message and one receiver is enqueued on channel m during the execution of $m(v) \mid m(x) => e \mid P$, then it is correct to transform it to $e[v/x] \mid P$. Then, partial evaluation may be performed for $e[v/x]$.

We use our linear logic-based concurrent calculus called Higher-Order ACL (HACL) for presenting our static analysis technique. HACL is theoretically clear and provides powerful programming models such as higher-order concurrent programming.[1] Since many concurrent programming languages including concurrent object-oriented languages can be naturally embedded in HACL[7][4], the techniques proposed here are easily applied to such languages. As far as the author knows, this is the first formal static analysis technique applicable to concurrent object-oriented languages.

The rest of this paper is organized as follows. Section 2 introduces the syntax and the basic type system of HACL. In Section 3 and Section 4, we present a simple version of our static analysis method. Section 5 discusses optimizations based on our static analysis. Section 6 discusses several refinements of the simple analysis in Section 3 and Section 4. Section 7 discusses related work and Section 8 concludes this paper.

2 Overview of the Core-HACL

This section briefly introduces the core-HACL. For the concrete definition of HACL and the correspondence between linear logic and HACL, please refer to [6][5][3].[2] The core-HACL introduced here can be considered as core-ML extended with primitives for channel creation, value passing along channels, guarded choice, and parallel composition. Channels are first-class values, that is, they can be passed along other channels as in π-calculus.

2.1 Syntax

In HACL, computation is performed by multiple processes which communicate each other via asynchronous message passing. Message can be sent by specifying a communication channel. If m is a communication channel, $m(v)$ is interpreted as a process which sends a value v via channel m. On the other hand, $m(x) => P$ is a process which waits for a message $m(v)$ and then behaves like $P[v/x]$. Communication channels are created by the operator \$. $P_1|P_2$ is a concurrent composition of P_1 and P_2. $(m(x) => P)\&(n(x) => Q)$ receives a message along channel m or

[1] Unlike CML, communication behaviors are explicitly specified in HACL, hence we need not an intermediate phase for extracting communication behaviors used in [9]. The intermediate representation in [9] is helpful for understanding analyses for CML, but it seems to us that useful information obtained by our analysis has been already lost in the intermediate representation.

[2] HACL has a strong connection with linear logic, but we don't discuss it here since it is not important for our analysis.

Syntax	Description
_	inaction
$x(e_1, \ldots, e_n)$ | P	message send
$x(x_1, \ldots x_n)$ => P	message receiver
$R_1 \& R_2$	(guarded) choice
$P_1 \mid P_2$	parallel composition
$\$x.P$	name creation
rec $p(x) = e$	recursive definition

Figure 1: Syntax of $HACL$ process terms

n and accordingly behaves like P or Q. For example, `$m.$n.(m(2) | (m(x)=>P)&(n(x)=>Q))` reduces to `$m.$n.(P[2/x])`. Processes of HACL are summarized in Figure 1. In the figure, x ranges over variables, P over process expressions, and R over process expressions of the form $m(x_1, \ldots, x_n)$=>P.

The entire syntax of preterms of the core-HACL is given as follows:

Definition 2.1 (preterms) A set of *preterms*, ranged over by e, is defined by the following syntax:

e ::=x (variables)
| c (constants)
| (e_1, e_2) (pair)
| #1(e1) | #2(e2) (projection)
| $e_1 e_2$ (function application, or message send)
| $\lambda x.e$ (abstraction)
| **let val** $x = e_1$ **in** e_2 **end** (polymorphic definition)
| rec$f(x) = e$ (recursive functions or processes)
| $(e_1 | e_2)$ (parallel composition)
| r (message receiver)
| $\$x.e$ (channel creation)
r ::= $x(y) => e \mid r_1 \& r_2$

where x, y stands for variables and c for constants including special constants _(inaction), and basic values such as integers.

Note that $e_1 e_2$ is either function application or message send according to whether e_1 is evaluated to a function or a communication channel. This overloading is useful for optimizations described in Section 5. We often write $m(x_1, \ldots, x_n) => e$ for

$$m(x) => (\textbf{let } x_1 = \#1(x) \textbf{ in } \cdots \textbf{let } x_n = \#2(\cdots \#2(x) \cdots) \textbf{ in } e \textbf{ end} \cdots \textbf{end}),$$

and write `proc p(x)=e` (or `fun p(x)=e`) for `val p=(rec p(x)=e)`.

2.2 Reduction Semantics

Reduction semantics for the core-HACL is defined via three relations: structural congruence, reduction relation on λ-terms, and reduction relation on processes. We identify variables up to renaming of bound variables.

Definition 2.2 (structural congruence: $\cong$) *Structural congruence* $\cong$ over preterms is the least reflexive and transitive congruence relation closed under the following rules.

(Cong-1) $p_1 \mid p_2 \cong p_2 \mid p_1$

(Cong-2) $(p_1 \mid p_2) \mid p_3 \cong p_1 \mid (p_2 \mid p_3)$

(Cong-3) $p_1 \& p_2 \cong p_2 \& p_1$

(Cong-4) $(p_1 \& p_2) \& p_3 \cong p_1 \& (p_2 \& p_3)$

(Cong-5) $\$x.(p_1 \mid p_2) \cong p_1 \mid \$x.p_2$ if x is not free in p_1.

Definition 2.3 (reduction on λ-terms) Reduction relation $\longrightarrow_\lambda$ on λ-terms is the least relation closed under the following rules:

(R-Beta) $(\lambda x.e)v \longrightarrow_\lambda e[v/x]$

(R-App1) $\dfrac{e_1 \longrightarrow_\lambda e_1'}{e_1 e_2 \longrightarrow_\lambda e_1' e_2}$

(R-App2) $\dfrac{e \longrightarrow_\lambda e'}{ve \longrightarrow_\lambda ve'}$

(R-Pair1) $\dfrac{e_1 \longrightarrow_\lambda e_1'}{(e_1, e_2) \longrightarrow_\lambda (e_1', e_2)}$

(R-Pair2) $\dfrac{e \longrightarrow_\lambda e'}{(v, e) \longrightarrow_\lambda (v, e')}$

(R-Proj1) $\#1(v_1, v_2) \longrightarrow_\lambda v_1$

(R-Proj2) $\#2(v_1, v_2) \longrightarrow_\lambda v_2$

(R-Let1) $\dfrac{e_1 \longrightarrow_\lambda e_1'}{\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \longrightarrow_\lambda \textbf{let } x = e_1' \textbf{ in } e_2 \textbf{ end}}$

(R-Let2) $\textbf{let } x = v \textbf{ in } e \textbf{ end} \longrightarrow_\lambda e[v/x]$

(R-Rec) $\texttt{rec} f(x) = e \longrightarrow_\lambda \lambda x.e[(\texttt{rec} f(x) = e)/f]$

where v ranges over a subset of preterms defined by:

$v ::= c | x | \lambda x.e | (v_1, v_2)$

Definition 2.4 (reduction relation) Reduction relation $\longrightarrow$ on preterms is the least relation closed under the following rules:

(R-Com) $m(x) => e \mid m(v) \longrightarrow e[v/x]$

(R-Cong): $\dfrac{e_1 \cong e_2 \quad e_1 \longrightarrow e_1' \quad e_1' \cong e_2'}{e_2 \longrightarrow e_2'}$

(R-Lambda): $\dfrac{e \longrightarrow_\lambda e'}{e \longrightarrow e'}$

(R-Par) $\dfrac{e_1 \longrightarrow e_1'}{e_1 \mid e_2 \longrightarrow e_1' \mid e_2}$

(R-Choice) $\dfrac{e_1 \mid e_3 \longrightarrow e'}{(e_1 \& e_2) \mid e_3 \longrightarrow e'}$

(R-New) $\dfrac{e \longrightarrow e'}{\$x.e \longrightarrow \$x.e'}$

(R-Inaction) $_ \mid e \longrightarrow e$

2.3 Type System

A *term* of the core-HACL is defined as a well-typed preterm.

Definition 2.5 (monotypes, type schemes) A set of *monotypes*, ranged over by τ, and a set of *type schemes*, ranged over by σ, are given by the following syntax:

$\tau ::= \alpha \mid \tau_b \mid o \mid \tau \to \tau \mid \tau \times \tau$
$\sigma ::= \tau \mid \forall \alpha.\sigma$

where α stands for type variables and τ_b for base types, and o is a distinguished type for propositions (i.e., processes or messages).

Definition 2.6 A preterm e is a term of type τ under a type assignment (i.e., a map from variables to types) $\mathcal{T}$ if there is a derivation for $\mathcal{T} \vdash e : \tau$ by typing rules in Figure 2.[3]

[3] For the convenience of the extension in the later section, we used substitution for presenting the rule for **let**-expressions It is well-known that for closed terms in core-ML, the type system is equivalent to the ordinary Damas-Milner's one using type scheme. The same argument also holds for closed terms in core-HACL.

(T-const) $\mathcal{T} \vdash c : \tau$ if $(c,\sigma) \in$ **Const** and τ is a generic instance of σ

(T-var) $\mathcal{T} \vdash x : \tau$ if $x \in dom\mathcal{T}, \mathcal{T}(x) = \tau$.

(T-abs) $\dfrac{\mathcal{T}\{x \mapsto \tau_1\} \vdash e : \tau_2}{\mathcal{T} \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$

(T-app) $\dfrac{\mathcal{T} \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{T} \vdash e_1 e_2 : \tau_2}$

(T-let) $\dfrac{\mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{T} \vdash e_2[e_1/x] : \tau_2}{\mathcal{T} \vdash \textbf{let val } x = e_1 \textbf{ in } e_2 \textbf{ end} : \tau_2}$

(T-new) $\dfrac{\mathcal{T}\{x \mapsto \tau \rightarrow o\} \vdash e : o}{\mathcal{T} \vdash \$x.e : o}$

(T-com) $\dfrac{\mathcal{T} \vdash m : \tau \rightarrow o \;\; \mathcal{T} \vdash \lambda x.e : \tau \rightarrow o}{\mathcal{T} \vdash m(x) => e : o}$

(T-rec) $\dfrac{\mathcal{T}\{p \mapsto \tau_1 \rightarrow \tau_2\} \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}{\mathcal{T} \vdash \textbf{rec } p(x) = e : \tau_1 \rightarrow \tau_2}$

Figure 2: Typing Rules

In Figure 2, **Const** is a set of pairs (c,σ), where c is a constant and σ is its type scheme. We assume **Const** contains at least the following constants:

$|, \&$[4] $: o \times o \rightarrow o$
$_ : o$
$(\bullet,\bullet) : \forall\alpha\forall\beta.(\alpha \rightarrow \beta \rightarrow \alpha \times \beta)$
$\#1 : \forall\alpha\forall\beta.(\alpha \times \beta \rightarrow \alpha)$
$\#2 : \forall\alpha\forall\beta.(\alpha \times \beta \rightarrow \beta)$

We write $\mathcal{T} \vdash_{\mathcal{TR}} e : \tau$ if $\mathcal{T} \vdash e : \tau$ is derivable by the rules in Figure 2. Throughout this paper, we assume that e_2 in $e_1 e_2$, e_1 in **let** $x = e_1$ **in** e_2 **end** never have type o. Therefore, for example, an expression **let** $x = \$m.m(1)$ **in** x **end** is not allowed, but **let** $x = \lambda().\$m.m(1)$ **in** $x()$ **end** is allowed. Due to this restriction, channels are not created when e_1 in **let** $x = e_1$ **in** e_2 **end** is evaluated, since communication channels can be created only in the term of type o (see the rule **($)**). Therefore, our type system works without any imperative type variables used in CML[13]. We use the basic type system just for the simplicity, and ignore how to ensure the above property, since it is not important for the purpose of this paper. Readers interested can refer to [6] on the way to ensure the above property by refining the basic type system.

Since the typing rules of the core-HACL are essentially the same as the ML type system, types can be automatically recovered from untyped terms. The following is a sample session of the HACL type inference system.

```
proc forwarder (m, n) = m x => n x;
```
val forwarder = proc: ('a->o)('a->o)->o*

The process `forwarder` takes two communication channels `m` and `n` as arguments, and receives a value of `x` along a channel `m` and sends it along a channel `n`. The inferred type for `forwarder` implies that it takes two channels as arguments, and that they take an argument of the same type.

The following is a concurrent object-oriented style definition for point process:

```
proc point (x, y) (getx,gety,set) =
   getx(reply) => (reply(x) | point(x,y)(getx,gety,set))
 & gety(reply) => (reply(y) | point(x,y)(getx,gety,set))
 & set(newx,newy,ack) => (ack() | point(newx,newy)(getx,gety,set));
```
val point=proc: 'a'b->((('a->o)->o)*((('b->o)->o)*('a*'b*(unit->o) ->o)->o*

The process `point` has three channels for receiving messages.[5] If it receives a message along a channel `getx`, it sends `x` along the received channel `reply` and repeat the same behavior. If it receives a message along a channel `gety`, it sends `y` along the received channel `reply` and repeat

[4] |, & are infix operators.

[5] In [7], we introduce records and put three channels into a record. Then, the resulting definition is much more similar to definition of concurrent objects. However, this paper does not use records for the simplicity.

the same behavior. If it receives a message via a channel `set`, it sends an acknowledgement to `ack` and updates values of `x` and `y` to `newx` and `newy`. Please keep in mind that concurrent objects are modelled by recursive calls for the same process (in the above case, recursive calls for `point`). This is one of the reasons why we cannot apply Nielson's techniques[9][10] to analysis for concurrent object-oriented style programs. With their analyses, the number of input operations for `getx` and output operations for `reply`, etc. is always counted as ∞, although it is actually the case only when external processes send the infinite number of `getx` messages. As is given in the later section, we can infer the number of enqueued recipients on the channel `getx` to be at most 1.

3 Type/Effect System for Analyzing Communication

This section presents an analysis technique for approximating the maximum number of enqueued messages and message receivers for each communication channel. The technique presented here was initially inspired by Nielson's analysis[9], but typing/effect rules are quite different (see `(com)` rule). For the clarity, we present a simple analysis which gives only a rough (but yet more informative than Nielson's analysis) information. We will discuss several refinements later in Section 6.

Definition 3.1 (effect) Effects, ranged over by b, are defined as follows:

$$b ::= \epsilon \mid \rho \mid b_1 + b_2 \mid b_1 \cup b_2 \mid minus(b,l) \mid \{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\}$$

where ρ ranges over variables, called *effect variables*, and l, l_is are labels associated with communication channels. I_i, O_i are non-negative integers less than a certain positive integer M or ∞.

Intuitively, an expression e of type o has a constant effect:

$$\{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\}$$

if, at any instance of evaluation of e, the number of enqueued messages into the communication channel labelled by l_i is at most O_i, and the number of processes waiting for messages from the channel labelled by l_i is at most I_i. For example, `$m.($n.m(n) | m(x)=>x())` has an effect $\{1 \mapsto (1,1), 2 \mapsto (0,1)\}$ provided that m and n respectively has label 1 and 2. `$m.$n.(m(1) | m(x)=>n(x) | m(x)=>(n(x+1)))` has an effect $\{1 \mapsto (2,1), 2 \mapsto (0,1)\}$ provided that m and n respectively has label 1 and 2. Please remember that each I, O ranges over a *finite* set of values $\{0, 1, \ldots, M-1, \infty\}$. If $M = 1$, then I_i and O_i can range only over two elements set: $\{0, \infty\}$, hence our analysis corresponds to I/O mode analysis.

Formally, a constant effect is a map from labels to pairs of integers in $\{0, 1, \ldots, M-1, \infty\}$. ϵ, called an empty effect, maps all labels to $(0,0)$. Since we are only concerned with information on communication, we consider all expressions of types other than o have an empty effect ϵ. A mapping $\{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\}$ is defined as follows:

$$\{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\}(l) = \begin{cases} (I_i, O_i) & \text{if } l = l_i \text{ for some } i \\ (0,0) & \text{otherwise} \end{cases}$$

We write $I(b,l) = m$ and $O(b,l) = n$ if $b(l) = (m,n)$ Then, operations $+, \cup, minus$ on constant effects are defined as follows:

$$I(b_1 + b_2, l) = \begin{cases} I(b_1,l) + I(b_2,l) & \text{if } I(b_1,l) + I(b_2,l) < M \\ \infty & \text{otherwise} \end{cases}$$

$$O(b_1 + b_2, l) = \begin{cases} O(b_1,l) + O(b_2,l) & \text{if } O(b_1,l) + O(b_2,l) < M \\ \infty & \text{otherwise} \end{cases}$$

$$I(b_1 \cup b_2, l) = max(I(b_1,l), I(b_2,l)),\ O(b_1 \cup b_2, l) = max(O(b_1,l), O(b_2,l))$$

$$I(minus(b,l'),l) = \begin{cases} I(b,l) - 1 & \text{if } l = l' \text{ and } 0 < I(b,l) < \infty \\ I(b,l) & \text{otherwise} \end{cases}$$

$$O(minus(b,l'),l) = O(b,l)$$

1. $\dfrac{}{\tau =_{EC} \tau}$

2. $$\frac{\tau_1 =_{EC} \tau_1' \quad \tau_2 =_{EC} \tau_2' \quad EC \Rightarrow \{b \geq b', b' \geq b\}}{\tau_1 \xrightarrow{b} \tau_2 =_{EC} \tau_1' \xrightarrow{b'} \tau_2'}$$

3. $$\frac{\tau_1 =_{EC} \tau_1' \quad \tau_2 =_{EC} \tau_2'}{\tau_1 \times \tau_2 =_{EC} \tau_1' \times \tau_2'}$$

Figure 3: Proof system for $\tau_1 =_{EC} \tau_2$

For example, if e_1 has an effect b_1 and e_2 have an effect b_2, then $(e_1 \mid e_2)$ (concurrent composition of e_1 and e_2) has an effect $b_1 + b_2$. An inequality $b_1 \geq b_2$ on effects is defined as:

$$b_1 \geq b_2 \Leftrightarrow \forall l.((I(b_1, l) \geq I(b_2, l)) \wedge (O(b_1, l) \geq O(b_2, l)))$$

Let EC_1, EC_2 be sets of inequalities on effects. Then, we define the relation $EC_1 \Rightarrow EC_2$ by:

> For any substitutions Φ of effect constants for all effect variables in EC_1, EC_2, if ΦEC_1 is satisfied, then ΦEC_2 is also satisfied.

For example, $\{\rho_1 \geq \rho_2 \cup \rho_3\} \Rightarrow \{\rho_1 \geq \rho_2\}$ holds. By abuse of notation, we often write $EC_1 \Rightarrow b_1 \geq b_2$ for $EC_1 \Rightarrow \{b_1 \geq b_2\}$. We do not give a complete proof system for $EC_1 \Rightarrow EC_2$. However, note that $EC_1 \cup EC_2 \Rightarrow EC_1$ is admissible as an axiom.

Now, types are extended as follows:

Definition 3.2 (types) Types, ranged over by τ, are defined as follows:

$$\tau ::= \alpha \mid \tau_b \mid o^l \mid \tau_1 \xrightarrow{b} \tau_2 \mid \tau_1 \times \tau_2$$

where b ranges over effects and l ranges over a set of labels of communication channel.

Intuitively, an expression of type $\tau_1 \xrightarrow{b} \tau_2$ has an effect b when applied to an expression of τ_1. The type o for a process or a message is labelled with an integer label, which is used to identify each communication channel. (When it is not important, we often omit the label and just write o.) Since type expressions may contain effect variables, an equality between types depend on the underlying effect constraints EC. Thus, we write $\tau_1 =_{EC} \tau_2$ when "τ_1 and τ_2 are identical under an effect constraint EC. On the other hand, we write $\tau_1 \equiv \tau_2$ if τ_1 and τ_2 are syntactically identical. A proof system for $\tau_1 =_{EC} \tau_2$ is given in Figure 3.

A new type judgement form is $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$, which should be read, "Under an effect constraint EC, a type assignment $\mathcal{T}$, and a label assignment $\mathcal{L}$, e has a type τ and an effect b. EC, called an *effect constraint*, is a set of inequalities on effects. $\mathcal{T}$, a *type assignment*, is a mapping from a finite set of variables to types. We associate each occurrence of **$** in a given term with a unique label to identify a program point, like $\$^1 m.\$^2 n.(m(x) => n(x))$. $\mathcal{L}$ is a map from each program point to a set of labels for communication channels. Extended typing rules are given in Figure 4. We write $\mathcal{ETR}$ for the set of the extended typing rules, and write $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$ if $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$ is derivable by in $\mathcal{ETR}$.

Rules **(ET-abs)** and **(ET-app)** are fairly standard. But please note that **(ET-app)** is the rule for both function application and message send. An intuitive explanation of the **(ET-com)** rule is as follows: If there is a receiver $m(x) => e$, sending a message $m(v)$ may dequeue the receiver and cause an effect b to execute $(\lambda x.e)v$, hence an effect b' of applying m to v should be greater or equal to $minus(b, l)$, where l is a label for the channel m. The operation *minus* represents an effect that the receiver $m(x) => e$ is dequeued from the channel m. The reason b' may be *greater* is that there might be another receiver waiting for a message from the same

(ET-const) $EC, \mathcal{T}, \mathcal{L} \vdash c : (\tau, \epsilon)$ if $(c, \sigma) \in \mathbf{Const}$ and τ is a generic instance of σ

(ET-var) $EC, \mathcal{T}\{x \mapsto \tau\}, \mathcal{L} \vdash x : (\tau, \epsilon)$

(ET-par) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e_1 : (o, b_1) \quad EC, \mathcal{T}, \mathcal{L} \vdash e_2 : (o, b_2)}{EC, \mathcal{T}, \mathcal{L} \vdash e_1 | e_2 : (o, b_1 + b_2)}$$

(ET-choice) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e_1 : (o, b_1) \quad EC, \mathcal{T}, \mathcal{L} \vdash e_2 : (o, b_2)}{EC, \mathcal{T}, \mathcal{L} \vdash e_1 \& e_2 : (o, b_1 \cup b_2)}$$

(ET-abs) $$\frac{EC, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash e : (\tau_2, b)}{EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\tau_1 \xrightarrow{b} \tau_2, \epsilon)}$$

(ET-app) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e_1 : (\tau_1 \xrightarrow{b} \tau_2, \epsilon) \quad EC, \mathcal{T}, \mathcal{L} \vdash e_2 : (\tau_1, \epsilon)}{EC, \mathcal{T}, \mathcal{L} \vdash e_1 e_2 : (\tau_2, b)}$$

(ET-new) $$\frac{EC, \mathcal{T}\{x \mapsto \tau \xrightarrow{b'} o^l\}, \mathcal{L} \vdash e : (o, b) \quad EC \Rightarrow b' \geq \{l \mapsto (0,1)\} \quad l \in \mathcal{L}(\pi)}{EC, \mathcal{T}, \mathcal{L} \vdash \$^{\pi} x.e : (o, b)}$$

(ET-com) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\tau \xrightarrow{b} o, \epsilon) \quad EC, \mathcal{T}, \mathcal{L} \vdash m : (\tau \xrightarrow{b'} o^l, \epsilon) \quad EC \Rightarrow b' \geq minus(b, l)}{EC, \mathcal{T}, \mathcal{L} \vdash m(x) => e : (o, \{l \mapsto (1,0)\})}$$

(ET-rec) $$\frac{EC, \mathcal{T}\{f \mapsto \tau_1 \xrightarrow{b} \tau_2\}, \mathcal{L} \vdash \lambda x.e : (\tau_1 \xrightarrow{b} \tau_2, \epsilon)}{EC, \mathcal{T}, \mathcal{L} \vdash \mathbf{rec}\ f(x) = e : (\tau_1 \xrightarrow{b} \tau_2, \epsilon)}$$

(ET-let) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e_1 : (\tau_1, \epsilon) \quad EC, \mathcal{T}, \mathcal{L} \vdash e_2[e_1/x] : (\tau_2, b)}{EC, \mathcal{T}, \mathcal{L} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : (\tau_2, b)}$$

(ET-weak) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b) \quad EC \Rightarrow b' \geq b}{EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b')}$$

(ET-eq) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b) \quad \tau =_{EC} \tau'}{EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau', b)}$$

Figure 4: extended typing rules

channel m which has a more effect. In the rule **(ET-new)**, the second premise indicates that if a message is sent along a channel, at least that message is enqueued to the channel. The rule **(ET-weak)** allows the obtained effect to be just an *upper-bound*, rather than exact information.

For example, we can derive

$$\{b_1 \geq minus(b_2, 1)\}, \phi, \mathcal{L} \vdash \lambda m.\lambda n.(m(x) => n(x)) : ((\tau \xrightarrow{b_1} o^1) \rightarrow (\tau \xrightarrow{b_2} o^2) \xrightarrow{\{1 \mapsto (1,0)\}} o, \epsilon).$$

as is shown in Figure 5. In the figure, EC and $\mathcal{T}$ denote $\{b_1 \geq minus(b_2, 1)\}$ and $\{m \mapsto \tau \xrightarrow{b_1} o^1, n \mapsto \tau \xrightarrow{b_2} o^2\}$ respectively. The extended type system $\mathcal{ETR}$ is sound w.r.t. the basic type system $\mathcal{TR}$ in the following sense:

Proposition 3.1 (soundness of the extended type system) If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$, then

$$Erase(\mathcal{T}) \vdash_{\mathcal{TR}} \hat{e} : Erase(\tau)$$

holds, where $Erase(\tau)$ is defined by:

$Erase(\tau_1 \xrightarrow{b} \tau_2) = Erase(\tau_1) \rightarrow Erase(\tau_2)$
$Erase(\tau_1 \times \tau_2) = Erase(\tau_1) \times Erase(\tau_2)$
$Erase(\tau) = \tau$ if τ is neither an arrow type nor a product type.

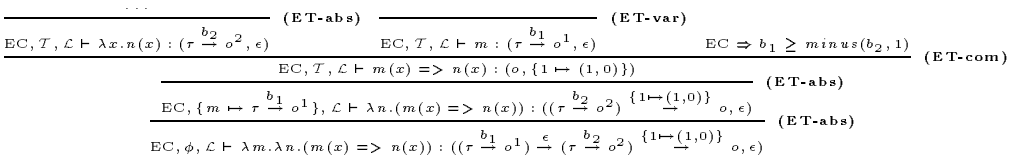

Figure 5: A sample derivation in the extended type system

The definition of *Erase* is extended pointwise to contexts as follows:

$$Erase(\phi) = \phi$$
$$Erase(\mathcal{T} \cup \{x : \tau\}) = Erase(\mathcal{T}) \cup \{x : Erase(\tau)\}$$

$\hat{e}$ is a term obtained by dropping all labels for program points.

Proof is straightforward by induction on the structure of the derivation for $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$.

We can prove the following subject reduction theorem for the extended type system.

Proposition 3.2 (subject reduction theorem)
If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (o^l, b)$ and $e \longrightarrow e'$, then $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (o^{l'}, b)$.[6]

The proof is done by induction on the structure of the proof for $e \longrightarrow e'$ in the same manner as the proof of subject reduction theorem[6] for the basic type system. The proof is given in Appendix.

As an immediate corollary of the subject reduction theorem, we can prove that the result of analysis really gives upper-bounds of lengths of message queues and receivers queues.

Corollary 3.3 *If* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (o, \{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n), \ldots\})$ *and* $\mathcal{L}(\pi) = \{l_1, \ldots, l_n\}$, *then during the evaluation of* e, *the number of enqueued messages sent to the communication channel created at the program point* π *is at most* $max(\{O_1, \ldots, O_n\})$ *and the number of waiting processes for the channel is at most* $max(\{I_1, \ldots, I_n\})$.

This corollary is based on the observation that if e is reduced to e' at some instance of reduction, and a message $m(v)$ is enqueued at that moment, then $e' \cong \$m.\$n_1.\cdots\$n_k.(m(v)|e_1)$ must hold for some $n_1, \ldots, n_k, e'$. Equally, if $m(x) => e_1$ is enqueued at that moment, then $e' \cong \$m.\$n_1.\cdots\$n_k.(m(x) => e_1|e_2)$.

Proof sketch of Corollary 3.3 Suppose $EC, \mathcal{T}, \mathcal{L} \vdash e : (o, \{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n), \ldots\})$, $\mathcal{L}(\pi) = \{l_1, \ldots, l_n\}$ and that at some moment the number of enqueued message sent to a channel m created at the program point π is $I'(> max(\{I_1, \ldots, I_n\}))$. Then, there is a transition sequence

$$e \longrightarrow^* \$^{\pi}m.\$n_1.\cdots\$n_k.(m(v_1) \mid \cdots \mid m(v_{I'}) \mid e'')$$

By the subject reduction theorem,

$$EC, \mathcal{T}, \mathcal{L} \vdash \$^{\pi}m.\$n_1.\cdots\$n_k.(m(v_1) \mid \cdots \mid m(v_{I'}) \mid e'') : (o, \{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n), \ldots\})$$

which contradicts the typing rules **(ET-new)** and **(ET-par)**. □

4 Type/Effect Reconstruction

By Corollary 3.3, for each closed process expression e, we know the upper-bounds of lengths of message queues and receiver queues for each channel, by computing $\mathcal{L}, b$ such that $\phi, \phi, \mathcal{L} \vdash e : o\&b$ (ϕ denotes an empty map). The procedure to compute $\mathcal{L}, b$ consists of the following two steps:

1. First, compute the *most general* $(EC, \mathcal{T}, \mathcal{L}, \tau, b)$ such that $EC, \mathcal{T}, \mathcal{L} \vdash e : \tau\&b$ is derivable from typing rules.
2. If e is a closed process expression, that is, if $\mathcal{T} = \phi$ and $\tau = o$, compute the minimum solution of effect constraints EC and substitute the solution for the effect variables in b.

[6] l and l' is not necessarily identical.

(ET-var') $\dfrac{\tau =_{EC} \tau'}{EC, \mathcal{T}\{x \mapsto \tau\}, \mathcal{L} \vdash x : (\tau', \epsilon)}$

(ET-abs') $\dfrac{EC, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash e : (\tau_2, b) \quad EC \Rightarrow b' \geq b}{EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\tau_1 \xrightarrow{b'} \tau_2, \epsilon)}$

Figure 6: Alternative rules

4.1 Step 1: Reconstruction of $(EC, \mathcal{T}, \mathcal{L}, \tau, b)$

The first step to reconstruct $(EC, \mathcal{T}, \mathcal{L}, \tau, b)$ is obtained by eliminating the rules **(ET-weak)** and **(ET-eq)** from $\mathcal{ETR}$ and replacing **(ET-abs)** and **(ET-var)** by the **(ET-abs')** and **(ET-var')** in Figure 6. We write $\mathcal{ETR}'$ for the set of the resulting typing rules, and write $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (\tau, b)$ if $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$ is derivable in $\mathcal{ETR}'$. Note that $\mathcal{ETR}'$ is syntax-directed; that is, for each syntactic category, the conclusion of only one rule matches. The following theorem ensures that $\mathcal{ETR}$ and $\mathcal{ETR}'$ are essentially equivalent.

Proposition 4.1 (soundness and completeness of syntax-directed rules)

1. If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (\tau, b)$, then $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$
2. If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$, then there is a b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (\tau, b')$ and $EC \Rightarrow b \geq b'$.

Proof Sketch 1. Trivial from the fact that **(ET-abs')** can be replaced by:

$$\dfrac{\dfrac{EC, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash e : (\tau_2, b) \quad EC \Rightarrow b' \geq b}{EC, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash e : (\tau_2, b')}\ \textbf{(ET-weak)}}{EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\tau_1 \xrightarrow{b'} \tau_2, \epsilon)}\ \textbf{(ET-abs)}$$

and **(ET-var')** can be replaced by:

$$\dfrac{\dfrac{}{EC, \mathcal{T}\{x \mapsto \tau\}, \mathcal{L} \vdash x : (\tau, \epsilon)}\ \textbf{(ET-var)}}{EC, \mathcal{T}\{x \mapsto \tau\}, \mathcal{L} \vdash x : (\tau', \epsilon)}\ \textbf{(ET-eq)}$$

2. Since the rule **(ET-abs)** and **(ET-var)** are special instances of **(ET-abs')** and **(ET-var')**, if we have a derivation for $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$ in $\mathcal{ETR}$, then we have a derivation for $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$ also in $\mathcal{ETR}' \cup \{\textbf{(ET-weak)}, \textbf{(ET-eq)}\}$. Therefore, it is enough to show that **(ET-weak)** can be eliminated or permuted below w.r.t. other rules and **(ET-eq)** can be eliminated or permuted above w.r.t. other rules.
□

Since $\mathcal{ETR}'$ is syntax-directed, a type/effect reconstruction algorithm is straightforwardly obtained by reading rules of $\mathcal{ETR}'$ in a bottom-up manner. We need some preliminary definitions.

Definition 4.1 (type substitution) A type substitution $S = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is defined by:

$$S\alpha = \begin{cases} \tau_i & \text{if } \alpha_i = \alpha \text{ for some } i \\ \alpha & \text{otherwise} \end{cases}$$
$$S\tau_b = \tau_b$$
$$So^l = So^l$$
$$S(\tau_1 \xrightarrow{b} \tau_2) = (S\tau_1) \xrightarrow{b} (S\tau_2)$$
$$S(\tau_1 \times \tau_2) = (S\tau_1) \times (S\tau_2).$$

A type substitution is extended pointwise to contexts.

Definition 4.2 (effect substitution) An effect substitution $\Phi = [b_1/\rho_1, \ldots, b_n/\rho_n]$ is defined by:

$\Phi b = b$ if b is a constant effect.
$\Phi\rho = \begin{cases} b_i & \text{if } \rho_i = \rho \text{ for some } i \\ \rho & \text{otherwise} \end{cases}$
$\Phi(b_1 + b_2) = \Phi b_1 + \Phi b_2$
$\Phi(b_1 \cup b_2) = \Phi b_1 \cup \Phi b_2$
$\Phi(minus(b, l)) = minus(\Phi b, l)$

The definition of an effect substitution is extended to types as follows:

$\Phi(\tau_1 \xrightarrow{b} \tau_2) = \Phi\tau_1 \xrightarrow{\Phi b} \Phi\tau_2$
$\Phi(\tau_1 \times \tau_2) = \Phi\tau_1 \times \Phi\tau_2$
$\Phi\tau = \tau$ if τ is neither an arrow type nor a product type.

The effect substitution is further extended pointwise to contexts and type substitutions.

Definition 4.3 (label renaming) A label renaming $\psi = [l'_1/l_1, \ldots, l'_n/l_n]$ is defined by:

$$\psi l = \begin{cases} l'_i & \text{if } l = l_i \text{ for some } i \\ l & \text{otherwise} \end{cases}$$

The label renaming is extended to effects and types as follows:

$\psi\epsilon = \epsilon$
$\psi\rho = \rho$
$\psi(b_1 + b_2) = \psi b_1 + \psi b_2$
$\psi(b_1 \cup b_2) = \psi b_1 \cup \psi b_2$
$\psi\, minus(b, l) = \begin{cases} minus(\psi b, l'_i) & \text{if } l = l_i \text{ for some } i \\ minus(\psi b, l) & \text{otherwise} \end{cases}$
$\psi\{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\}$
$= \begin{cases} \{l_1 \mapsto (I_1, O_1), \ldots, l_{i-1} \mapsto (I_{i-1}, O_{i-1}), l_{i+1} \mapsto (I_{i+1}, O_{i+1}), l_n \mapsto (I_n, O_n)\} + \{l'_i \mapsto (I_i, O_i)\} \\ \qquad \text{if } l = l_i \text{ for some } i \\ \{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\} \qquad \text{otherwise} \end{cases}$
$\psi\tau_1 \xrightarrow{b} \tau_2 = \psi\tau_1 \xrightarrow{\psi b} \psi\tau_2$
$\psi(\tau_1 \times \tau_2) = \psi\tau_1 \times \psi\tau_2$
$\psi\tau = \tau$ if τ is neither an arrow type nor a product type.

The label renaming is further extended pointwise to contexts, type substitutions, etc.

In the reconstruction algorithm, we need to solve an equation system E, which is a set consisting of type equations $\tau_1 = \tau_2$ where all effects in τ_1, τ_2 are effect variables. The solution is represented as a triple consisting of a type substitution, an effect substitution, and a label renaming.

Definition 4.4 (extended substitution) An *extended substitution* is a triple (S, Φ, ψ) consisting of a type substitution, an effect substitution, and a label renaming. Substitutions for types, effects, and labels are defined as:

$(S, \Phi, \psi)\tau = S\Phi\psi\tau$
$(S, \Phi, \psi)b = \Phi\psi b$
$(S, \Phi, \psi)l = \psi l$

Let $S_1 \circ S_2, \Phi_1 \circ \Phi_2, \psi_1 \circ \psi_2$ be ordinary compositions for substitutions. For example, $S_1 \circ S_2$ is a type substitution S such that $S_2 S_1 \tau = S\tau$ for any type expression τ. Then, a composition for extended substitution is defined as follows.

Definition 4.5 (composition of extended substitution) A composition $(S_1, \Phi_1, \psi_1) \circ (S_2, \Phi_2, \psi_2)$ of two extended substitutions are defined as $((\Phi_2\psi_2 S_1) \circ S_2, (\psi_2\Phi_1) \circ \Phi_2, \psi_1 \circ \psi_2)$,

Lemma 4.2 Let (S_1, Φ_1, ψ_1) and (S_2, Φ_2, ψ_2) be extended substitutions. Then, for any type expressions τ, any effect b, and any label l, the following equation holds.

$$((S_1, \Phi_1, \psi_1) \circ (S_2, \Phi_2, \psi_2))\tau = (S_2, \Phi_2, \psi_2)((S_1, \Phi_1, \psi_1)\tau)$$
$$((S_1, \Phi_1, \psi_1) \circ (S_2, \Phi_2, \psi_2))b = (S_2, \Phi_2, \psi_2)((S_1, \Phi_1, \psi_1)b)$$
$$((S_1, \Phi_1, \psi_1) \circ (S_2, \Phi_2, \psi_2))l = (S_2, \Phi_2, \psi_2)((S_1, \Phi_1, \psi_1)l)$$

Definition 4.6 (unifier) Let E be an equation system. An extended substitution (S, Φ, ψ) is a *unifier* of E if for any $\tau_1 = \tau_2 \in E$, $(S, \Phi, \psi)\tau_1 = (S, \Phi, \psi)\tau_2$ holds.

Definition 4.7 (most general unifier) A unifier (S, Φ, ψ) of E is *most general* if for any unifier (S', Φ', ψ') of E, there is an extended substitution (S'', Φ'', ψ'') such that $(S', \Phi', \psi') = (S'', \Phi'', \psi'') \circ (S, \phi, \psi)$.

It is easy to check that if there is a unifier of E, then there is a most general unifier. (Note that this does not necessarily hold without an assumption that all effects in type expressions in an equation system are effect variables.)

Proposition 4.3 There is an algorithm $\mathcal{U}$, which inputs an equation system E and outputs the most general unifier of E if there is a unifier of E, or reports failure otherwise.

For example,

$$\mathcal{U}(\{int \xrightarrow{\rho} o^l = \alpha \xrightarrow{\rho'} o^{l'}\}) = (\{\alpha \mapsto int\}, \{\rho \mapsto \rho'\}, \{l \mapsto l'\})$$

Using the unification algorithm, the reconstruction algorithm is shown in Figure 7 and Figure 8. In the figure, if f_1 and f_2 are maps, then $f_1 \cup f_2$ denotes a map f such that $f(x) = f_1(x)$ if $x \in dom(f_1)$ and $f(x) = f_2(x)$ otherwise. $\mathcal{L}_1 + \mathcal{L}_2$ is defined by: $(\mathcal{L}_1 + \mathcal{L}_2)(\pi) = \mathcal{L}_1(\pi) \cup \mathcal{L}_2(\pi)$.

Soundness and Completeness of The Reconstruction Algorithm It is easy to check that the algorithm PT is sound with respect to $\mathcal{ETR}$.

Proposition 4.4 Let $PT(e, \phi) = (EC, \mathcal{T}, \mathcal{L}, \tau, b)$. Then, for any extended substitution (S, Φ, ψ), $\Phi\psi EC, S\Phi\psi\mathcal{T}, \psi\mathcal{L} \vdash_{\mathcal{ETR}} e : (S\Phi\psi\tau, \Phi\psi b)$.

In order to claim that PT computes the most general typing, we introduce the notion of *principal typing* below. We need to consider only some canonical type judgements.

Definition 4.8 (canonical judgement) A type judgement $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$ is *canonical* if all effect expressions in $\mathcal{T}, \tau, b$ are effect variables.

It is trivial that given any type judgement $EC, \mathcal{T}, \mathcal{L} \vdash e : (\tau, b)$, we can easily obtain canonical judgement by replacing each effect expression b_i in $\mathcal{T}, \tau, b$ with a fresh effect variable ρ_i and adding an effect constraint $\{\rho_i \geq b_i, b_i \geq \rho_i\}$ to EC.

Definition 4.9 (principal typing) $(EC, \mathcal{T}, \mathcal{L}, \tau, b)$ is a *principal typing* of e if

1. EC is satisfiable, i.e., there is a ground effect substitution (i.e., effect substitution that maps each effect variable to constant effect) Φ such that ΦEC holds.
2. $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$
3. If a canonical type judgement $EC', \mathcal{T}', \mathcal{L}' \vdash_{\mathcal{ETR}} e : (\tau', b')$ is derivable, then for some extended substitution (S, Φ, ψ),
   (a) $EC' \Rightarrow \Phi\psi EC$
   (b) $S\Phi\psi\mathcal{T} \subseteq_{EC'} \mathcal{T}'$
   (c) $\psi\mathcal{L} \subseteq \mathcal{L}'$

$PT(c, A) =$ $(\phi, \phi, \phi, \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \epsilon)$
where $\langle c, \forall\alpha_1.\cdots\forall\alpha_n.\tau\rangle \in \mathbf{Const}$ and $\beta_1, \ldots, \beta_n$ are distinct fresh variables.

$PT(x, A) =$ **if** $A(x) = (EC, \mathcal{T}, \mathcal{L}, \tau, \epsilon)$
then $(EC, \mathcal{T}, \mathcal{L}, \tau, \epsilon)$ with type and effect variables renamed with fresh variables, and labels renamed with fresh labels
else $(\phi, \{x \mapsto \alpha\}, \phi, \alpha, \epsilon)$ (α is a fresh type variable)

$PT(UV, A) =$ **let** $(EC_1, \mathcal{T}_1, \mathcal{L}_1, \tau, _) = PT(U, A)$
$(EC_2, \mathcal{T}_2, \mathcal{L}_2, \tau', _) = PT(V, A)$
$(S, \Phi, \psi) = \mathcal{U}(\{\tau_1 = \tau_2 \mid \mathcal{T}_1(x) = \tau_1 \text{ and } \mathcal{T}_2(x) = \tau_2\} \cup \{\tau = \tau' \xrightarrow{\rho} \alpha\})$
where α is a fresh type variable and ρ is a fresh effect variable
in $(\Phi\psi EC_1 \cup \Phi\psi EC_2, S\Phi\psi\mathcal{T}_1 \cup S\Phi\psi\mathcal{T}_2, \psi\mathcal{L}_1 + \psi\mathcal{L}_2, S\Phi\psi\alpha, \Phi\psi\rho)$

$PT(\lambda x.U, A) =$ **let** $(EC, \mathcal{T}, \mathcal{L}, \tau_2, b) = PT(U, A)$
in **if** $\mathcal{T}(x) = \tau_1$ for some τ_1
then $(EC \cup \{\rho \geq b\}, \mathcal{T}\backslash x, \mathcal{L}, \tau_1 \xrightarrow{\rho} \tau_2, \epsilon)$
else $(EC \cup \{\rho \geq b\}, \mathcal{T}, \mathcal{L}, \alpha \xrightarrow{\rho} \tau_2, \epsilon)$
where α is a fresh variable and ρ is a fresh effect variable.

$PT($**let** $x = U$ **in** V **end**$, A) =$
let $(EC_1, \mathcal{T}_1, \mathcal{L}_1, \tau_1, _) = PT(U, A)$
$(EC_2, \mathcal{T}_2, \mathcal{L}_2, \tau_2, b) = PT(V, A\{x \mapsto (EC_1, \mathcal{T}_1, \mathcal{L}_1, \tau_1, \epsilon)\})$
$(S, \Phi, \psi) = \mathcal{U}(\{\tau_1 = \tau_2 \mid \mathcal{T}_1(y) = \tau_1 \text{ and } \mathcal{T}_2(y) = \tau_2\}$
in $(\Phi\psi EC_1 \cup \Phi\psi EC_2, S\Phi\psi\mathcal{T}_1 \cup S\Phi\psi\mathcal{T}_2, \psi\mathcal{L}_1 + \psi\mathcal{L}_2, S\Phi\psi\tau_2, \Phi\psi b)$

$PT(m(x) => U, A) =$
let $(EC_1, \mathcal{T}_1, \mathcal{L}_1, \tau, _) = PT(m, A)$
$(EC_2, \mathcal{T}_2, \mathcal{L}_2, \tau' \xrightarrow{\rho} \tau'', _) = PT(\lambda x.U, A)$
$(S, \Phi, \psi) = \mathcal{U}(\{\tau_1 = \tau_2 \mid \mathcal{T}_1(y) = \tau_1 \text{ and } \mathcal{T}_2(y) = \tau_2\} \cup \{\tau = \tau' \xrightarrow{\rho'} o^k, \tau'' = o\}$
where ρ' is a fresh effect variable and k is a fresh label.
in $(\Phi\psi EC_1 \cup \Phi\psi EC_2 \cup \{\Phi\rho' \geq minus(\Phi\rho, \psi k)\}, S\Phi\psi\mathcal{T}_1 \cup S\Phi\psi\mathcal{T}_2, \psi\mathcal{L}_1 + \psi\mathcal{L}_2, o, \{\psi k \mapsto (1,0)\})$

$PT(\$^{\pi} x.U, A) =$
let $(EC, \mathcal{T}, \mathcal{L}, \tau, b) = PT(U, A)$
in **if** $\mathcal{T}(x) = \tau'$ for some τ'
then $(\Phi\psi EC \cup \{\Phi\rho \geq \{\psi l \mapsto (0,1)\}, S\Phi\psi(\mathcal{T}\backslash x), \psi\mathcal{L} + \{\pi \mapsto \{\psi l\}\}, o, \psi\Phi b)$
where $(S, \Phi, \psi) = \mathcal{U}(\{\tau' = \alpha \xrightarrow{\rho} o^l, \tau = o\})$ and α is a fresh type variable, ρ is a fresh effect variable, and l is a fresh label.
else $(EC, S\mathcal{T}, \mathcal{L}, o, b)$ where $(S, _, _) = \mathcal{U}(\{\tau = o\})$.

$PT(\texttt{rec}f(x) = U, A) =$
let $(EC, \mathcal{T}, \mathcal{L}, \tau \xrightarrow{\rho} \tau', _) = PT(\lambda x.U, A)$
in **if** $\mathcal{T}(f) = \tau''$ for some τ''
then $(\Phi\psi EC, S\Phi\psi(\mathcal{T}\backslash f), \psi\mathcal{L}, S\Phi\psi\tau'', \epsilon)$
where $(S, \Phi, \psi) = \mathcal{U}(\{\tau'' = \tau \xrightarrow{\rho} \tau'\})$
else $(EC, \mathcal{T}, \mathcal{L}, \tau \xrightarrow{\rho} \tau', \epsilon)$

Figure 7: Type inference algorithm PT

$$
\begin{array}{rl}
PT(U \mid V, A) = & \textbf{let } (EC_1, \mathcal{T}_1, \mathcal{L}_1, \tau, b_1) = PT(U, A) \\
& \quad (EC_2, \mathcal{T}_2, \mathcal{L}_2, \tau', b_2) = PT(V, A) \\
& \quad (S, \Phi, \psi) = \mathcal{U}(\{\tau_1 = \tau_2 \mid \mathcal{T}_1(x) = \tau_1 \text{ and } \mathcal{T}_2(x) = \tau_2\} \cup \{\tau = o, \tau' = o\} \\
& \textbf{in } \quad (\Phi\psi EC_1 \cup \Phi\psi EC_2, S\Phi\psi\mathcal{T}_1 \cup S\Phi\psi\mathcal{T}_2, \psi\mathcal{L}_1 + \psi\mathcal{L}_2, o, \Phi\psi b_1 + \Phi\psi b_2) \\
PT(U\&V, A) = & \textbf{let } (EC_1, \mathcal{T}_1, \mathcal{L}_1, \tau, b_1) = PT(U, A) \\
& \quad (EC_2, \mathcal{T}_2, \mathcal{L}_2, \tau', b_2) = PT(V, A) \\
& \quad (S, \Phi, \psi) = \mathcal{U}(\{\tau_1 = \tau_2 \mid \mathcal{T}_1(x) = \tau_1 \text{ and } \mathcal{T}_2(x) = \tau_2\} \cup \{\tau = o, \tau' = o\} \\
& \textbf{in } \quad (\Phi\psi EC_1 \cup \Phi\psi EC_2, S\Phi\psi\mathcal{T}_1 \cup S\Phi\psi\mathcal{T}_2, \psi\mathcal{L}_1 + \psi\mathcal{L}_2, o, \Phi\psi b_1 \cup \Phi\psi b_2)
\end{array}
$$

Figure 8: Type inference algorithm PT(cont.)

(d) $S\Phi\psi\tau =_{EC'} \tau'$

(e) $EC' \Rightarrow \Phi\psi b \leq b'$

where $\mathcal{T}_1 \subseteq_{EC'} \mathcal{T}_2$ if $\mathcal{T}_1(x) =_{EC'} \mathcal{T}_2(x)$ for any $x \in dom(\mathcal{T}_1)$, and $\mathcal{L}_1 \subseteq \mathcal{L}_2$ if $\mathcal{L}_1(\pi) \subseteq \mathcal{L}_2(\pi)$ for any $\pi \in dom(\mathcal{L}_1)$.

The first condition is to exclude an effect constraint that contradicts by itself. For example, if we take $EC = \{\epsilon \geq \{1 \mapsto (1, 0)\}\}$, then $EC \Rightarrow EC'$ holds for any EC. We need to exclude out such an effect constraint.

Now, we can prove that the algorithm PT computes the principal typing.

Proposition 4.5 (completeness of reconstruction algorithm) Given a closed term e, $PT(e, \phi)$ computes a principal typing of e if it exists, or report failure otherwise.

Proof sketch of the above proposition is given in Appendix B.

4.2 Step 2: Solving the effect constraint EC

In the second step, in order to compute the minimum effect for a process expression (i.e., an expression of type o), we need to solve an effect constraint EC output by the inference algorithm PT. The following lemma is immediate from the definition of PT.

Lemma 4.6 Let $PT(e, \phi) = (EC, \ldots)$. Then, EC is a set of inequalities of the following form:

$$
\begin{array}{l}
\rho_1 \geq b_1(\rho_1, \ldots, \rho_n) \\
\cdots \\
\rho_n \geq b_n(\rho_1, \ldots, \rho_n)
\end{array}
$$

where $\rho_1, \ldots, \rho_n$ are effect variables, and $b_1, \ldots, b_n$ are functions on effects composed of operations defined in Definition 3.1.

Note that all operations on effects in Definition 3.1 are *monotonic*, hence the above functions $b_1, \ldots, b_n$ are monotonic. Moreover, since labels (for communication channels) in $b_1, \ldots, b_n$ are finite, constant effects range over the finite space. Therefore, we can prove that the minimum solution of EC always exists and is computable in finite steps.

Definition 4.10 Let EC be a set of inequalities of the following form.

$$
\begin{array}{l}
\rho_1 \geq b_{1,EC}(\rho_1, \ldots, \rho_n) \\
\cdots \\
\rho_n \geq b_{n,EC}(\rho_1, \ldots, \rho_n)
\end{array}
$$

The above inequalities are abbreviated as $\vec{\rho} \geq \vec{b}_{EC}(\vec{\rho})$. Then, we define $\vec{\rho}^{(k)}_{EC} = (\rho^{(k)}_{1,EC}, \ldots, \rho^{(k)}_{n,EC})(k \geq 0)$ by:

$$\vec{\rho}_{EC}^{(0)} = \vec{\epsilon} = (\epsilon, \ldots, \epsilon)$$
$$\vec{\rho}_{EC}^{(i+1)} = \vec{b}_{EC}(\vec{\rho}_{EC}^{(i)}) = (b_{1,EC}(\rho_{1,EC}^{(i)}, \ldots, \rho_{n,EC}^{(i)}), \ldots, b_{n,EC}(\rho_{1,EC}^{(i)}, \ldots, \rho_{n,EC}^{(i)}))$$

Proposition 4.7 (existence of minimum solution) For some K, $\vec{\rho}_{EC}^{(K)}$ is the *least fixpoint* of $\vec{b}_{EC}$, that is,

1. $\vec{b}_{EC}(\vec{\rho}_{EC}^{(K)}) = \vec{\rho}_{EC}^{(K)}$
2. If $\vec{b}_{EC}(b') = b'$ for some effect b', then $b' \geq \vec{\rho}_{EC}^{(K)}$

The above proposition ensures that $\rho = \vec{\rho}_{EC}^{(K)}$ is the minimum solution of effect constraints EC. Then, if $PT(e, \phi) = (EC, \phi, \mathcal{L}, o, b)$, the minimum b' such that $\phi, \phi, \mathcal{L} \vdash e : o \& b'$ is given by $b' \equiv b[\vec{\rho}_{EC}^{(K)}/\vec{\rho}]$.[7]

Example The following expression creates a new `point` process (`point` has been defined in Section 2) with state variables $x = 1.0, y = 0.0$, and then sends `getx` messages twice to the point process.

```
$getx.$gety.$set.(point(1.0,0.0)(getx,gety,set) (* creates new point process *)
                 | $reply.(getx(reply) |        (* sends getx message *)
                           reply(x)=>(getx(reply) | reply(x')=>_)))
                  (* receives reply, and then sends getx message again *)
```

Our analysis infers an effect of the above expression as

$$\{l_1 \mapsto (1,1), l_2 \mapsto (1,0), l_3 \mapsto (1,0), l_4 \mapsto (1,1)\}$$

where l_1, l_2, l_3, l_4 is respectively a label associated with `getx`, `gety`,`set`,`reply`. Please note that the number of enqueued messages for channels `getx` and `reply` is counted as 1, although messages are sent twice for channels `getx` and `reply`.

5 Examples of Optimization

This section gives two examples of optimizations based on information obtained by our analysis technique.

Reduction of costs for queue manipulation Since communication in HACL is based on one-to-one-of-many communication, there might be multiple senders and receivers for each communication channel. Therefore, two queues, one for messages and the other for message receivers, are required for each communication channel. Typical implementation of message sending is: (1)if the receiver queue is empty, just enqueue the argument of the message, (2)otherwise, picking up a receiver from the queue and process it. Conversely, message reception is implemented as: (1) if the message queue is empty, just enqueue the continuation, (2)otherwise picking up a message from the queue and execute it. The above implementation strategy ensures that at least either the message queue or the receiver queue is always empty, hence at most one queue manipulation is required for each communication. However, queue manipulation is still costly due to pointer manipulation, boundary check, etc. Given upper-bounds of the enqueued messages and receivers by our program analysis, we can represent each queue as an array of a fixed size and no boundary check is required if the upper-bound is finite.

[7] There are more efficient algorithms for solving the effect constraint. In fact, our current experimental analyzer incorporates a symbolic simplification method instead of the above iterative method.

Elimination of redundant message passing By replacing the typing rule **(ET-com)** by the following rule **(ET-com')** in Section 3, we can approximate the maximum number of messages that might be sent for each communication channel.

$$\textbf{(ET-com')} \quad \frac{\begin{array}{c} EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\tau \xrightarrow{b} o, \epsilon) \\ EC, \mathcal{T}, \mathcal{L} \vdash m : (\tau \xrightarrow{b'} o^l, \epsilon) \\ EC \Rightarrow b' \geq b \end{array}}{EC, \mathcal{T}, \mathcal{L} \vdash m(x) => e : (o, \{l \mapsto (1,0)\})}$$

Then, under a certain condition, we can completely eliminate message passing. Consider the following fragment of a program:

```
$reply.(m(reply) | reply(x)=>e)
```

The above fragment creates a new communication channel `reply`, sends message `m(reply)`, and gets a value of `x` via `reply`. If we know that a message `reply` can be sent at most once and `reply` is received only by `reply(x)=>e`, we can transform the above fragment to: [8]

```
let fun k x = e in m(k) end
```

Thus, creation of the channel `reply` and message passing on the channel are eliminated. Moreover, consider the case where `e` is just message sending `n(x)`:

```
$reply.(m(reply) | reply(x)=>n(x))
```

It can be transformed to:

```
let fun k x = n x in m(k) end
```

Since `k(x)=n(x)`, it is simplified to:

```
m(n)
```

This kind of situations often occur in concurrent object-oriented programming. For example, consider the situation where an object A delegates a message to another object B, and receives a reply from B, and then forwards the reply to the sender of the original message. By the above optimization, the reply from B can be directly sent to the sender.

The last example looks quite similar to tail recursion elimination in continuation passing style compilation for functional languages[2]. But please note that the above optimization is possible only by analyzing communication behaviors between concurrent processes.

6 Limitations and refinements of the simple analysis

Although the simple version of our analysis gives much more useful information than Nielson's one[9], there are still several limitations. We address two major problems and discuss refinements to overcome those problems.

6.1 Polymorphism on Labels

The first problem is that too many channels are identified by unification during the type/effect reconstruction. For example, in the following expression,

```
$getx.$gety.$set.(point(1.0,0.0)(getx,gety,set)
                  | $reply.(getx(reply)) | $reply'(getx(reply')))
```

[8] Note that this transformation is possible because message send and function application is overloaded in HACL.

our analysis identifies channels `reply` and `reply'` and counts the number of messages sent to the channels as 2, although they are separate channels, and the number of messages sent for each channel is 1. In order to overcome this problem, we need to introduce some form of polymorphism on labels. Types are extended as follows:

$$\tau ::= \alpha \mid \tau_b \mid o^l \mid \tau_1 \xrightarrow{b} \tau_2 \mid \tau_1 \times \tau_2 \mid \forall l.\tau$$

where b ranges over effects and l ranges over a set of labels of communication channel. We often write $\forall l_1 \cdots l_n.\tau$ for $\forall l_1.\cdots\forall l_n.\tau$.

We can accordingly refine typing rules **(ET-abs)**, **(ET-app)**, **(ET-new)**, **(ET-rec)** and **(ET-com)** in $\mathcal{ETR}$ as follows:

(ET2-abs) $$\frac{EC, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash e : (\tau_2, b) \qquad \{l_1, \ldots, l_n\} \cap FL(EC, \mathcal{T}, \mathcal{L}) = \phi}{EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\forall l_1 \cdots l_n.(\tau_1 \xrightarrow{b} \tau_2), \epsilon)}$$

(ET2-app) $$\frac{EC, \mathcal{T}, \mathcal{L} \vdash e_1 : (\forall l_1 \cdots l_n.(\tau_1 \xrightarrow{b} \tau_2), \epsilon) \quad EC, \mathcal{T}, \mathcal{L} \vdash e_2 : (\tau_1', \epsilon) \quad \tau_1' =_{EC} [l_1'/l_1, \ldots, l_n'/l_n]\tau_1}{EC, \mathcal{T}, \mathcal{L} \vdash e_1 e_2 : ([l_1'/l_1, \ldots, l_n'/l_n]\tau_2, [l_1'/l_1, \ldots, l_n'/l_n]b)}$$

(ET2-rec) $$\frac{EC, \mathcal{T}\{f \mapsto \forall l_1 \cdots l_n.(\tau \xrightarrow{b} o)\}, \mathcal{L} \vdash \lambda x.e : (\forall l_1 \cdots l_n.(\tau \xrightarrow{b} o), \epsilon)}{EC, \mathcal{T}, \mathcal{L} \vdash \mathbf{rec}\ f(x) = e : (\forall l_1 \cdots l_n.(\tau \xrightarrow{b} o), \epsilon)}$$

(ET2-new) $$\frac{EC, \mathcal{T}\{x \mapsto \forall l_1 \cdots l_n.(\tau \xrightarrow{b'} o^l)\}, \mathcal{L} \vdash e : (o, b) \qquad EC \Rightarrow b' \geq \{l \mapsto (0,1)\} \qquad l \in \mathcal{L}(\pi)}{EC, \mathcal{T}, \mathcal{L} \vdash \$^{\pi} x.e : (o, b)}$$

(ET2-com)

$$\frac{\begin{array}{c} EC, \mathcal{T}, \mathcal{L} \vdash \lambda x.e : (\forall l_1 \cdots l_n.(\tau \xrightarrow{b} o), \epsilon) \\ EC, \mathcal{T}, \mathcal{L} \vdash m : (\forall l_1 \cdots l_n.(\tau \xrightarrow{b'} o^l), \epsilon) \\ EC \Rightarrow b' \geq minus(b, l) \end{array}}{EC, \mathcal{T}, \mathcal{L} \vdash m(x) => e : (o, \{l \mapsto (1,0)\})}$$

$FL(EC, \mathcal{T}, \mathcal{L})$, a set of *free labels* in $EC, \mathcal{T}, \mathcal{L}$ is defined by:

$$\begin{array}{l} FL(EC, \mathcal{T}, \mathcal{L}) = FL_{TA}(EC, \mathcal{T}) \cup FL_{LA}(EC, \mathcal{L}) \\ FL_{TA}(EC, \phi) = \phi \\ FL_{TA}(EC, \mathcal{T}\{x \mapsto \tau\}) = FL_{TA}(EC, \mathcal{T}) \cup FL_T(EC, \tau) \\ FL_T(EC, \alpha) = \phi \\ FL_T(EC, \tau_b) = \phi \\ FL_T(EC, o^l) = \{l\} \\ FL_T(EC, \tau_1 \xrightarrow{b} \tau_2) = FL_T(EC, \tau_1) \cup FL_T(EC, \tau_2) \cup FL_E(EC, b) \\ FL_T(EC, \tau_1 \times \tau_2) = FL_T(EC, \tau_1) \cup FL_T(EC, \tau_2) \\ FL_T(EC, \forall l.\tau) = FL_T(EC, \tau) - \{l\} \\ FL_E(EC, \epsilon) = \phi \\ FL_E(EC, \rho) = FL_{EV}(EC, \rho) \\ FL_E(EC, b_1 + b_2) = FL_E(EC, b_1) \cup FL_E(EC, b_2) \\ FL_E(EC, b_1 \cup b_2) = FL_E(EC, b_1) \cup FL_E(EC, b_2) \\ FL_E(EC, minus(b, l)) = FL_E(EC, b) \cup \{l\} \\ FL_E(EC, \{l_1 \mapsto (I_1, O_1), \ldots, l_n \mapsto (I_n, O_n)\}) = \{l_1, \ldots, l_n\} \\ FL_{LA}(EC, \mathcal{L}) = \cup\{L \mid \mathcal{L}(\pi) = L \text{ for some } \pi\} \end{array}$$

$FL_{EV}(EC, \rho)$ is defined as the least set S_ρ which satisfies the following set of relations:

$$\{S_{\rho_i} \supseteq S_{\rho_1'} \cup \cdots \cup S_{\rho_n'} \cup \{l_1, \ldots, l_m\} \mid b_1 \geq b_2 \in EC,$$
$$\rho_i \text{ appears in } b_1,$$
$$\{\rho_1', \ldots, \rho_n'\} \text{ is a set of all effect variables in } b_2,$$
$$\text{and } \{l_1, \ldots, l_m\} \text{ is a set of all labels in } b_2.\}$$

For example, if $EC = \{\rho_1 \geq \rho_2 + \{l_1 \mapsto (1,0)\}, \rho_2 \geq \{l_2 \mapsto (0,1)\}\}$, then $FL_{EV}(EC, \rho_1) = \{l_1, l_2\}$. Thus, we say that l_1, l_2 are NOT free in $EC, \{x \mapsto \tau_1 \xrightarrow{\rho_1} \rho_2\}, \mathcal{L}$.

Let $\mathcal{ETR}_2$ be a refined set of rules. Then, it is not difficult to check that the subject reduction theorem still holds.

Proposition 6.1 (subject reduction theorem)
If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}_2} e : (o^l, b)$ *and* $e \longrightarrow e'$, *then* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}_2} e' : (o^{l'}, b)$.

Type reconstruction becomes a little complicated as a result, and the effect constraint $\vec{\rho} \geq \vec{b}_{EC}(\vec{\rho})$ may contain renaming maps ψ on labels. However, if we restrict ψ to be injective, then the effect constraint is again written in the form: $\vec{\rho} \geq \vec{b}_{EC}(\vec{\rho})$, where $\vec{b}$ may contain renaming maps. Since ψ and its inverse map ψ^{-1} are also monotonic, the effect constraint is still solvable in finite steps.

6.2 Channel Creation in Recursion

The second problem is related to channel creations in recursive processes. With the rule **(ET2-rec)**, if channels are recursively created in an expression **rec** $f(x) = e$, all effects for them are added, although each channel is actually separate. Thus, we can obtain only rough information for channels created in recursion. The fundamental difficulty lies in this problem, because we must deal with infinite labels. In order to overcome this problem, we introduce new kinds of variables $\chi_1, \chi_2, \ldots$ for representing a set of (possibly infinite) labels. Effect constants are extended as follows:

$$\{\chi_1 \mapsto (I_1, O_1), \ldots, \chi_m \mapsto (I_m, O_m), l_1 \mapsto (I'_1, O'_1), \ldots, l_n \mapsto (I'_n, O'_n)\}$$

Intuitively, $\{\chi \mapsto (I_1, O_1)\}$ represents a constant effect $\cup\{\{l \mapsto (I_1, O_1)\} \mid l \in \chi\}$. We also extend a label assignment to a map from each program point to a set consisting of labels and these new variables. We introduce the following new rule for recursion.

(ET3-rec)

$$\frac{\begin{array}{c} EC', \mathcal{T}'\{f \mapsto \forall l_1 \cdots l_n.(\tau \xrightarrow{b} o)\}\mathcal{L}' \vdash \lambda x.e : ([(\chi \cup L_1)/\chi](\forall l_1 \cdots l_n.(\tau \xrightarrow{b} o)), \epsilon) \\ EC', \mathcal{T}', \mathcal{L}' \equiv [(\chi \cup L_1)/\chi](EC, \mathcal{T}, \mathcal{L}) \\ (L_1 \cup L_2) \cap FL(EC, \mathcal{T}\{f \mapsto \forall l_1 \cdots l_n.(\tau \xrightarrow{b} o)\}, \mathcal{L}) = \phi \end{array}}{[(\chi \cup L_2)/\chi](EC, \mathcal{T}, \mathcal{L}) \vdash \textbf{rec } f(x) = e : (\forall l_1 \cdots l_n.\tau \xrightarrow{b} o), \epsilon)}$$

where L_1, L_2 denotes a finite set of labels. Intuitively, a variable χ in the above rule represents infinite labels of recursively created channels at the program point π, and L_1 is a set of labels of channels created in each expansion of the recursive definition. Let $\mathcal{ETR}_3$ be a set of rules obtained from $\mathcal{ETR}_2$ by replacing **(ET2-rec)** with **(ET3-rec)**. The subject reduction theorem need to be modified as follows.

Proposition 6.2 (subject reduction theorem)
If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}_3} e : (o^l, b)$ *and* $e \longrightarrow e'$, *then there are* $EC', \mathcal{T}', \mathcal{L}', b', L_1, L_2, L_3, L_4$ *such that*

- $EC, \mathcal{T}, \mathcal{L} \equiv [(\chi \cup L_1)/\chi](EC', \mathcal{T}', \mathcal{L}')$
- $b \equiv [(\chi \cup L_2)/\chi]b'$
- $[(\chi \cup L_3)/\chi](EC', \mathcal{T}', \mathcal{L}') \vdash_{\mathcal{ETR}_3} e : (o^{l'}, [(\chi \cup L_4)/\chi]b')$
- $(L_1 \cup L_2 \cup L_3 \cup L_4) \cap (FL(EC', \mathcal{T}', \mathcal{L}') \cup FL_E(EC, b')) = \phi$

From the above subject reduction, if $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}_3} e : (o, b)$, then $max(\{b(x) \mid x \in \mathcal{L}(\pi)\})$ is invariant during the reduction. Therefore, the following corollary still holds.

Figure 9: A process `pipe`

Corollary 6.3 *If* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}_3} e : (o, \{\chi_1 \mapsto (I_1, O_1), \ldots, \chi_m \mapsto (I_m, O_m), l_1 \mapsto (I'_1, O'_1), \ldots, l_n \mapsto (I'_n, O'_n)\}$ *and* $\mathcal{L}(\pi) = \{\chi_1, \ldots, \chi_m, l_1, \ldots, l_n\}$*, then during the evaluation of* e*, the number of enqueued messages sent to the communication channel created at the program point* π *is at most* $max(\{O_1, \ldots, O_m, O'_1, \ldots, O'_n\})$ *and the number of waiting processes for the channel is at most* $max(\{I_1, \ldots, I_m, I'_1, \ldots, I'_n\})$*.*

Example Consider the following process:

```
let proc node(f, from, to) = from(x)=>(to(f(x)) | node(f,from,to))
in
  let proc pipe (f, n, from, to) =
    if n=0 then from(x)=>to(x)
    else $m.(node(f, from, m) | pipe(f,n-1,m,to))
  in
    $m1.$m2.(m1(0) | pipe(fn x=>x+1, 10, m1, m2))
  end
end
```

The process `pipe(f, n, from, to)` receives a value along the channel `from`, applies `f` to it `n` times, and outputs the result along the channel `to`, as is illustrated in Figure 9. In $\mathcal{ETR}_3$, during the reduction of the above program, we can analyze that the number of enqueued messages for each channel `m`, `m1`, `m2` is at most one.

7 Related Work

Formal static analysis methods proposed so far for concurrent languages have been very limited. Although several work has been done on I/O mode checking/inference[11][6], as far as the author knows, only Nielson's techniques[9][8][10] for Concurrent ML[13] can analyze more useful information such as the number of input/output operations performed. However, as is briefly discussed in Section 2, information obtained by Nielson's methods, especially for recursive processes, seem too rough for various compile-time optimizations including the ones described in Section 5. Although our method is initially inspired by their work[9], it differs from theirs in several essential points (especially, the rule **(ET-com)** for communication), and as a result it gives a more precise information than theirs.

Nielson's analysis focuses on the total number of the input and output operations through each communication channel, while our primal goal is to infer not only the total number of those operations but also the number of the enqueued messages and receivers. However, even if we ignore this difference in the goal, there are the following two essential differences. First of all, Nielson's analysis cannot obtain much useful information for recursive programs. As is briefly discussed in Section 2, the number of input and output operations performed in recursive processes is always counted as ∞. On the other hand, our analysis counts the number of input and output operations as ∞ *only when* the process may be really infinitely invoked. More precisely, in Nielson's analysis, the effect of $m(x) => e$ is counted as one input operation from the channel m *plus* the effect of e. Thus, for example, if a process is defined recursively as follows:

```
proc p() = m(x)=> (x() | p())
```

then the number of input operations from the channel `m` is counted as ∞. On the other hand, our analysis counts the effect of $m(x) => e$ as just single input operation from the channel m.

Instead, the effect to execute e is considered as the effect that occurs when a value is sent along the channel m. Thus, even if we define a process recursively, the effect of the process is not necessarily counted as ∞. The second difference, which is related to the first difference, is (a restricted form of) polymorphism introduced in Section 6. If there is a receiver $m(x) => e$, then sending a value v along the channel m causes $e[v/x]$ to be executed. Thus the resulting effect should depend on v. This is why we count the effect to execute e not in $m(x) => e$ but in a message send operation $m(v)$. In Nielson's analysis, the effect to execute e is counted in $m(x) => e$, independent of which channel will be sent along the channel m. Thus there is no room for introducing this form of polymorphism in their analysis.

8 Conclusion

We proposed a static analysis technique on message passing for asynchronous concurrent programming languages, and discussed optimizations based on the proposed analysis. We have already implemented and tested the proposed static analysis algorithm, and are now trying to integrate it into compiler of HACL. Future work includes refinements of the analysis technique, and an application to the compilers of concurrent object-oriented languages. As for the refinement, introduction of subtyping[16] to effect systems seems promising. We expect that subtyping can be introduced for our analysis by taking I/O mode of channels into account.

References

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

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

[3] Kobayashi, N., and A. Yonezawa, "Asynchronous Communication Model Based on Linear Logic." to appear in Journal of Formal Aspects of Computing, Springer-Verlag.

[4] Kobayashi, N., and A. Yonezawa, "Towards Foundations for Concurrent Object-Oriented Programming – Types and Language Design." submitted.

[5] Kobayashi, N., and A. Yonezawa, "ACL – A Concurrent Linear Logic Programming Paradigm," in *Logic Programming: Proceedings of the 1993 International Symposium*, pp. 279–294, MIT Press, 1993.

[6] Kobayashi, N., and A. Yonezawa, "Higher-Order Concurrent Linear Logic Programming." Invited talk at Theory and Practice of Parallel Programming (TPPP'94), Sendai, Japan (Proceedings will appear in Springer Lecture Notes in Computer Science), November 1994.

[7] Kobayashi, N., and A. Yonezawa, "Type-Theoretic Foundations for Concurrent Object-Oriented Programming," in *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pp. 31–45, 1994.

[8] Nielson, F., and H. R. Nielson, "Constraints for Polymorphic Behaviors of Concurrent ML," in *Proceedings of CCL'94*, vol. 845 of *Lecture Notes in Computer Science*, pp. 73–88, Springer Verlag, 1994.

[9] Nielson, H. R., and F. Nielson, "Higher-Order Concurrent Programs with Finite Communicationn Topology," in *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Language*, pp. 84–97, 1994.

[10] Nielson, H. R., and F. Nielson, "Static and Dynamic Processor Allocation for Higher-Order Concurrent Languages." Technical Report DAIMI PB-483, Dept. of Comp. Science, Aarhus University, 1994.

[11] Pierce, B., and D. Sangiorgi, "Typing and Subtyping for Mobile Processes," in *Proceedings of IEEE Symposium on Logic in Computer Science*, pp. 376–385, 1993.

[12] Pierce, B. C., "Programming in the Pi-Calculus: An Experiment in Programming Language Design." Lecture notes for a course at the LFCS, University of Edinburgh., 1993.

[13] Reppy, J. H., "CML: A Higher-order Concurrent Language," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 293–305, 1991.

[14] Shapiro, E., "Concurrent Prolog: A Progress Report," in *Concurrent Prolog* (E. Shapiro, ed.), vol. 1, pp. 157–187, The MIT Press, 1987.

[15] Talpin, J.-P., and P. Jouvelot, "The Type and Effect Discipline," in *Proceedings of IEEE Symposium on Logic in Computer Science*, pp. 162–173, 1992.

[16] Tang, Y.-M., and P. Jouvelot, "Effect systems with subtyping," in *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (to appear)*, 1995.

[17] Taura, K., S. Matsuoka, and A. Yonezawa, "An Efficient Implementation Scheme of Concurrent Object-Oriented Language on Stock Multicomputers," in *Proc. ACM Conf. on Principles and Practice of Parallel Programming (PPOPP)*, 1993.

[18] Ueda, K., "Guarded Horn Clauses," in *Concurrent Prolog* (E. Shapiro, ed.), vol. 1, pp. 140–156, The MIT Press, 1987.

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

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

[9]

[9] Papers by Naoki Kobayashi are available via anonymous ftp from camille.is.s.u-tokyo.ac.jp:pub/papers.

Appendix

A Proof of Subject Reduction Theorem

We need the following three lemmas.

Lemma A.1 (substitution lemma) *If* $EC, \mathcal{T}\{x_1 \mapsto \tau_1, \cdots, x_n \mapsto \tau_n\}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$ *and* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_i : (\tau_i, \epsilon)(1 \leq i \leq n)$, *then* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e[e_1/x_1, \ldots, e_n/x_n] : (\tau, b)$.

Proof sketch Induction on the structure of the proof for $EC, \mathcal{T}\{x_1 \mapsto \tau_1, \cdots, x_n \mapsto \tau_n\}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$. □

Lemma A.2 *If* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$ *and* $e \cong e'$, *then* $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (\tau, b)$.

Proof sketch We show a slightly stronger property:

> If $e \cong e'$, then $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$ if and only if $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (\tau, b)$.

We show by induction on the structure of the proof for $e \cong e'$. Case analysis on the last rule. **(Cong-1)**,**(Cong-2)** **(Cong-3)**,**(Cong-4)** and **(Cong-6)** are trivial.

- If the last rule is one for congruence:

  $$\frac{e_1 \cong e_2}{C[e_1] \cong C[e_2]}$$

  This case is trivial by induction hypothesis.

- **(Cong-5)**
  ($\Rightarrow$) Suppose that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} \$^{\pi} x.(p_1|p_2) : (\tau, b)$ and x is not free in p_1. By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} \$^{\pi} x.(p_1|p_2) : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-new)** and **(ET-par)**,

  $$\begin{array}{l} EC, \mathcal{T}\{x \mapsto \tau_1 \xrightarrow{b_1} o^l\}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1 : (o, b2) \\ EC, \mathcal{T}\{x \mapsto \tau_1 \xrightarrow{b_1} o^l\}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_2 : (o, b3) \\ EC \Rightarrow b_1 \geq \{l \mapsto (0,1)\} \\ l \in \mathcal{L}(\pi) \\ \tau \equiv o \\ b' \equiv b_2 + b_3 \end{array}$$

  Since x is not free in p_1, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1 : (o, b2)$ holds. Therefore, we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1|\$^{\pi} x.p_2 : (\tau, b')$. By **(ET-weak)**,

  $$EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} p_1|\$^{\pi} x.p_2 : (\tau, b).$$

  ($\Leftarrow$) Suppose that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} p_1|\$^{\pi} x.p_2 : (\tau, b)$ and x is not free in p_1. By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1|\$^{\pi} x.p_2 : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-new)** and **(ET-par)**,

  $$\begin{array}{l} EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1 : (o, b2) \\ EC, \mathcal{T}\{x \mapsto \tau_1 \xrightarrow{b_1} o^l\}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_2 : (o, b3) \\ EC \Rightarrow b_1 \geq \{l \mapsto (0,1)\} \\ l \in \mathcal{L}(\pi) \\ \tau \equiv o \\ b' \equiv b_2 + b_3 \end{array}$$

From $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1 : (o, b2)$, $EC, \mathcal{T}\{x \mapsto \tau_1 \xrightarrow{b_1} o^l\}, \mathcal{L} \vdash_{\mathcal{ETR}'} p_1 : (o, b2)$ also holds. Therefore, we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} \$^{\pi} x.(p_1|p_2) : (\tau, b')$. By **(ET-weak)**,

$$EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} \$^{\pi} x.(p_1|p_2) : (\tau, b).$$

□

Lemma A.3 (subject reduction on $\longrightarrow_\lambda$) If $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$ and $e \longrightarrow_\lambda e'$, then $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (\tau, b)$.

Proof sketch Suppose that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (o, b)$ and $e \longrightarrow_\lambda e'$. We show $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (o, b)$ by induction on the structure of the proof for $e \longrightarrow_\lambda e'$. Case analysis on the last rule.

- **(R-Beta)**
  In this case, $e \equiv (\lambda x.e_1)v$ and $e' \equiv e_1[v/x]$. By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-app)** and **(ET-abs)**,

  $$EC, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash_{\mathcal{ETR}'} e_1 : (\tau, b')$$
  $$EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} v : (\tau_1, \epsilon)$$

  By Lemma A.1, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e[v/x] : (\tau, b')$. By **(ET-weak)**, we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e[v/x] : (\tau, b)$.

- **(R-Let2)**
  This case is trivial from **(ET-let)**.

- **(R-Rec)**
  In this case, $e \equiv \mathbf{rec}\ f(x) = e_1$ and $e' \equiv \lambda x.e_1[(\mathbf{rec}\ f(x) = e_1)/f]$. By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-rec)**, $EC, \mathcal{T}\{f \mapsto \tau\}, \mathcal{L} \vdash_{\mathcal{ETR}'} \lambda x.e_1 : (\tau, \epsilon)$. By substitution lemma, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} \lambda x.e_1[(\mathbf{rec}\ f(x) = e_1)/f] : (\tau, \epsilon)$. By **(ET-weak)**, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} \lambda x.e_1[(\mathbf{rec}\ f(x) = e_1)/f] : (\tau, b)$.

- Other cases are trivial by induction hypothesis.

□

Proof of Proposition 3.2 Suppose that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (o, b)$ and $e \longrightarrow e'$. We show $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (o, b)$ by induction on the structure of the proof for $e \longrightarrow e'$. Case analysis on the last rule.

- **(R-Com)**
  In this case, $e \equiv ((m(x) => e) \mid m(v))$, and $e' \equiv e_1[v/x]$. By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-par)**, from some b_1, b_2, $b' = b_1 + b_2$, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} m(x) => e : (o, b_1)$ and $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} m(v) : (o, b_2)$ hold. From $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} m(x) => e$ and **(ET-com)**, we also have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} m : (\tau \xrightarrow{b_2} o^l, \epsilon)$, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} \lambda x.e : (\tau \xrightarrow{b_2} o, \epsilon)$. By **(ET-abs)**, we have $EC, \mathcal{T}\{x \mapsto \tau\}, \mathcal{L} \vdash_{\mathcal{ETR}'} e : (o, b_3)$ and $EC \Rightarrow b_2 \geq b_3$. By Proposition 4.1, $EC, \mathcal{T}\{x \mapsto \tau\}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (o, b_3)$. By Lemma A.1, we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e[v/x] : (o, b_3)$. From conditions $EC \Rightarrow b_2 \geq b_3$, $b' = b_1 + b_2$ and $EC \Rightarrow b \geq b'$, $EC \Rightarrow b \geq b_3$ holds. Therefore, by applying **(ET-weak)**, we obtain:

  $$EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e[v/x] : (o, b).$$

- **(R-Cong)**
  Trivial from Lemma A.2.

- **(R-Lambda)**
  Trivial from Lemma A.3.

- **(R-Par)**
  In this case, $e \equiv e_1|e_2$, $e' \equiv e_1|e_2'$, and $e_2 \longrightarrow e_2'$ By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} (e_1|e_2) : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-par)** and Proposition 4.1, for some b_1 and b_2, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_1 : (o, b_1)$ and $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_2 : (o, b_2)$. By I.H., we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_2' : (o, b_2)$. Therefore, by **(ET-par)**, we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_1|e_2' : (o, b')$. By **(ET-weak)**, we can obtain

  $$EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_1|e_2' : (o, b).$$

- **(R-Choice)** In this case, $e \equiv (e_1 \& e_2)|e_3$, and $e_1|e_3 \longrightarrow e'$ By Proposition 4.1, there is b' such that $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}'} (e_1 \& e_2)|e_3 : (\tau, b')$ and $EC \Rightarrow b \geq b'$. By **(ET-par)**, **(ET-choice)** and Proposition 4.1, for some b_1, b_2 and b_3, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_1 : (o, b_1)$, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_2 : (o, b_2)$, $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e_3 : (o, b_3)$, and $b' = (b_1 \cup b_2) + b_3$. By I.H., we have $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (o, b_1 + b_3)$. From conditions $b' = (b_1 \cup b_2) + b_3$ and $EC \Rightarrow b \geq b'$, we have $EC \Rightarrow b \geq b_1 + b_3$. Therefore, by **(ET-weak)**, we can obtain $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e' : (o, b)$.

- **(R-New)** Trivial from the rule **(ET-new)**.

- **(R-Inaction)** Trivial from that _ is a constant of type o and the rule **(ET-const)**.

B Proof sketch of Proposition 4.5

Induction on the structure of proof for $EC, \mathcal{T}, \mathcal{L} \vdash_{\mathcal{ETR}} e : (\tau, b)$. Case analysis on the last rule used. We show only one case in this paper. The other cases are shown in the similar manner.

(ET-app) In this case, $e \equiv e_1 e_2$, and

$$EC', \mathcal{T}', \mathcal{L}' \vdash e_1 : \tau_1' \xrightarrow{b'} \tau' \& \epsilon$$
$$EC', \mathcal{T}', \mathcal{L}' \vdash e_2 : \tau_1' \& \epsilon$$

Let

$$PT(e_1, \phi) = (EC_1, \mathcal{T}_1, \mathcal{L}_1, u_1, b_1)$$
$$PT(e_2, \phi) = (EC_2, \mathcal{T}_2, \mathcal{L}_2, u_2, b_2)$$

By induction hypothesis, for some $S_1, \Phi_1, \psi_1, S_2, \Phi_2, \psi_2$,

$EC' \Rightarrow \Phi_1 \psi_1 EC_1$	$EC' \Rightarrow \Phi_2 \psi_2 EC_2$
$S_1 \Phi_1 \psi_1 \mathcal{T}_1 \subseteq_{EC'} \mathcal{T}'$	$S_2 \Phi_2 \psi_2 \mathcal{T}_2 \subseteq_{EC'} \mathcal{T}'$
$\psi_1 \mathcal{L}_1 \subseteq \mathcal{L}'$	$\psi_2 \mathcal{L}_2 \subseteq \mathcal{L}'$
$S_1 \Phi_1 \psi_1 u_1 =_{EC'} \tau_1' \xrightarrow{b'} \tau'$	$S_2 \Phi_2 \psi_2 u_2 =_{EC'} \tau'$

hold.

Let

$$\psi_3 = \psi_1 \circ \psi_2,\ \Phi_3 = \Phi_1 \circ \Phi_2\{\rho \mapsto b'\},\ S_3 = S_1 \circ S_2 \circ \{\alpha \mapsto \tau'\}$$

Then, the following conditions hold:

$$EC' \Rightarrow \Phi_3 \psi_3 EC_1 \cup \Phi_3 \psi_3 EC_2$$
$$S_3 \Phi_3 \psi_3 \mathcal{T}_1 \cup S_3 \Phi_3 \psi_3 \mathcal{T}_2 \subseteq \mathcal{T}'$$
$$\psi_3 \mathcal{L}_1 + \psi_3 \mathcal{L}_2 \subseteq \mathcal{L}'$$
$$S_3 \alpha = \tau'$$
$$\Phi_3 \psi_3 \rho = b'$$

Since (S_3, Φ_3, ψ_3) is a unifier of $\{\tau_1 = \tau_2 \mid \mathcal{T}_1(x) = \tau_1, \mathcal{T}_2(x) = \tau_2\} \cup \{u_1 = \tau_1' \xrightarrow{\rho} \alpha\}$ and (S, Φ, ψ) is the most general unifier, there exists (S', Φ', ψ') such that

$$(S', \Phi', \psi') \circ (S, \Phi, \psi) = (S_3, \Phi_3, \psi_3)$$

Therefore,

$$EC_1 \Rightarrow \Phi'\psi'(\Phi\psi EC_1 \cup \Phi\psi EC_2))$$
$$S'\Phi'\psi'(S\Phi\psi\mathcal{T}_1 \cup S\Phi\mathcal{T}_2)) \subseteq \mathcal{T}'$$
$$\psi'(\psi\mathcal{L}_1 + \psi\mathcal{L}_2) \subseteq \mathcal{L}'$$
$$S'\Phi'\psi'(S\Phi\psi\alpha) = \tau'$$
$$\Phi'\psi'(\Phi\psi\rho) = b'$$

Hence,

$$(\Phi\psi EC_1 \cup \Phi\psi EC_2, S\Phi\psi\mathcal{T}_1 \cup S\Phi\psi\mathcal{T}_2, \psi\mathcal{L}_1 + \psi\mathcal{L}_2, S\Phi\psi\alpha, \Phi\psi\rho)$$

is the principal typing of $e \equiv e_1 e_2$.

□