Resource Usage Analysis

Atsushi Igarashi
Department of Graphics and Computer Science
Graduate School of Arts and Sciences
University of Tokyo
igarashi@graco.c.u-tokyo.ac.jp

Naoki Kobayashi
Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
kobayasi@cs.titech.ac.jp

ABSTRACT

It is an important criterion of program correctness that a program accesses resources in a valid manner. For example, a memory region that has been allocated should be eventually deallocated, and after the deallocation, the region should no longer be accessed. A file that has been opened should be eventually closed. So far, most of the methods to analyze this kind of property have been proposed in rather specific contexts (like studies of memory management and verification of usage of lock primitives), and it was not so clear what is the essence of those methods or how methods proposed for individual problems are related. To remedy this situation, we formalize a general problem of analyzing resource usage as a *resource usage analysis problem*, and propose a type-based method as a solution to the problem.

1. INTRODUCTION

It is an important criterion of program correctness that a program accesses resources in a valid manner. For example, a memory cell that has been allocated should be eventually deallocated, and after the deallocation, the cell should not be read or updated. A file that has been opened should be eventually closed. A lock should be acquired before a shared resource is accessed. After the lock has been acquired, it should be eventually released.

A number of program analyses have been proposed to ensure such a properly. Type systems for region-based memory management [1, 3, 25, 28] ensure that deallocated regions are no longer read or written. Linear type systems [18, 26, 27, 30] ensure that a linear (use-once) value that has been already accessed is never accessed again. Abadi and Flanagan's type systems for race detection [7, 8] ensure that appropriate locks will be acquired before a reference cell or a concurrent object is accessed. Freund and Mitchell's type system [9] for JVM ensures that every object is initialized before it is accessed. Bigliardi and Laneve's type system [2] for JVM ensures that an object that has been locked will be eventually unlocked. DeLine and Fähndrich's type system [5] keeps track of the state of each resource in order to control access to the resource.

The problems attacked in the above-mentioned pieces of work are similar: There are different types of primitives to access resources (initialization, read, write, deallocation, etc.) and we want to ensure that those primitives are applied in a valid order. In spite of such similarity, however, most of the solutions (except for DeLine and Fähndrich's work [5]) have been proposed for specific problems. As a result, solutions are often rather ad hoc, and it is not clear how they can be applied to other similar problems and how solutions for different problems are related. This is in contrast with standard program analysis problems like flow analysis: For the flow analysis problem, there is a standard definition and there are several standard methods, whose properties (computational cost, precision, etc.) are well studied.

Based on the observation above, our aims are:

1. To formalize a general problem of analyzing how each resource is accessed as a *resource usage analysis problem* (usage analysis problem, in short[1]), to make it easy to relate existing methods and to stimulate further studies of the problem.

2. To propose a type-based method for usage analysis. Unlike DeLine and Fähndrich's type system [5], our type-based analysis does not need programmers' type annotation to guide the analysis. Our analysis automatically gathers information about how resources are accessed, and checks whether it matches the programmer's intention.

We give an overview of each point below.

1.1 Resource Usage Analysis Problem

We formalize a resource usage analysis problem in a manner similar to a formalization of the flow analysis problem [23]. Suppose that each expression of a program is annotated with a label, and let $\mathcal{L}$ be the set of labels. The standard flow analysis problem for λ-calculus is to obtain a function $\mathit{flow} \in \mathcal{L} \to 2^{\mathcal{L}}$ ($2^{\mathcal{L}}$ denotes the powerset of $\mathcal{L}$) where $\mathit{flow}(l) = \{l_1, \ldots, l_n\}$ means that an expression labeled with l evaluates to a value generated by an expression labeled with one of $l_1, \ldots, l_n$. (Or, equivalently, the problem is to obtain a function $\mathit{flow}^{-1} \in \mathcal{L} \to 2^{\mathcal{L}}$ where

[1]The term "usage analysis" is also used to refer to linearity analysis [11]. Our resource usage analysis problem can be considered generalization of the problem of linearity analysis.

POPL'02 1/02, Portland, Oregon, USA

$\mathit{flow}^{-1}(l) = \{l_1, \ldots, l_n\}$ means that only expressions labeled with $l_1, \ldots, l_n$ can evaluate to the value generated by an expression labeled with l.) From a flow function, we know what access may occur to each resource. For example, consider the following fragment ML-like program:

$$\textbf{let } x = \textbf{fopen}(s)^l \textbf{ in} \ldots \textbf{fread}(M^{l_r}) \ldots \textbf{fclose}(N^{l_c}) \ldots$$

Here, we assume that **fopen** opens a file of name s and returns a file pointer to access the file, and that **fread** (**fclose**, resp.) takes a file pointer as an input and reads (closes, resp.) the file. If $\mathit{flow}^{-1}(l) = \{l_r\}$, then we know that the file opened at l may be read, but is not closed (since expression N^{l_c} cannot evaluate to the file by the definition of flow^{-1}).

A flow function does not provide information about the order of resource accesses. Suppose that $flow^{-1}(l)$ is $\{l_c, l_r\}$ in the above program. From the flow information, we can't tell whether the file created at l is closed after it has been read, or the file is read after it has been closed.

Let us write $\mathcal{L}^*$ for the set of finite sequences of labels. We formalize *usage analysis* as a problem of (i) computing a function $\mathit{use} \in \mathcal{L} \rightarrow 2^{\mathcal{L}^*}$ where $l_1 \cdots l_n \in \mathit{use}(l)$ means that a value generated by an expression labeled with l may be accessed by primitives labeled with $l_1, \ldots, l_n$ in this order, and then (ii) checking whether $\mathit{use}(l)$ contains only valid access sequences. Let us reconsider the above example:

$$\textbf{let } x = \textbf{fopen}^l(s) \textbf{ in} \ldots \textbf{fread}^{l_r}(M) \ldots \textbf{fclose}^{l_c}(N) \ldots$$

(Here, labels are moved to primitives for creating or accessing resources.) If $\mathit{use}(l) = \{l_r l_c, l_r l_r l_c\}$, we know that the file opened at l may be closed after it is read once or twice, and the file is never read after being closed. On the other hand, if $\mathit{use}(l) = \{l_r l_c, l_c l_r\}$, the file may be read after it has been closed.

Many problems can be considered instances of the usage analysis problem. In region-based memory management [25, 3, 1, 28], we can regard regions as resources. Suppose that every primitive for reading a value from a region (writing a value into a region, deallocating a region, resp.) is annotated with l_r (l_w, l_F, resp.). Then, a region-annotated program is correct if $\mathit{use}(l) \subseteq (l_r + l_w)^* l_F$, where $(l_r + l_w)^* l_F$ is a regular expression. In linear type systems [27, 26, 18, 30], we can regard values as resources. A linear type system is correct if for every label l of a primitive for creating linear (use-once) values, $\mathit{use}(l)$ contains only sequences of length 1. The object initialization is correct [9] if for every label l of an (occurrence of) object creation primitive, every sequence in $\mathit{use}(l)$ begins with the label of a primitive for object initialization. Usage of lock primitives is correct if each occurrence of a label of the lock primitive is followed by an occurrence of a label of the unlock primitive. The control flow analysis problem can also be considered an instance of the usage analysis problem. We can regard functions as resources, function abstraction as the primitive for creating a function, and function application as the primitive for accessing a function. Then, a function created at l may be called at l' if $\mathit{use}(l)$ contains l'.

1.2 Type-Based Usage Analysis

We present a type-based resource usage analysis for a call-by-value, simply-typed λ-calculus extended with primitives for creating and accessing resources.

The main idea is to augment types with information about a resource access order. For example, the type of a file is written as $(\textbf{File}, U)$, where U, called a *usage*, expresses how the file is accessed. Its syntax is given by:

$$U ::= l \mid U_1\,; U_2 \mid U_1 \,\&\, U_2 \mid \cdots$$

Usage l means that the resource is accessed by a primitive labeled with l. $U_1\,; U_2$ means that the resource is accessed according to U_1 and then accessed according to U_2. $U_1 \,\&\, U_2$ means that the resource is accessed according to either U_1 or U_2. For example, a file that is accessed by a primitive labeled with l_1 and then by a primitive labeled with l_2 has type $(\textbf{File}, l_1\,; l_2)$.

Based on the extension of types with usages, we extend ordinary typing rules for the simply-typed λ-calculus. For example, the ordinary rule for let-expressions is:

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma, x : \tau \vdash N : \sigma}{\Gamma \vdash \textbf{let } x = M \textbf{ in } N : \sigma}$$

It is replaced by the following rule:

$$\frac{\Gamma \vdash M : \tau \qquad \Delta, x : \tau \vdash N : \sigma}{\Gamma; \Delta \vdash \textbf{let } x = M \textbf{ in } N : \sigma}$$

Type environment $\Gamma; \Delta$ indicates that the resources referred to by free variables are first accessed according to Γ and then according to Δ, reflecting the evaluation rule that M is evaluated and then N is evaluated. For example, if we have $y : (\textbf{File}, l_1) \vdash M : \textbf{bool}$ (which implies that y is a file accessed at l_1 in M) and $y : (\textbf{File}, l_2), x : \textbf{bool} \vdash N : \textbf{bool}$, then we get $y : (\textbf{File}, l_1; l_2) \vdash \textbf{let } x = M \textbf{ in } N : \textbf{bool}$. The resulting type environment indicates that y is a file accessed at l_1 and then at l_2.

Actually, the type system is a little more complicated than it might seem. Consider an expression $M \stackrel{\triangle}{=} \textbf{let } x = y \textbf{ in } (\textbf{fread}^{l_r}(y); \textbf{fwrite}^{l_w}(x))$. If we naively apply the above rule, we get:

$$\frac{\begin{array}{c} y : (\textbf{File}, l_w) \vdash y : (\textbf{File}, l_w) \\ y : (\textbf{File}, l_r), x : (\textbf{File}, l_w) \vdash \textbf{fread}^{l_r}(y); \textbf{fwrite}^{l_w}(x) : \textbf{bool} \end{array}}{(y : (\textbf{File}, l_w)); (y : (\textbf{File}, l_r))(= y : (\textbf{File}, l_w; l_r)) \vdash M : \textbf{bool}}$$

The conclusion implies that y is first written at l_w and then read at l_r, which is wrong. This wrong reasoning comes from the fact that the access represented by the type environment $y : (\textbf{File}, l_w)$ occurs not when y is evaluated but when the body of the let-expression $\textbf{fread}^{l_r}(y); \textbf{fwrite}^{l_w}(x)$ is evaluated. To solve this problem, we introduce a usage constructor $\Box U$, which means that the resource is accessed according to U *now* (when the expression is evaluated) or *later* (when the value of the expression is used). Using the operator $\Box$, we replace the above inference with:

$$\frac{\begin{array}{c} y : (\textbf{File}, \Box l_w) \vdash y : (\textbf{File}, l_w) \\ y : (\textbf{File}, l_r), x : (\textbf{File}, l_w) \vdash \textbf{fread}^{l_r}(y); \textbf{fwrite}^{l_w}(x) : \textbf{bool} \end{array}}{y : (\textbf{File}, \Box l_w; l_r) \vdash M}$$

The premise $y : (\textbf{File}, \Box l_w) \vdash y : (\textbf{File}, l_w)$ reflects the fact that the resource y is accessed only when the value of y is used later (when $\textbf{fwrite}^{l_w}(x)$ is evaluated). The conclusion implies that that y may be accessed at l_w either immediately before an access at l_r occurs, or later after an access at l_r occurs. (In order to obtain a more accurate usage $l_r; l_w$, we need to keep dependencies between different variables: See Section 6.)

In order to get accurate information about the access order, we also need to have a rule to remove $\Box$. Suppose that $x : (\mathbf{File}, \Box l) \vdash M : \tau$ is derived and that we know that the value (evaluation result) of M cannot contain a reference to x. Then, we know that x is accessed at l when M is evaluated, *not later.* To allow such reasoning, we introduce the following rule:

$$\frac{\Gamma, x : \tau \vdash M : \sigma \qquad x \text{ does not escape from } M}{\Gamma, x : \blacksquare\tau \vdash M : \sigma}$$

Here, $\blacksquare$ is an operator to cancel the $\Box$-operator.

Based on the above idea, we formalize a type system for usage analysis and prove its correctness. We also develop a type inference algorithm to infer resource usage information automatically so that programmers only have to declare what access sequences are valid: the type inference algorithm automatically computes the function *use*, and checks whether *use*(l) contains only valid access sequences for each resource creation point l.

1.3 The Rest of This Paper

Section 2 introduces a target language. Section 3 defines the problem of resource usage analysis. Sections 4 and 5 present a type-based method for resource usage analysis. Section 6 discusses extensions of the type-based method. Section 7 discusses related work and Section 8 concludes. A proof of the correctness of our type-based analysis and a detailed type inference algorithm are given in a longer version of this paper, available at `http://www.kb.cs.titech.ac.jp/~kobayasi/publications.html`.

2. TARGET LANGUAGE

This section introduces a call-by-value λ-calculus extended with primitives to create and access resources.

We assume that there is a countably infinite set $\mathcal{L}$ of labels, ranged over by meta-variable l. We write $\mathcal{L}^*$ for the set of finite sequences of labels, and write $\mathcal{L}^{*,\downarrow}$ for the set $\mathcal{L}^* \cup \{s\downarrow \mid s \in \mathcal{L}^*\}$. The special symbol '$\downarrow$' is used to denote the termination of a program. We call an element of $\mathcal{L}^{*,\downarrow}$ a *trace.* We write ϵ for the empty sequence, and $s_1 s_2$ for the concatenation of two traces s_1 and s_2.

A *trace set*, denoted by a meta-variable Φ, is a subset of $\mathcal{L}^{*,\downarrow}$ that is prefix-closed, i.e, $ss' \in \Phi$ implies $s \in \Phi$. $S^\sharp$ denotes the set of all prefixes of elements of S, i.e., $\{s \in \mathcal{L}^{*,\downarrow} \mid ss' \in S\}$.

Definition 1 (terms). *The syntax of terms is given by:*

$$\begin{array}{rl} M ::= & \mathbf{true} \mid \mathbf{false} \mid x \mid \lambda^\Phi x.M \mid \mathbf{fix}^\Phi(f, x, M) \\ & \mid \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \mid M_1 @^l M_2 \\ & \mid \mathbf{new}^\Phi() \mid \mathbf{acc}_i^l(M) \mid \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \end{array}$$

Here, we extended a standard λ-calculus with two constructs: $\mathbf{new}^\Phi()$ for creating a new resource and $\mathbf{acc}_i^l(M)$ for accessing resource M. For simplicity, we consider a single kind of resource except for functions (hence the single primitive for resource creation). Also, we assume that access primitives ($\mathbf{acc}_1, \ldots, \mathbf{acc}_n$) always return **true** or **false**. This is not so restrictive from the viewpoint of usage analysis: For example, the behavior of a primitive that accesses a resource and then returns the updated resource can be simulated by $\lambda r.(\mathbf{let}\ x = \mathbf{acc}_i^l(r)\ \mathbf{in}\ r)$. $\mathbf{fix}^\Phi(f, x, M)$ denotes a recursive function f that satisfies $f = \lambda x.M$. A let-expression $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ is computationally equivalent to $(\lambda^\Phi x.M_2)@^l M_1$, but we include it to make our type-based analysis in Section 4 more precise (see Section 6). A formal operational semantics of the language is defined in the next section.

A label (denoted by l) is attached to each occurrence of a primitive to access resources and functions. The same label may be attached to multiple occurrences of a primitive.

A trace set Φ is attached to each occurrence of λ-abstraction, a recursive function, and the resource creation primitive. It represents the programmer's intention on how the function or resource should be accessed during evaluation. A trace of the form $s\downarrow$ is a possible sequence of accesses performed to a resource by the time when evaluation terminates, while a trace of the form $s (\in \mathcal{L}^*)$ is a possible sequence of accesses performed by some time during evaluation. For example, $\mathbf{new}^{\{l_2\downarrow, l_1 l_2\downarrow\}^\sharp}()$ creates a resource that should be accessed at l_1 at most once and then accessed once at l_2 before evaluation of the whole term terminates. It is important to distinguish between traces ending with $\downarrow$ and those without $\downarrow$. For example, for a file, the trace set may contain $l_R; l_W$ but not $l_R; l_W\downarrow$, since the file should be closed before the program terminates.

We do not fix a particular way to specify trace sets Φ. They could be specified in various ways, for example, using regular expressions, shuffle expressions [10, 16] context-free grammars, modal logics [6], or usage expressions we introduce in Section 4.

Bound and free variables are defined in a standard manner. We write $\mathbf{FV}(M)$ for the set of free variables in M. When $x \notin \mathbf{FV}(M_2)$, we often write $M_1; M_2$ for $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$.

Example 1. Let **init**, **read**, **write**, and **free** be primitives to initialize, read, update, and deallocate a resource respectively. (In examples, we often use more readable names for primitives, rather than $\mathbf{acc}_i$.) The following program creates a new resource r, initializes it, and then calls function f. Inside function f, resource r is read and updated several times and then deallocated.

$$\begin{array}{l} \mathbf{let}\ f = \mathbf{fix}^{\Phi_f}(f, x, \mathbf{if}\ \mathbf{read}^{l_R}(x)\ \mathbf{then}\ \mathbf{free}^{l_F}(x) \\ \qquad\qquad\qquad\qquad \mathbf{else}\ (\mathbf{write}^{l_W}(x); f@^{l_1}x))\ \mathbf{in} \\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ (\mathbf{init}^{l_I}(r); f@^{l_2}r) \end{array}$$

Here, $\Phi_r = (l_I(l_R + l_W)^* l_F\downarrow)^\sharp$, and $\Phi_f = \mathcal{L}^{*,\downarrow}$ (where $l_I(l_R + l_W)^* l_F\downarrow$ is a regular expression). Φ_r specifies that r should be initialized first and deallocated at the end. This kind of access pattern (initialized, accessed, and then deallocated) often occurs to various types of resources (e.g., memory, files, Java objects [9]). The trace set $\mathcal{L}^{*,\downarrow}$ for Φ_f means that the programmer does not care about how function f is called.

3. RESOURCE USAGE ANALYSIS PROBLEM

The purpose of resource usage analysis is to infer how each resource is used in a given program, and check whether the inferred resource usage matches the programmer's intention (specified using trace sets). We give below a formal definition of the resource usage analysis problem, by using an operational semantics that takes the usage of resources into account.

3.1 Operational Semantics

We first introduce the notion of *heaps* to keep track of how each resource is used during evaluation: Formally, a heap is a mapping from variables to pairs of a heap value (either a resource or function) and a trace set.

DEFINITION 2 (HEAP VALUES, HEAP). *The set of* heap values*, ranged over by h, is given by the syntax:* $h ::= \mathcal{R} \mid \lambda x.M$*, where* $\mathcal{R}$ *denotes a resource created by* $\mathbf{new}^{\Phi}()$*. A* heap H *is a function from a finite set of variables to pairs of a heap value and a trace set.*

We write $\{x_1 \mapsto^{\Phi_1} h_1, \ldots, x_n \mapsto^{\Phi_n} h_n\}$ (n may be 0) for the heap H such that $dom(H) = \{x_1, \ldots, x_n\}$ and $H(x_i) = (h_i, \Phi_i)$. When $dom(H_1) \cap dom(H_2) = \emptyset$, we write $H_1 \uplus H_2$ for the heap H such that $dom(H) = dom(H_1) \cup dom(H_2)$ and $H(x) = H_i(x)$ if $x \in dom(H_i)$.

Following [18, 22, 26], program execution is represented by reduction of pairs of a heap and a term. When a resource is used at a program point l, the attached traces are "consumed" — the label l at the head of a trace is removed (if the trace begins with l; the traces not beginning with l are discarded). We define Φ^{-l}, which represents the trace set after the use at l, by $\{s \mid ls \in \Phi\}$. The formal reduction relation is defined below after a few auxiliary definitions.

DEFINITION 3 (SMALL VALUES, SUBSTITUTION). *A* small value *(or just* value*) v is either a variable,* **true***, or* **false***. We write* $[v/x]$ *for the standard capture-avoiding substitution of v for x.*

DEFINITION 4 (EVALUATION CONTEXTS). *The syntax of evaluation contexts is given by:*

$$\begin{array}{rcl}\mathcal{E} & ::= & [\,] \mid \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \mid \mathcal{E}@^l M \mid v@^l \mathcal{E} \\ & & \mid \mathbf{acc}^l_i(\mathcal{E}) \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ M\end{array}$$

We write $\mathcal{E}[M]$ *for the expression obtained by replacing* $[\,]$ *with M in* $\mathcal{E}$.

DEFINITION 5. *A reduction relation* $(H, M) \rightsquigarrow P$*, where P is either* **Error** *or a pair* (H', M')*, is defined as the least relation closed under th rules below.*

$$\frac{z \text{ fresh}}{(H, \mathcal{E}[\mathbf{new}^{\Phi}()]) \rightsquigarrow (H \uplus \{z \mapsto^{\Phi} \mathcal{R}\}, \mathcal{E}[z])} \quad \text{(R-AlcRes)}$$

$$\frac{z \text{ fresh}}{(H, \mathcal{E}[\lambda^{\Phi} x.M]) \rightsquigarrow (H \uplus \{z \mapsto^{\Phi} \lambda x.M\}, \mathcal{E}[z])} \quad \text{(R-AlcLam)}$$

$$\frac{z \text{ fresh}}{(H, \mathcal{E}[\mathbf{fix}^{\Phi}(f, x, M)]) \rightsquigarrow (H \uplus \{z \mapsto^{\Phi} \lambda x.[z/f]M\}, \mathcal{E}[z])} \quad \text{(R-AlcFix)}$$

$$\frac{b = \mathbf{true} \text{ or } \mathbf{false} \qquad \Phi^{-l} \neq \emptyset}{(H \uplus \{x \mapsto^{\Phi} \mathcal{R}\}, \mathcal{E}[\mathbf{acc}^l_i(x)]) \rightsquigarrow (H \uplus \{x \mapsto^{\Phi^{-l}} \mathcal{R}\}, \mathcal{E}[b])} \quad \text{(R-Acc)}$$

$$\frac{\Phi^{-l} = \emptyset}{(H \uplus \{x \mapsto^{\Phi} \mathcal{R}\}, \mathcal{E}[\mathbf{acc}^l_i(x)]) \rightsquigarrow \mathbf{Error}} \quad \text{(R-AccErr)}$$

$$\frac{\Phi^{-l} \neq \emptyset}{\begin{array}{l}(H \uplus \{x \mapsto^{\Phi} \lambda y.M\}, \mathcal{E}[x@^l v]) \\ \rightsquigarrow (H \uplus \{x \mapsto^{\Phi^{-l}} \lambda y.M\}, \mathcal{E}[[v/y]M])\end{array}} \quad \text{(R-App)}$$

$$\frac{\Phi^{-l} = \emptyset}{(H \uplus \{x \mapsto^{\Phi} \lambda y.M\}, x@^l v) \rightsquigarrow \mathbf{Error}} \quad \text{(R-AppErr)}$$

$$(H, \mathcal{E}[\mathbf{if\ true\ then}\ M_1\ \mathbf{else}\ M_2]) \rightsquigarrow (H, \mathcal{E}[M_1]) \quad \text{(R-IfT)}$$

$$(H, \mathcal{E}[\mathbf{if\ false\ then}\ M_1\ \mathbf{else}\ M_2]) \rightsquigarrow (H, \mathcal{E}[M_2]) \quad \text{(R-IfF)}$$

We write $\rightsquigarrow^*$ *for the reflexive transitive closure of* $\rightsquigarrow$.

Most of the rules are straightforward. In rules R-Acc and R-App, the attached trace set must include a trace beginning with l (represented by $\Phi^{-l} \neq \emptyset$). On the other hand, if no such traces are included, a usage error is signaled (R-AccErr and R-AppErr). Since we do not care about the result of resource access here, it is left unspecified which boolean value is returned in R-Acc. When an ordinary type error like application of a non-functional value occurs, the reduction will get stuck.

EXAMPLE 2. Let M be the following program, obtained by removing $\mathbf{init}^{l_I}(r)$ from the program in Example 1:

$$\begin{array}{l}\mathbf{let}\ f = \mathbf{fix}^{\Phi_f}(f, x, \mathbf{if\ read}^{l_R}(x)\ \mathbf{then\ free}^{l_F}(x) \\ \qquad\qquad\qquad\qquad \mathbf{else}\ (\mathbf{write}^{l_W}(x); f@^{l_1}x))\ \mathbf{in} \\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ f@^{l_2}r\end{array}$$

The evaluation of M fails because r is read before it is initialized.

$$\begin{array}{ll} & (\{\}, M) \\ \rightsquigarrow^* & (\{x_1 \mapsto^{\mathcal{L}^{*,\downarrow}} \lambda x.\mathbf{if\ read}^R(x)\ \mathbf{then}\ \cdots\ \mathbf{else}\ \cdots, \\ & \quad x_2 \mapsto^{\Phi_r} \mathcal{R}\}, x_1@^{l_2}x_2) \\ \rightsquigarrow & (\{x_1 \mapsto^{\mathcal{L}^{*,\downarrow}} \lambda x.(\cdots), x_2 \mapsto^{(l_I(l_R+l_W)^* l_F \downarrow)^{\sharp}} \mathcal{R}\}, \\ & \ \mathbf{if\ read}^{l_R}(x_2)\ \mathbf{then}\ \cdots\ \mathbf{else} \cdots) \\ \rightsquigarrow & \mathbf{Error}\end{array}$$

3.2 Resource Usage Analysis

Now, we define the problem of resource usage analysis. Intuitively, M is resource-safe if evaluation of M does not cause any usage errors and if all the resources are used up when the evaluation terminates.

DEFINITION 6. *M is* resource-safe *iff (1)* $(\{\}, M) \not\rightsquigarrow^*$ **Error** *and (2) if* $(\{\}, M) \rightsquigarrow^* (H, v)$*, then for any* $x \in dom(H)$*,* $H(x) = (h, \Phi)$ *and* $\downarrow \in \Phi$*. The* resource usage analysis problem *is, given a program M, to check whether M is resource-safe.*

Since the problem is undecidable, a resource usage analysis should be sound but need not be complete: If the answer is yes, the program should indeed be resource-safe, but even if the answer is no, the program may be resource-safe.

EXAMPLE 3. The program M in Example 1 is resource-safe.

EXAMPLE 4. Let M be the following program, obtained from the program in Example 1 by replacing $\mathbf{free}^{l_F}(x)$ in the definition of f with **true**:

$$\begin{array}{l}\mathbf{let}\ f = \mathbf{fix}^{\Phi_f}(f, x, \mathbf{if\ read}^{l_R}(x)\ \mathbf{then\ true} \\ \qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathbf{write}^{l_W}(x); f@^{l_1}x))\ \mathbf{in} \\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ (\mathbf{init}^{l_I}(r); f@^{l_2}r)\end{array}$$

It is evaluated as follows:

$$\begin{array}{ll} & (\{\}, M) \\ \rightsquigarrow^* & (\{x_1 \mapsto^{\mathcal{L}^{*,\downarrow}} \lambda x.(\cdots), x_2 \mapsto^{\{(l_R+l_W)^* l_F \downarrow\}^\sharp} \mathcal{R}\}, \\ & \ \mathbf{if\ true\ then\ true\ else}\ (\mathbf{write}^{l_W}(x_2); x_1@^{l_1}x_2)) \\ \rightsquigarrow & (\{x_1 \mapsto^{\mathcal{L}^{*,\downarrow}} \lambda x.(\cdots), x_2 \mapsto^{\{(l_R+l_W)^* l_F \downarrow\}^\sharp} \mathcal{R}\}, \mathbf{true})\end{array}$$

In the final state of the execution, the trace set associated to x_2 indicates that the resource still needs to be accessed at l_F before the execution terminates. Since the term cannot be reduced further, the program M is not resource-safe (the second condition of Definition 6 is violated).

REMARK 1. Alternatively, we can formalize usage analysis as a problem of giving not only an "yes"/"no" answer but also a trace set (consisting of possible access sequences) for each resource, as explained in Section 1. Our type-based analysis presented in Sections 4 and 5 can solve this problem too.

4. A TYPE SYSTEM FOR RESOURCE USAGE ANALYSIS

In this section, we present a type system that guarantees that every well-typed (closed) program is resource-safe. As mentioned in Section 1, a main idea is to augment the type of a resource with a usage, which expresses how the resource may be accessed. Note that programmers need not explicitly declare any usage in their programs: the type inference algorithm described in the next section can automatically recover usage information from (untyped) terms.

4.1 Usages, Types

DEFINITION 7 (USAGES). *The set $\mathcal{U}$ of* usages*, ranged over by U, is defined by:*

$$\begin{array}{rcl} U & ::= & \mathbf{0} \mid \alpha \mid l \mid U_1 \,\&\, U_2 \mid U_1\,;U_2 \mid U_1 \otimes U_2 \mid \mu\alpha.U \\ & & \mid \Box U \mid \blacksquare U \mid U_1 \odot U_2 \mid U_1 \rhd U_2 \end{array}$$

We assume that the unary usage constructors $\Box$ and $\blacksquare$ bind tighter than the binary constructors ($\&, ;, \otimes, \ldots$), so that $\Box l_1\,; l_2$ means $(\Box l_1)\,; l_2$.

$\mathbf{0}$ is the usage of a resource that cannot be accessed at all. α denotes a usage variable (which is bound by $\mu\alpha.$). Usages l, $U_1\,;U_2$, and $U_1 \,\&\, U_2$ have been explained in Section 1. $U_1 \otimes U_2$ is the usage of a resource that can be accessed according to U_1 and U_2 in an interleaved manner. So, $(l_1\,;l_2)\otimes l_3$ is equivalent to $(l_3;l_1;l_2)\,\&\,(l_1;l_3;l_2)\,\&\,(l_1;l_2;l_3)$. $\mu\alpha.U$ denotes a recursive usage such that $\alpha = U$. For example, $\mu\alpha.(\mathbf{0}\,\&\,(l;\alpha))$ means that the resource is accessed at l an arbitrary number of times. As mentioned in Section 1, $\Box U$ means that the resource may be accessed now or later according to U. So, a resource of usage $\Box l_1; l_2$ may be accessed either at l_1 and then at l_2, or at l_2 and then at l_1. $\blacksquare U$ means that the access represented by U must occur *now.* So, for example, $\blacksquare(\Box l_1; l_2; \Box l_3)$ is equivalent to $l_1 \otimes (l_2; l_3)$. Usage $U_1 \odot U_2$ means that the access represented by U_2 occurs for each single access represented by U_1. For example, $(l_1; l_2) \odot U$ is equivalent to $U;U$. Usage $U_1 \rhd U_2$ means that for each single access l represented by U_1, the access represented by $l;U_2$ happens. For example, $(l_1;l_2)\rhd U$ is equivalent to $l_1;U;l_2;U$. The precise meaning of each usage is defined in Subsection 4.2.

Probably, we do not need some of the usage constructors (like $\odot$ and $\rhd$) to express the final result of resource usage inference, but we need them to define the type system and the type inference algorithm.

DEFINITION 8 (TYPES). *The set of types, ranged over by τ, is defined by:*

$$\tau ::= \mathbf{bool} \mid (\tau_1 \to \tau_2, U) \mid (\mathbf{R}, U)$$

$(\tau_1 \to \tau_2, U)$ is the type of a function that is accessed (i.e., called) according to U. For example, $(\mathbf{bool} \to \mathbf{bool}, l_1; l_2)$ is the type of a boolean function that is called at l_1 first and then called at l_2. (This kind of information is, for example, useful to determine when function closures should be deallocated.) $(\mathbf{R}, U)$ is the type of a resource that is accessed according to U.

The outermost usage of τ, written $\mathit{Use}(\tau)$, is defined by: $\mathit{Use}(\mathbf{bool}) = \mathbf{0}$, $\mathit{Use}(\tau_1 \to \tau_2, U) = U$, and $\mathit{Use}(\mathbf{R}, U) = U$.

4.2 Semantics of Usages

We define the meaning of usages using a labeled transition semantics. A usage denotes a set of traces, obtained from possible transition sequences. We also define a subusage relation, which induces a subtyping relation, using the labeled transition system and the usual notion of simulation.

We first define auxiliary relations. We write $U_1 \preceq U_2$ when U_2 is obtained from U_1 by unfolding some recursive usages ($\mu\alpha.U$) and removing some branches from choices ($U \,\&\, U'$).

DEFINITION 9. *A relation $\preceq$ is the least reflexive and transitive relation on usages that satisfies the rules in Figure 1.*

For example, $l_1; (l_2 \,\&\, l_3) \preceq l_1; l_2$.

DEFINITION 10. *Unary relations $\cdot^{\downarrow}$ and $\cdot^{\Downarrow}$ are the least relations on usages that satisfies the following conditions:*

$$\begin{array}{l}\mathbf{0}^{\downarrow} \\ U^{\downarrow} \Rightarrow ((\Box U)^{\downarrow} \wedge (\blacksquare U)^{\downarrow} \wedge (U \odot U')^{\downarrow} \\ \qquad\qquad\qquad \wedge (U' \odot U)^{\downarrow} \wedge (U \rhd U')^{\downarrow}) \\ ({U_1}^{\downarrow} \wedge {U_2}^{\downarrow}) \Rightarrow ((U_1 \otimes U_2)^{\downarrow} \wedge (U_1\,; U_2)^{\downarrow} \wedge (U_1 \,\&\, U_2)^{\downarrow}) \\ ([\mu\alpha.U/\alpha]U)^{\downarrow} \Rightarrow (\mu\alpha.U)^{\downarrow} \\ \mathbf{0}^{\Downarrow} \\ (\Box U)^{\Downarrow} \\ U^{\downarrow} \Rightarrow (\blacksquare U)^{\Downarrow} \\ U^{\Downarrow} \Rightarrow ((U \odot U')^{\Downarrow} \wedge (U' \odot U)^{\Downarrow} \wedge (U \rhd U')^{\Downarrow}) \\ ({U_1}^{\Downarrow} \wedge {U_2}^{\Downarrow}) \Rightarrow ((U_1 \otimes U_2)^{\Downarrow} \wedge (U_1\,; U_2)^{\Downarrow} \wedge (U_1 \,\&\, U_2)^{\Downarrow}) \\ ([\mu\alpha.U/\alpha]U)^{\Downarrow} \Rightarrow (\mu\alpha.U)^{\Downarrow}\end{array}$$

Intuitively, $U^{\downarrow}$ means that the resource is no longer accessed before evaluation of the whole term terminates. $U^{\Downarrow}$ means that the resource is not accessed for now. (In other words all labels are "boxed.") We write $U^{\preceq\downarrow}$ if $U \preceq U'$ and $U'^{\downarrow}$ for some U'. Similarly, we write $U^{\preceq\Downarrow}$ if $U \preceq U'$ and $U'^{\Downarrow}$ for some U'.

$$U_1 \,\&\, U_2 \preceq U_1$$
$$U_1 \,\&\, U_2 \preceq U_2$$
$$\mu\alpha.U \preceq [\mu\alpha.U/\alpha]U$$
$$\frac{U_1 \preceq U_1' \qquad U_2 \preceq U_2'}{U_1 \,;\, U_2 \preceq U_1' \,;\, U_2'}$$
$$\frac{U_1 \preceq U_1' \qquad U_2 \preceq U_2'}{U_1 \otimes U_2 \preceq U_1' \otimes U_2'}$$
$$\frac{U \preceq U'}{\square U \preceq \square U'}$$
$$\frac{U \preceq U'}{\blacksquare U \preceq \blacksquare U'}$$
$$\frac{U_1 \preceq U_1' \qquad U_2 \preceq U_2'}{U_1 \odot U_2 \preceq U_1' \odot U_2'}$$
$$\frac{U_1 \preceq U_1' \qquad U_2 \preceq U_2'}{U_1 \triangleright U_2 \preceq U_1' \triangleright U_2'}$$

Figure 1: Relation $U \preceq U'$

We write $\square\mathcal{L}$ for the set $\{\square l \mid l \in \mathcal{L}\}$. We call an element of $\mathcal{L} \cup \square\mathcal{L}$ an *extended label*, and use a meta-variable L for it. When L is an extended label, $\square L$ and $\blacksquare L$ is defined by:

$$\square L = \begin{cases} \square L & \text{if } L \in \mathcal{L} \\ L & \text{if } L \in \square\mathcal{L} \end{cases}$$
$$\blacksquare L = \begin{cases} L & \text{if } L \in \mathcal{L} \\ l & \text{if } L = \square l \end{cases}$$

Let S be a set of extended labels. We write $\square S$ and $\blacksquare S$ for the sets $\{\square L \mid L \in S\}$ and $\{\blacksquare L \mid L \in S\}$ respectively.

Now we define a transition relation $U \xrightarrow{L} U'$. Intuitively, it implies that a resource of usage U can be first accessed at L and then accessed according to U'.

Definition 11. *A transition relation $U \xrightarrow{L} U'$ on usages is the least relation closed under the rules in Figure 2.*

Example 5. $\square l_1; l_2$ has two transition sequences: $\square l_1; l_2 \xrightarrow{\square l_1} \mathbf{0}; l_2 \xrightarrow{l_2} \mathbf{0}; \mathbf{0}$ and $\square l_1; l_2 \xrightarrow{l_2} \square l_1; \mathbf{0} \xrightarrow{\square l_1} \mathbf{0}; \mathbf{0}$ but $l_1; l_2$ has only the transition sequence: $l_1; l_2 \xrightarrow{l_1} \mathbf{0}; l_2 \xrightarrow{l_2} \mathbf{0}; \mathbf{0}$. (Note the righthand premise of rule (UR-SeqR).)

The set of traces denoted by a usage U, written $[\![U]\!]$, is defined as follows.

Definition 12. *Let U be a usage. $[\![U]\!]$ denotes the set:*

$$\{(\blacksquare L_1)\cdots(\blacksquare L_n) \mid \exists U_1,\ldots,U_n.(U \xrightarrow{L_1} U_1 \cdots U_{n-1} \xrightarrow{L_n} U_n)\}$$
$$\cup\{(\blacksquare L_1)\cdots(\blacksquare L_n)\downarrow \mid$$
$$\exists U_1,\ldots,U_n.((U \xrightarrow{L_1} U_1 \cdots U_{n-1} \xrightarrow{L_n} U_n) \wedge U_n{}^{\preceq\downarrow})\}$$

Here, n can be 0 (so $\epsilon \in [\![U]\!]$ for any U).

It is trivial by definition that $[\![U]\!]$ is a trace set (i.e., prefix-closed).

Example 6.

$$[\![\mathbf{0}]\!] = \{\epsilon, \downarrow\}, \quad [\![\mu\alpha.\alpha]\!] = \{\epsilon\}$$
$$[\![\square(l_1; l_2); l_3]\!] = \{l_1 l_2 l_3 \downarrow, l_1 l_3 l_2 \downarrow, l_3 l_1 l_2 \downarrow\}^\sharp$$
$$[\![\mu\alpha.(\mathbf{0} \,\&\, (l;\alpha))]\!] = \{\downarrow, l\downarrow, ll\downarrow, lll\downarrow, \ldots\}^\sharp$$

4.3 Subtyping

We define *subusage* and *subtype* relations $U_1 \leq U_2$ and $\tau_1 \leq \tau_2$ below. Intuitively, $U_1 \leq U_2$ means that U_1 represents a more general usage than U_2, so that a resource of usage U_1 may be used as that of usage U_2. Similarly, $\tau_1 \leq \tau_2$ means that a value of type τ_1 may be used as a value of type τ_2.

We define the subusage relation to be closed under usage contexts. Formally, a *usage context*, written C, is an expression obtained from a usage by replacing one occurrence of a free usage variable with $[\,]$. Suppose that the set of free usage variables in U are disjoint from the set of bound usage variables in C. We write $C[U]$ for the usage obtained by replacing $[\,]$ with U. For example, if $C = \mu\alpha.([\,]\,;\alpha)$, then $C[U] = \mu\alpha.(U\,;\alpha)$.

Definition 13. Subusage *relation $\leq$ is the largest binary relation such that for all $U_1, U_2 \in \mathcal{U}$, if $U_1 \leq U_2$, then the following conditions are satisfied:*

1. *$C[U_1] \leq C[U_2]$ for any usage context C;*
2. *If $U_2 \xrightarrow{L} U_2'$, then $U_1 \xrightarrow{L'} U_1'$ and $U_1' \leq U_2'$ for some U_1' and L' with $L' = L$ or $L' = \square L$; and*
3. *If $U_2{}^{\preceq\downarrow}$, then $U_1{}^{\preceq\downarrow}$.*

We write $U_1 \cong U_2$ if and only if $U_1 \leq U_2$ and $U_2 \leq U_1$.

Definition 14. Subtype *relation $\leq$ is the least binary relation on types that satisfies the following rules:*

$$\mathbf{bool} \leq \mathbf{bool} \qquad \text{(Sub-Bool)}$$

$$\frac{U \leq U'}{(\tau_1 \to \tau_2, U) \leq (\tau_1 \to \tau_2, U')} \qquad \text{(Sub-Fun)}$$

$$\frac{U \leq U'}{(\mathbf{R}, U) \leq (\mathbf{R}, U')} \qquad \text{(Sub-Res)}$$

Remark 2. Actually, we could relax the above subusage and subtype relations. For the subtype relation, for example, we can replace rule (Sub-Fun) with the following rule.

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad U \leq U'}{(\tau_1 \to \tau_2, U) \leq (\tau_1' \to \tau_2', U')}$$

4.4 Type environments

A type environment is a mapping from a finite set of variables to types. We use meta-variables Γ and Δ for type environments. We write $\emptyset$ for the type environment whose domain is empty. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Δ such that $dom(\Delta) = dom(\Gamma) \cup \{x\}$, $\Delta(x) = \tau$, and $\Delta(y) = \Gamma(y)$ for $y \in dom(\Gamma)$.

We define several operations and relations on types and type environments.

$$l \xrightarrow{l} \mathbf{0} \qquad \text{(UR-Zero)}$$

$$\frac{U_1 \xrightarrow{L} U_1'}{U_1 \otimes U_2 \xrightarrow{L} U_1' \otimes U_2} \qquad \text{(UR-ParL)}$$

$$\frac{U_2 \xrightarrow{L} U_2'}{U_1 \otimes U_2 \xrightarrow{L} U_1 \otimes U_2'} \qquad \text{(UR-ParR)}$$

$$\frac{U_1 \xrightarrow{L} U_1'}{U_1 \,;\, U_2 \xrightarrow{L} U_1' \,;\, U_2} \qquad \text{(UR-SeqL)}$$

$$\frac{U_2 \xrightarrow{L} U_2' \qquad U_1{}^{\Downarrow}}{U_1 \,;\, U_2 \xrightarrow{L} U_1 \,;\, U_2'} \qquad \text{(UR-SeqR)}$$

$$\frac{U \xrightarrow{L} U'}{\square U \xrightarrow{\square L} \square U'} \qquad \text{(UR-Box)}$$

$$\frac{U \xrightarrow{L} U'}{\blacksquare U \xrightarrow{\blacksquare L} \blacksquare U'} \qquad \text{(UR-Unbox)}$$

$$\frac{U_1 \xrightarrow{l} U_1' \qquad U_2 \xrightarrow{L} U_2'}{U_1 \odot U_2 \xrightarrow{L} U_2' \,;\, (U_1' \odot U_2)} \qquad \text{(UR-Mult1)}$$

$$\frac{U_1 \xrightarrow{\square l} U_1' \qquad \square U_2 \xrightarrow{L} U_2'}{U_1 \odot U_2 \xrightarrow{L} U_2' \,;\, (U_1' \odot U_2)} \qquad \text{(UR-Mult2)}$$

$$\frac{U_1 \xrightarrow{l} U_1'}{U_1 \rhd U_2 \xrightarrow{l} U_2 \,;\, (U_1' \rhd U_2)} \qquad \text{(UR-SMult1)}$$

$$\frac{U_1 \xrightarrow{\square l} U_1'}{U_1 \rhd U_2 \xrightarrow{\square l} \square U_2 \,;\, (U_1' \rhd U_2)} \qquad \text{(UR-SMult2)}$$

$$\frac{U \preceq U'' \qquad U'' \xrightarrow{L} U'}{U \xrightarrow{L} U'} \qquad \text{(UR-PCong)}$$

Figure 2: Usage Reduction Rules

Definition 15. *Let C be a usage context. Suppose that the set of free usage variables appearing in τ or Γ is disjoint from the set of bound usage variables in C. We define $C[\tau]$ and $C[\Gamma]$ by:*

$$\begin{array}{rcl} C[\mathbf{bool}] & = & \mathbf{bool} \\ C[(\tau_1 \to \tau_2, U)] & = & (\tau_1 \to \tau_2, C[U]) \\ C[(\mathbf{R}, U)] & = & (\mathbf{R}, C[U]) \\ dom(C[\Gamma]) & = & dom(\Gamma) \\ C[\Gamma](x) & = & C[\Gamma(x)] \end{array}$$

Let $\mathbf{op}$ *be a binary usage constructor* $\_\,;\,\_$ *or* $\_\,\&\,\_$. *It is extended to operations on types and type environments by:*

$$\begin{array}{rcl} \mathbf{bool}\,\mathbf{op}\,\mathbf{bool} & = & \mathbf{bool} \\ (\tau_1 \to \tau_2, U_1)\,\mathbf{op}\,(\tau_1 \to \tau_2, U_2) & = & (\tau_1 \to \tau_2, U_1\,\mathbf{op}\,U_2) \\ (\mathbf{R}, U_1)\,\mathbf{op}\,(\mathbf{R}, U_2) & = & (\mathbf{R}, U_1\,\mathbf{op}\,U_2) \end{array}$$

$$dom(\Gamma_1\,\mathbf{op}\,\Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$$

$$(\Gamma_1\,\mathbf{op}\,\Gamma_2)(x) = \begin{cases} \Gamma_1(x)\,\mathbf{op}\,\Gamma_2(x) & \\ \qquad\qquad \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) & \\ \Gamma_1(x) \quad \text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2) & \\ \Gamma_2(x) \quad \text{if } x \in dom(\Gamma_2) \backslash dom(\Gamma_1) & \end{cases}$$

The type environment $\blacksquare_x \Gamma$ is defined by

$$\blacksquare_x \Gamma = \begin{cases} \Gamma & \text{if } x \notin dom(\Gamma) \\ \Gamma'', x : \blacksquare \tau_x & \text{if } \Gamma = \Gamma'', x : \tau_x \text{ and } \tau_x \neq \mathbf{bool} \end{cases}$$

Note that if $\Gamma(x) = \mathbf{bool}$, then $\blacksquare_x \Gamma$ is undefined.

Example 7. Let Γ be $x : (\mathbf{R}, U)$ and Δ be $x : (\mathbf{R}, U'), y : \mathbf{bool}$. Then, $\square\Gamma = x : \square(\mathbf{R}, U) = x : (\mathbf{R}, \square U)$ and $\Gamma; \Delta = x : (\mathbf{R}, U); (\mathbf{R}, U'), y : \mathbf{bool} = x : (\mathbf{R}, U; U'), y : \mathbf{bool}$.

We write $\Gamma_1 \leq \Gamma_2$ when $dom(\Gamma_1) \supseteq dom(\Gamma_2)$, $\Gamma_1(x) \leq \Gamma_2(x)$ for all $x \in dom(\Gamma_2)$, and $Use(\Gamma_1(x)) \leq \mathbf{0}$ for all $x \in dom(\Gamma_1) \backslash dom(\Gamma_2)$.

4.5 Typing

A type judgment is of the form $\Gamma \vdash M : \tau$. It carries not only ordinary type information but information about how resources are used. For example, $x : (\mathbf{R}, l_1; \square l_2) \vdash M : (\mathbf{R}, l_2)$ implies that if x is a resource that can be accessed at l_1 and then at l_2, then M is evaluated to a resource that can be accessed at l_2. It further implies that the access at l_1 occurs during evaluation of M, while the access at l_2 occurs either during evaluation of M or when the value of M is used (because l_1 is not boxed but l_2 is boxed). For example, if $M = \mathbf{let}\ y = \mathbf{acc}_i^{l_1}(x)\ \mathbf{in}\ x$, this type judgment holds. (Actually, annotation on escape information is necessary: See Example 8 below.)

As mentioned in Section 1, an escape analysis [4, 12] is useful to refine the accuracy of our type-based usage analysis. To make our type system simple and clarify its essence, we assume that a kind of escape analysis has been already performed and that a program is annotated with the result of the escape analysis. We extend the syntax of terms by introducing a term of the form $M^{\{x\}}$, which means that x does not escape from M, in the sense that a heap value referred to by x is not contained in (unreachable from) the value of M. A simplest escape analysis would be to compare the type of M and that of x, as in variants of linear type system [27, 29]: For example if the type of M is **bool** or if the type of M is a resource type but the type of x is a function type, x cannot escape from M (in the above sense).

Typing rules are shown in Figure 3. In rule (T-Var), the $\square$-operator is applied to the type of x in the type environment, because x is used only later, not when x is evaluated.

In rules (T-New), (T-Abs), and (T-Fix), the premise $[\![\, U \,]\!] \subseteq \Phi$ checks that resources or functions created here are accessed according to Φ (which represents the programmer's intention). In rule (T-Abs), the lefthand premise $\Gamma, x : \tau_1 \vdash M : \tau_2$ means that the resources referred to by free variables in $\lambda^\Phi x.M$ are accessed according to Γ *each time* the function is called. Since the function itself is called according to U, we multiply Γ by U to obtain a type environment expressing the total access.[2] Finally, since the resources are accessed only later when the function is called, $\square$ is applied.

Rule (T-Fix) is similar to (T-Abs), except that the usage describing how the function is called is more complex.

[2]Similar calculation is performed in linear type systems [18, 14, 13].

$$\frac{c = \mathbf{true} \text{ or } \mathbf{false}}{\emptyset \vdash c : \mathbf{bool}} \quad (\text{T-Const})$$

$$x : \Box\tau \vdash x : \tau \quad (\text{T-Var})$$

$$\frac{[\![U]\!] \subseteq \Phi}{\emptyset \vdash \mathbf{new}^{\Phi}() : (\mathbf{R}, U)} \quad (\text{T-New})$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2 \qquad [\![U]\!] \subseteq \Phi}{\Box(U \odot \Gamma) \vdash \lambda^{\Phi} x.M : (\tau_1 \to \tau_2, U)} \quad (\text{T-Abs})$$

$$\frac{\Gamma, f : (\tau_1 \to \tau_2, U_1), x : \tau_1 \vdash M : \tau_2 \qquad [\![U_2 \rhd (\mu\alpha.U_1 \rhd \alpha)]\!] \subseteq \Phi \qquad \alpha \text{ fresh}}{\Box(U_2 \odot (\mu\alpha.(\Gamma \otimes (U_1 \odot \alpha)))) \vdash \mathbf{fix}^{\Phi}(f, x, M) : (\tau_1 \to \tau_2, U_2)} \quad (\text{T-Fix})$$

$$\frac{\Gamma_1 \vdash M_1 : (\tau_1 \to \tau_2, l) \qquad \Gamma_2 \vdash M_2 : \tau_1}{\Gamma_1 ; \Gamma_2 \vdash M_1 @^{l} M_2 : \tau_2} \quad (\text{T-App})$$

$$\frac{\Gamma \vdash M : (\mathbf{R}, l)}{\Gamma \vdash \mathbf{acc}_i^{l}(M) : \mathbf{bool}} \quad (\text{T-Acc})$$

$$\frac{\Gamma_1 \vdash M_1 : \mathbf{bool} \qquad \Gamma_2 \vdash M_2 : \tau \qquad \Gamma_3 \vdash M_3 : \tau}{\Gamma_1 ; (\Gamma_2 \,\&\, \Gamma_3) \vdash \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : \tau} \quad (\text{T-If})$$

$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 ; \Gamma_2 \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2} \quad (\text{T-Let})$$

$$\frac{\Gamma \vdash M : \tau}{\blacksquare_x \Gamma \vdash M^{\{x\}} : \tau} \quad (\text{T-Now})$$

$$\frac{\Gamma' \vdash M : \tau' \qquad \Gamma \leq \Gamma' \qquad \tau' \leq \tau}{\Gamma \vdash M : \tau} \quad (\text{T-Sub})$$

Figure 3: Typing Rules

The type of $\mathbf{fix}^{\Phi}(f, x, M)$ implies that the function is called according to U_2 from the outside. Upon each call, M is evaluated and the function is internally called according to U_1, and for each internal call, the function is again called according to U_1. Therefore, the usage of the function in total is represented by $U_2 \rhd (\mu\alpha.U_1 \rhd \alpha)$. As for usage of the resources referred to by the free variables of the function, upon each call, M is evaluated and the resources are accessed according to Γ. Moreover, during the evaluation, internal calls occur and for each internal call, the resources are again accessed according to Γ. The usage in total is therefore represented by $U_2 \odot (\mu\alpha.(\Gamma \otimes (U_1 \odot \alpha)))$ and $\Box$ is applied to this, since the resources are accessed only later, not when the term $\mathbf{fix}^{\Phi}(f, x, M)$ is evaluated.

In rule (T-App), the premises imply that resources are accessed according to Γ_1 and Γ_2 in M_1 and M_2 respectively. Because M_1 is evaluated first, the usage of resources in total is represented by $\Gamma_1; \Gamma_2$. Because the function M_1 is called at l, the usage of M_1 must be l. Similarly, in rule (T-Acc), the usage of M must be l because it is accessed at l.[3]

In rule (T-If), after M_1 is evaluated, either M_2 or M_3 is evaluated. Thus, the usage of resources in total is represented by $\Gamma_1; (\Gamma_2 \& \Gamma_3)$. In rule (T-Now), $M^{\{x\}}$ asserts that x does not escape from M. So, the access represented by τ should happen now, when M is evaluated. The operator $\blacksquare$ is applied to reflect this fact. We exclude the case where $\tau_1 = \mathbf{bool}$, since in that case escape information is useless.

Example 8. A derivation for the type judgment

$$x : (\mathbf{R}, l_1; \Box l_2) \vdash \mathbf{let}\ y = \mathbf{acc}_i^{l_1}(x)^{\{x\}}\ \mathbf{in}\ x : (\mathbf{R}, l_2)$$

is shown in Figure 4.

4.6 Type Soundness

The above type system is sound in the sense that every closed well-typed expression of type τ where $\mathit{Use}(\tau) \leq \mathbf{0}$ is resource-safe, provided that the escape analysis is sound. The condition $\mathit{Use}(\tau) \leq \mathbf{0}$ means that resources contained in the result of the evaluation may no longer be accessed.

[3]Actually, because the value of $\mathbf{acc}_i^{l}(M)$ cannot contain references to resources, it is safe to apply $\blacksquare$ to Γ in the conclusion.

In order to make explicit the assumption on the escape analysis, we extend the operational semantics of the target language to deal with terms of the form $M^{\{x\}}$. First, we extend the syntax of evaluation contexts by:

$$\mathcal{E} \quad ::= \quad \cdots \mid \mathcal{E}^{\{x\}}$$

We add the following reduction rule:

$$\frac{x \notin \mathit{Reach}(v, H)}{(H, \mathcal{E}[v^{\{x\}}]) \leadsto (H, \mathcal{E}[v])} \quad (\text{R-ECheck})$$

Here, $\mathit{Reach}(v, H)$ is the least set that satisfies the following conditions:

$$x \in \mathit{Reach}(x, H)$$
$$(y \in \mathit{Reach}(v, H) \wedge y \in \mathit{dom}(H)) \Rightarrow \mathbf{FV}(H(y)) \subseteq \mathit{Reach}(v, H)$$

Intuitively, $\mathit{Reach}(v, H)$ is the set of variables (i.e., heap locations) reachable from v.

Rule (R-ECheck) makes sure that if the escape analysis were wrong, evaluation would get stuck. Now soundness of our type system is stated as follows.

Theorem 1 (type soundness). *If $\emptyset \vdash M : \tau$ and $\mathit{Use}(\tau) \leq \mathbf{0}$, then M is resource-safe.*

Proof Sketch. We use a technique similar to the one used in Kobayashi's quasi-linear type system [18]. We introduce another operational semantics to the target language—the semantics takes into account not only how but also *where* in the expression each heap value is used during evaluation. This alternative semantics is shown to be equivalent to the standard semantics in a certain sense and the type system is shown to be sound with respect to the alternative semantics. Interested readers are referred to a longer version of the paper available at `http://www.kb.cs.titech.ac.jp/~kobayasi/publications.html` for details. $\Box$

5. A TYPE INFERENCE ALGORITHM

Let M be a closed term. By the type soundness theorem (Theorem 1), in order to verify that all resources are used

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{x : (\mathbf{R}, \Box l_1) \vdash x : (\mathbf{R}, l_1)}\ \text{(T-Var)}
}{x : (\mathbf{R}, \Box l_1) \vdash \mathbf{acc}_i^{l_1}(x) : \mathbf{bool}}\ \text{(T-Acc)}
}{x : (\mathbf{R}, \blacksquare\Box l_1) \vdash \mathbf{acc}_i^{l_1}(x)^{\{x\}} : \mathbf{bool}}\ \text{(T-Now)}
}{x : (\mathbf{R}, l_1) \vdash \mathbf{acc}_i^{l_1}(x)^{\{x\}} : \mathbf{bool}}\ \text{(T-Sub)}
\qquad
\dfrac{
\dfrac{}{x : (\mathbf{R}, \Box l_2) \vdash x : (\mathbf{R}, l_2)}\ \text{(T-Var)}
}{x : (\mathbf{R}, \Box l_2), y : \mathbf{bool} \vdash x : (\mathbf{R}, l_2)}\ \text{(T-Sub)}
}{x : (\mathbf{R}, l_1; \Box l_2) \vdash \mathbf{let}\ y = \mathbf{acc}_i^{l_1}(x)^{\{x\}}\ \mathbf{in}\ x : (\mathbf{R}, l_2)}\ \text{(T-Let)}
$$

Figure 4: An Example of Type Derivation

correctly in M, it suffices to verify that $\emptyset \vdash M : \tau$ holds for some type τ with $\mathit{Use}(\tau) \leq \mathbf{0}$. In this section, we describe an algorithm to check it.

For simplicity, we assume the following conditions.

- Escape analysis has been already performed, and an input term is annotated with the result of the escape analysis.
- The *standard type* (the part of a type obtained by removing usages) of each term has been already obtained by the usual type inference. We write ρ_N for the standard type of each occurrence of a term N.
- Given a usage U and a set Φ of traces, there is an algorithm that verifies $[\![U]\!] \subseteq \Phi$. This algorithm should be sound but may not be complete; in fact, depending on U and how Φ is specified, the problem can become undecidable.

Because we do not expect a complete algorithm in the third assumption, our algorithm described below is sound but incomplete.

Our algorithm proceeds as follows, in a manner similar to an ordinary type inference algorithm [17, 19] for the simply-typed λ-calculus.

Step 1 Construct a template of a derivation tree for $\emptyset \vdash M : \tau$, using usage variables to denote unknown usages.

Step 2 Extract constraints on the usage variables from the template.

Step 3 Solve constraints on usage variables.

5.1 Step 1: Constructing a template of a type derivation tree

First, we obtain syntax-directed typing rules equivalent to the typing rules given in Section 4, so that there is exactly one rule that matches each term. It is obtained by combining each rule with (T-Sub) and removing (T-Sub). For example, (T-App) is replaced by the following rule:

$$
\dfrac{\Gamma_1 \vdash M_1 : (\tau_1 \to \tau_2, l) \qquad \Gamma_2 \vdash M_2 : \tau_1 \\ \Gamma \leq \Gamma_1; \Gamma_2 \qquad \tau_2 \leq \tau_2'}{\Gamma \vdash M_1 @^l M_2 : \tau_2'}\ \text{(T-App')}
$$

For each subterm N of an input term M, we prepare:

(i) a type τ_N such that all the usages in τ_N are fresh usage variables, and except for the usages, τ_N is identical to ρ_N.

(ii) a type environment Γ_N such that $\mathit{dom}(\Gamma_N) = \mathbf{FV}(N)$ and for each $x \in \mathit{dom}(\Gamma_N)$, $\Gamma_N(x)$ is identical to τ_x except for their outermost usages. The outermost usage of $\Gamma_N(x)$ (i.e., $\mathit{Use}(\Gamma_N(x))$) is a fresh usage variable.

We can construct a template of a type derivation tree, by labeling each node with a judgment $\Gamma_N \vdash N : \tau_N$. For example, consider a term $f @^l x$ where the standard type of f is $\mathbf{R} \to \mathbf{bool}$. The template is:

$$
\dfrac{f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_3) \vdash f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_2) \\ x : (\mathbf{R}, \alpha_5) \vdash x : (\mathbf{R}, \alpha_4)}{f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_6), x : (\mathbf{R}, \alpha_7) \vdash f @^l x : \mathbf{bool}}
$$

5.2 Step 2: Extracting constraints

In order to make the template a valid type derivation tree, it suffices to instantiate usage variables so that the side conditions of a syntax-directed typing rule are satisfied at each derivation step. The side conditions are expressed as constraints on types and usages. For example, for the node where the rule (T-App') is applied, the side conditions can be expressed by:

$$
\begin{aligned}
&\{\mathit{Use}(\Gamma_{M_1 @^l M_2}(x)) \leq \mathit{Use}(\Gamma_{M_1}(x); \Gamma_{M_2}(x)) \\
&\qquad\qquad\qquad \mid x \in \mathit{dom}(\Gamma_{M_1 @^l M_2})\} \\
&\cup \{\mathit{domty}(\tau_{M_1}) = \tau_{M_2}, \tau_{M_1 M_2} \leq \mathit{codty}(\tau_{M_1})\}
\end{aligned}
$$

Here, *domty* and *codty* is defined by: $\mathit{domty}(\tau_1 \to \tau_2, U) = \tau_1$ and $\mathit{codty}(\tau_1 \to \tau_2, U) = \tau_2$.

Let CS be the set of constraints obtained by gathering the side conditions for every node of the template, plus the constraint $\mathit{Use}(\tau_M) \leq \mathbf{0}$. Then, a substitution θ for usage variables satisfies CS if and only if the derivation tree obtained by applying θ to the template is a valid type derivation tree. Therefore, the problem of deciding whether $\emptyset \vdash M : \mathbf{bool}$ holds is reduced to the problem of deciding whether CS is satisfiable.

We can reduce the constraints on types and obtain the following set of constraints on usages:

$$
\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\} \cup \{[\![U_1']\!] \subseteq \Phi_1, \ldots, [\![U_m']\!] \subseteq \Phi_m\}
$$

We can assume without loss of generality that $\alpha_1, \ldots, \alpha_n$ are distinct usage variables, because $\alpha \leq U_1 \wedge \alpha \leq U_2$ holds if and only if $\alpha \leq U_1 \,\&\, U_2$ holds.

5.3 Step 3: Solving constraints

Given the set of constraints $\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\} \cup \{[\![U_1']\!] \subseteq \Phi_1, \ldots, [\![U_m']\!] \subseteq \Phi_m\}$, we can eliminate the first set of constraints by repeatedly applying the transformation: $\mathit{CS} \cup \{\alpha \leq U\} \Longrightarrow [\mu\alpha.U/\alpha]\,\mathit{CS}$. Then, we check whether the remaining set of constraints is satisfied (using the algorithm stated in the third assumption).

5.4 Properties of the Algorithm

The above algorithm is *relatively* sound and complete with respect to an algorithm to judge $[\![U]\!] \subseteq \Phi$: The former is sound (complete, resp.) if the latter is sound (complete, resp.). Note that in the step 3 above, we are using the fact that $\mu\alpha.U$ is the least solution of $\alpha \leq U$ in the sense that $U' \leq [U'/\alpha]U$ implies $[\![\mu\alpha.U]\!] \subseteq [\![U']\!]$.

Suppose that the size of the standard types ρ_N of subterms is bound by a constant. Then, computational cost of the above algorithm, excluding the cost for checking the validity of constraints of the form $[\![U]\!] \subseteq \Phi$, is quadratic in the size n of an input term. Note that the size of each constraint set $\mathcal{C}(N)$ in Step 2 is $O(n)$. So, the size of the set CS of all constraints is $O(n^2)$. It is reduced to constraints on usages in $O(n^2)$ steps and the size of the resulting constraints in Step 2 is also $O(n^2)$. Therefore, the total cost of the algorithm is $O(n^2)$. Actually, we expect that we can remove the assumption that the size of standard types is bound, by performing inference of standard types and that of usages simultaneously, in a manner similar to [19].

We assumed above that a whole program is given as an input. It is not difficult to adapt our algorithm to perform a modular analysis: The first and second steps of extracting and reducing constraints can be applied to open terms. The third step can also be partially performed, because constraints on a usage variable α can be solved when we know that no constraint on α is imposed by the outside of the program being analyzed.

5.5 Examples

We give examples of our analysis. We omit annotations on escape information below, but assume that terms of type **bool** are appropriately annotated with escape information (as in $(\mathbf{acc}_1^{l_I}(r))^{\{r\}}$, $(f@^{l_2}r)^{\{r\}}$).

EXAMPLE 9. *Let us consider the program in Example 1. Let the types of f and r in* $\mathbf{write}^{l_I}(x); f@^{l_2}r$ *be* $((\mathbf{R}, \alpha_x) \rightarrow \mathbf{bool}, \alpha_f)$ *and* $(\mathbf{R}, \alpha_r)$*. Then, we get the following constraints on usage variables* α_x, α_r, *and* α_f*:*

$$\{\alpha_x \leq l_R; (l_F \,\&\, (l_W; \alpha_x)),\ \alpha_r \leq l_I; \alpha_x,\ \alpha_f \leq l_2\}$$

By solving this, we know that the usages of r and f in the whole program are $l_I; \mu\alpha_x.(l_R; (l_F \,\&\, (l_W; \alpha_x)))$ *and* $l_2 \triangleright \mu\alpha.((\mathbf{0} \,\&\, l_1) \triangleright \alpha)$*. The usage of r implies that r is first initialized, read and written several times, and then deallocated.*

EXAMPLE 10. *Let us consider the following program:*

$$\begin{array}{l}\mathbf{let}\ f = \mathbf{fix}^{\Phi_f}(f, x, \mathbf{if}\ \mathbf{read}^{l_R}(x)\ \mathbf{then\ true}\\ \qquad\qquad\qquad\quad \mathbf{else}\ (\mathbf{push}^{l_{Push}}(x); f@^{l_1}x; \mathbf{pop}^{l_{Pop}}(x)))\ \mathbf{in}\\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ f@^{l_2}r\end{array}$$

The usage of r, inferred in a manner similar to the above example, is $\mu\alpha.(l_R; (\mathbf{0} \,\&\, (l_{Push}; \alpha; l_{Pop})))$*. It implies that r is accessed in a stack-like manner: Each access* **push** *is followed by an access* **pop***. This kind of access pattern appears in stacks, JVM lock primitives [2], memory management with reference counting [29] (counter increment corresponds to* **push** *and decrement to* **pop***).*

EXAMPLE 11. *Let us consider the following program:*

$$\begin{array}{l}\mathbf{let}\ f = \mathbf{fix}^{\Phi_f}(f, g, g@^{l_1}\mathbf{true}; f@^{l_2}g)\\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ f@^{l_3}\lambda^{\Phi_g}x.\mathbf{read}^{l_r}(r)\end{array}$$

It first creates a new resource r, and passes to f a function to access the resource. f calls the function repeatedly, forever. Let the types of f and r in $f@^{l_3}\lambda^{\Phi_g}x.\mathbf{read}^{l_r}(r)$ *be* $((\mathbf{bool} \rightarrow \mathbf{bool}, \alpha_g) \rightarrow \mathbf{bool}, \alpha_f)$ *and* $(\mathbf{R}, \alpha_r)$*. Then, we get the following constraints on* α_g *and* α_r*:*

$$\{\alpha_g \leq l_1; \alpha_g,\ \alpha_r \leq \blacksquare\square(\alpha_g \odot l_r)\}.$$

From this, we get:

$$\alpha_g = \mu\alpha.(l_1; \alpha), \alpha_r = (\mu\alpha.(l_1; \alpha)) \odot l_r (= \mu\alpha.(l_r; \alpha))$$

So, we know that r is accessed at l_r infinitely many times. (As a by-product, we also know that the program never terminates, because no trace in $[\![\alpha_r]\!]$ *contains* $\downarrow$*.)*

6. EXTENSIONS

Polymorphism and subtyping As in other type-based analysis, polymorphism on types and usages improves the accuracy of our analysis. Consider the following program:

$$\mathbf{let}\ f = \lambda^{\Phi}x.(\mathbf{acc}_1^{l_1}(x); x)\ \mathbf{in}\ (\mathbf{acc}_2^{l_2}(f@^{l_4}y); \mathbf{acc}_3^{l_3}(f@^{l_5}z))$$

There are two calls of f. The return value of the first call is used at l_2 and that of the second call is used at l_3. So, the best type we can assign to f is $((\mathbf{R}, l_1; (l_2 \,\&\, l_3)) \rightarrow (\mathbf{R}, l_2 \,\&\, l_3), l_4; l_5)$, and the type of y is $(\mathbf{R}, l_1; (l_2 \,\&\, l_3))$. If we introduce polymorphism, we can give f a type $\forall\alpha.((\mathbf{R}, l_1; \alpha) \rightarrow (\mathbf{R}, \alpha), l_4; l_5)$, and we can assign a more accurate type $(\mathbf{R}, l_1; l_2)$ to y. Similarly, our analysis becomes more precise if we relax the subtype relation (see Remark 2).

Dependencies between different variables Our type-based analysis is imprecise when there is an alias. For example, consider the following program:

$$(\mathbf{let}\ y = x\ \mathbf{in}\ (\mathbf{acc}_1^{l_1}(x); \mathbf{acc}_2^{l_2}(y)))^{\{x\}}$$

The type inferred for x is $(\mathbf{R}, \blacksquare(\square l_2; l_1))$ (which is equivalent to $(\mathbf{R}, l_2 \otimes l_1)$). So, we lose information that x is actually used at l_1 and then at l_2. The problem is that a type environment is just a binding of variables to types and it does not keep track of the order of accesses through different variables. To solve the problem, we can extend type environments, following our generic type system for the π-calculus [15]. For example, the type environment of the expression $\mathbf{acc}_1^{l_1}(x); \mathbf{acc}_2^{l_2}(y)$ can be represented as $x : (\mathbf{R}, l_1); y : (\mathbf{R}, l_2)$, which means that x is accessed at l_1, *and then* y is accessed at l_2. Then, we can obtain the type environment of the whole expression by: $[x/y](x : (\mathbf{R}, l_1); y : (\mathbf{R}, l_2)) = x : (\mathbf{R}, l_1; l_2)$.

Combination with region inference Regions and effects [3, 25] are also useful to improve the accuracy of the analysis. Consider a term $(\lambda^{\{l_3\}}y.\mathbf{acc}_1^{l_1}(x))@^{l_3}\mathbf{acc}_2^{l_2}(x)$. The best type we can assign to x is $(\mathbf{R}, \square l_1; l_2)$, although the term is computationally equivalent to $\mathbf{let}\ y = \mathbf{acc}_2^{l_2}(x)\ \mathbf{in}\ \mathbf{acc}_1^{l_1}(x)$. The problem is that rule (T-ABS) loses information that free variables in $\lambda^{\Phi}x.M$ are accessed only after the function is applied.

We can better handle this problem using region and effect systems [3, 25]. Let us introduce a region to express a set of resources, and let r be the region of the resource x above. Then, we can express the type of $\lambda^{\{l_3\}}y.\mathbf{acc}_1^{l_1}(x)$ as $\mathbf{bool} \xrightarrow{r^{l_1}} \mathbf{bool}$, where the latent effect r^{l_1} means that a resource in region r is accessed at l_1 when the function is invoked. Using this precise information, we can obtain $r^{l_2}; r^{l_1}$ as the effect of the whole expression.

A problem of the above method is that since the effect $r^{l_2}; r^{l_1}$ tells only that *some* resource in region r is accessed at l_2 and then *some* resource in region r is accessed at l_1, we don't know whether x is indeed accessed at l_1 and l_2 if r represents multiple resources. Multiple resources are indeed aliased to the same region, for example, when they are passed to the same function:

$$\mathbf{let}\ x = \mathbf{new}()\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{new}()\ \mathbf{in}\ (f(x), f(y))$$

A common solution to this problem is to use region polymorphism, existential types, etc. [5, 25, 28], at the cost of complication of type systems.

We are currently studying a method to combine our analysis with region/effect systems to take the best of both worlds. The resulting analysis would no longer require a separate escape analysis, because region/effect information subsumes escape information.

Recursive data structures It is not difficult to extend our type-based analysis to deal with recursive data structures like lists. For example, we can write $(\mathbf{R}, U)$ **list** for the type of a list of resources used according to U. (Note that in DeLine and Fähndrich's type system [5], existential types are required to express similar information.) The rules for constructing and destructing lists can be given as:

$$\frac{\Gamma_1 \vdash M_1 : \tau \qquad \Gamma_2 \vdash M_2 : \tau\ \mathbf{list}}{\Gamma_1; \Gamma_2 \vdash M_1 :: M_2 : \tau\ \mathbf{list}}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash M_1 : \tau\ \mathbf{list} \\ \Gamma_2 \vdash M_2 : \tau' \qquad \Gamma_3, x : \tau, y : \tau\ \mathbf{list} \vdash M_3 : \tau'\end{array}}{\Gamma_1; (\Gamma_2 \,\&\, \Gamma_3) \vdash \mathbf{case}\ M_1\ \mathbf{of\ nil} \Rightarrow M_2 \mid x :: y \Rightarrow M_3 : \tau'}$$

If we are also interested in how cons cells are accessed, we can further extend the list type to $((\mathbf{R}, U_1)\ \mathbf{list}, U_2)$, which means that each cons cell is accessed according to U_2.

7. RELATED WORK

The goal of the present work is close to that of DeLine and Fähndrich's Vault programming language [5]. Vault's type system keeps track of the state (called a *key*) of a resource. The state of a resource determines what operations can be performed on the resource, and the state changes after operations are performed. Therefore, keys in their type system roughly correspond to usages in our type-based usage analysis. A main difference is that our analysis automatically gather information on resource usage, while their type system requires programmers' explicit type annotations (including keys) to guide an analysis. In fact, Vault's type system seems rather complicated (it requires existential types, etc.), and unsuitable for type inference. On the other hand, annotation of trace sets (Φ) in our framework is only used to declare valid access sequences. This declaration is necessary because the valid access sequences vary depending on the type of each resource. Typically, declaration of a trace set needs to be done only once for each kind of resource. For example, the following program defines *new_ro* and *new_rw* as functions to create a read-only file and a read-write file respectively:

$$\begin{array}{l}\mathbf{let}\ \mathit{new\_ro} = \lambda x.\mathbf{new}^{(l_R^* l_C \downarrow)^\sharp}()\ \mathbf{in} \\ \mathbf{let}\ \mathit{new\_rw} = \lambda x.\mathbf{new}^{((l_R + l_W)^* l_C \downarrow)^\sharp}()\ \mathbf{in}\ \cdots\end{array}$$

Here, we assume that the primitives for reading, writing, and closing a file are annotated with l_R, l_W, and l_C, respectively. Another difference is that resources can have only finite states in Vault, while we can express possibly infinite states (recall Example 10).

Technical ideas of our type-based analysis are similar to the quasi-linear type system [18] for memory management and type systems for concurrent processes (especially, those for deadlock-free processes) [15, 20, 21, 24]. The quasi-linear type system distinguishes between candidates for the last access (labeled with 1) to a heap value and other accesses (labeled with δ or ω), and guarantees that heap values judged to be quasi-linear are never accessed after they are accessed by an operation labeled with 1. Similar typing rules are used to keep track of the access order (although the details are different). The idea of usage expressions was borrowed from type systems for concurrent processes [15, 20, 21, 24]. In those type systems, usage expressions express how each communication channel is used.

As mentioned in Section 1, many pieces of previous work on memory management, safe locking, etc. are related with our resource usage analysis problem. We remark on some of them below; Detailed comparison of our type-based analysis with previous work is left for future work.

The problem of linearity analysis [11, 26, 27, 30] can be viewed as an instance of the resource usage analysis problem: By removing information on label names and access order from usage information, we get linearity information. Our type-based analysis subsumes the linear type system of [13].

Among previous work on region-based memory management, most closely related would be Walker et al's work [28, 29]. Given programs explicitly annotated with region operations, their type system checks the safety of the region operations through a type system. (On the other hand, most of other work on region-based memory management [1, 3, 25] inserts region operations automatically.) However, unlike in our type-based usage analysis, programs have to be explicitly annotated with type information that guides the program analysis in their type system.

Freund and Mitchell [9] proposed a type system for Java bytecode, which guarantees that every object is initialized before being used. Although the problem of checking this property is an instance of the usage analysis problem, our type-based analysis presented in Section 4 is not powerful enough to guarantee the same property. The main difficulty is that in typical Java bytecode, a pointer to an uninitialized object is duplicated into two pointers, one of which is used to initialize the object, and then the other is used to access the object. To deal with this, our analysis must be extended to keep track of dependencies between different variables, as mentioned in Section 6.

8. CONCLUSION

We have formalized a resource usage analysis problem as generalization of various program analysis problems concerning resource access order. Our intention is to provide a uniform view for various problems attacked individually so far, and to stimulate development of general methods to solve those problems. As a starting point towards development of general methods for resource usage analysis, we have also presented a type-based method.

A lot of work is left for future work. In order to deal with various kinds of resources and programming styles, it is probably necessary to extend our type-based method as discussed in Section 6. In fact, our current type-based method does not subsume many solutions proposed for individual problems [9, 28]. It is also left for future work to choose a language appropriate to specify valid trace sets (Φ), and design a practically good algorithm to check that inferred usages conform to the specification (i.e., $[\![U]\!] \subseteq \Phi$). As a specification language for trace sets, we are currently planning to use shuffle expressions [10, 16] or context-free grammars. (Note that regular expressions are a little too weak to express access patterns like that in Example 10.)

We used the call-by-value simply-typed λ-calculus as a target language of our type-based analysis. It would be interesting to develop a method for usage analysis for other languages such as imperative languages, low-level languages (like assembly languages and bytecode languages), and lazy functional languages. A rather different method may be necessary to analyze those languages.

Acknowledgment

We would like to thank Haruo Hosoya, Tatsuro Sekiguchi, and Eijiro Sumii for discussions and comments.

9. REFERENCES

[1] A. Aiken, M. Fähndrich, and R. Levien. Improving region-based analysis of higher-order languages. In *Proc. of PLDI*, pages 174–185, 1995.

[2] G. Bigliardi and C. Laneve. A type system for JVM threads. In *Proc. of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, 2000.

[3] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proc. of POPL*, pages 171–183, 1996.

[4] B. Blanchet. Escape analysis: Correctness, proof, implementation and experimental results. In *Proc. of POPL*, pages 25–37, 1998.

[5] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. of PLDI*, pages 59–69, 2001.

[6] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B*, chapter 16, pages 995–1072. The MIT press/Elsevier, 1990.

[7] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99*, LNCS 1664, pages 288–303. Springer-Verlag, 1999.

[8] C. Flanagan and M. Abadi. Types for safe locking. In *Proc. of ESOP 1999*, LNCS 1576, pages 91–108, 1999.

[9] S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Trans. Prog. Lang. Syst.*, 21(6):1196–1250, 1999.

[10] J. Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Comm. ACM*, 24(9):597–605, 1981.

[11] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of IFL'00, Implementation of Functional Languages*, LNCS 2011, pages 140–157, 2000.

[12] J. Hannan. A type-based analysis for stack allocation in functional languages. In *Proceedings of SAS'95*, LNCS 983, pages 172–188, 1995.

[13] A. Igarashi and N. Kobayashi. Garbage collection based on a linear type system. In *Proc. of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, 2000.

[14] A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Info. Comput.*, 161:1–44, 2000.

[15] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proc. of POPL*, pages 128–141, 2001.

[16] J. Jędrzejowicz and A. Szepietowski. Shuffle languages are in P. *Theor. Comput. Sci.*, 250(1-2):31–53, 2001.

[17] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML Type Reconstruction. In J.-L. Lassez and G. D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. The MIT Press, 1991.

[18] N. Kobayashi. Quasi-linear types. In *Proc. of POPL*, pages 29–42, 1999.

[19] N. Kobayashi. Type-based useless variable elimination. In *Proc. of PEPM*, pages 84–93, 2000.

[20] N. Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *Proc. of IFIP International Conference on Theoretical Computer Science (TCS2000)*, LNCS 1872, pages 365–389, 2000.

[21] N. Kobayashi, E. Sumii, and S. Saito. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR2000*, LNCS 1877, pages 489–503. Springer-Verlag, 2000.

[22] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 66–76, 1995.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[24] E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, ENTCS 16(3), pages 55–77, 1998.

[25] M. Tofte and J.-P. Talpin. Implementation of the call-by-value lambda-calculus using a stack of regions. In *Proc. of POPL*, pages 188–201, 1994.

[26] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 1–11, 1995.

[27] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North Holland, 1990.

[28] D. Walker, K. Crary, and J. G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, 2000.

[29] D. Walker and K. Watkins. On linear types and regions. In *Proc. of ICFP*, 2001.

[30] K. Wansbrough and S. L. P. Jones. Once upon a polymorphic type. In *Proc. of POPL*, pages 15–28, 1999.