Resource Usage Analysis

ATSUSHI IGARASHI
Kyoto University
and
NAOKI KOBAYASHI
Tohoku University

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 eventually be 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 clear what the essence of those methods was 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.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis; operational semantics*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages, Reliability, Theory, Verification

Additional Key Words and Phrases: Resource usage, type inference

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

This is a revised and extended version of a paper presented in the Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL2002). *ACM SIGPLAN Notices, 37*, 1 (Jan.), 331–342.
This work was supported in part by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas Research No. 12133202, 2000.
Authors' addresses: A. Igarashi, Graduate School of Informatics, Kyoto University, Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan; email: igarashi@kuis.kyoto-u.ac.jp; N. Kobayashi, Graduate School of Information Sciences, Tohoku University, 6-3-9 Aoba, Aramaki, Aoba-Ku, Sendai 980-8579, Japan; email: koba@ecei.tohoku.ac.jp.

should eventually be deallocated,[1] 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 property. Type systems for region-based memory management [Aiken et al. 1995; Birkedal et al. 1996; Tofte and Talpin 1994; Walker et al. 2000] ensure that deallocated regions are no longer read or written. Linear type systems [Kobayashi 1999; Turner et al. 1995; Wadler 1990; Wansbrough and Peyton Jones 1999] 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 [Flanagan and Abadi 1999a, 1999b] ensure that appropriate locks will be acquired before a reference cell or a concurrent object is accessed. Freund and Mitchell's [1999] type system for the Java Virtual Machine (JVM) ensures that every object is initialized before it is accessed. Bigliardi and Laneve's type system [Bigliardi and Laneve 2000] for the JVM ensures that an object that has been locked will be eventually unlocked. DeLine and Fähndrich's type system [DeLine and Fähndrich 2001] 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 [2001] work) 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[2]), 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 [2001] type system, 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]Or, a program can just call `exit()` before memory is exhausted.

[2]The term "usage analysis" is also used to refer to linearity analysis [Gustavsson and Svenningsson 2000]. Our resource usage analysis problem can be considered generalization of the problem of linearity analysis.

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 [Nielson et al. 1999]. 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 $\mathit{flow}^{-1}(l) = \{l_1, \ldots, l_n\}$ 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 of an ML-like program:

$$\textbf{let } x = (\textbf{fopen}(s))^{l_o} \textbf{ in } \ldots \textbf{fread}(M^{l_R}) \ldots \textbf{fclose}(N^{l_C}) \ldots.$$

Here, we assume that **fopen** opens a file of name s and returns a file pointer to access the file, and that **fread** (**fclose**, respectively) takes a file pointer as an input and reads (closes, respectively) the file. If $\mathit{flow}^{-1}(l_o) = \{l_R\}$, then we know that the file opened at l_o 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 $\mathit{flow}^{-1}(l_o)$ is $\{l_C, l_R\}$ in the above program. From the flow information, we cannot tell whether the file created at l_o 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 *resource usage analysis* as a problem of (1) computing a function $\mathit{use} \in \mathcal{L} \to 2^{\mathcal{L}^*}$ 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 (2) checking whether $\mathit{use}(l)$ contains only valid access sequences. Let us reconsider the above example:

$$\textbf{let } x = \textbf{fopen}^{l_o}(s) \textbf{ in } \ldots \textbf{fread}^{l_R}(M) \ldots \textbf{fclose}^{l_C}(N) \ldots.$$

(Here, labels are moved to primitives for creating or accessing resources, rather than expressions **fopen**(s), M, or N to be evaluated to file pointers.) If $\mathit{use}(l_o) = \{l_R l_C, l_R l_R l_C\}$, we know that the file opened at l_o may be closed after it is read once or twice, and the file is never read after being closed. On the other hand, if $\mathit{use}(l_o) = \{l_R l_C, l_C l_R\}$, 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 [Tofte and Talpin 1994; Birkedal et al. 1996; Aiken et al. 1995; Walker et al. 2000], 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, respectively) is annotated with l_R (l_W, l_F, respectively). Then, a region-annotated program is correct if $\mathit{use}(l) \subseteq (l_R + l_W)^* l_F$, where the regular expression $(l_R + l_W)^* l_F$ denotes the set of sequences consisting of zero or more occurrences of l_R or l_W followed by one l_F. In linear type systems [Wadler 1990; Turner et al. 1995; Kobayashi 1999; Wansbrough and Peyton Jones 1999], 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, $use(l)$ contains only sequences of length 1. The object initialization is correct [Freund and Mitchell 1999] if for every label l of an (occurrence of) object creation primitive, every sequence in $use(l)$ begins with the label of a primitive for object initialization. The problem of checking usage of lock primitives [Bigliardi and Laneve 2000] is reduced to that of checking whether in every sequence in $use(l)$, 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 $use(l)$ 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 $(\mathbf{File}, U)$, where U, called a *usage*, expresses how the file is accessed. Its syntax is given by:

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

(We shall introduce other usage constructors later.) 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 $(\mathbf{File}, l_1; l_2)$.

A type judgment of our type system is of the usual form $\Gamma \vdash M : \tau$ except that types are extended. Here, while the type τ of M expresses how the resource M should be accessed by the context in which M appears, the type environment Γ expresses how the resources pointed to by free variables should be accessed *during the evaluation of M* (strictly speaking, it is not always the case that those accesses happen during the evaluation of M, as we will see below). For example, $x : (\mathbf{File}, l_R; l_W)$ specifies that the resource x should be read once and then written once. So, $x : (\mathbf{File}, l_R; l_W) \vdash \mathbf{fread}^{l_R}(x); \mathbf{fwrite}^{l_W}(x) : \mathbf{bool}$ is a valid judgment, but $x : (\mathbf{R}, l_R; l_W) \vdash \mathbf{fwrite}^{l_W}(x); \mathbf{fread}^{l_R}(x) : \mathbf{bool}$ is not.

Then, we extend typing rules for the simply typed λ-calculus so that the evaluation order is taken into account. For example, the ordinary rule for let-expressions is:

$$\frac{\Gamma \vdash M : \tau \quad \Gamma, x : \tau \vdash N : \sigma}{\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma}.$$

It is replaced by the following rule:

$$\frac{\Gamma \vdash M : \tau \quad \Delta, x : \tau \vdash N : \sigma}{\Gamma; \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma}.$$

Type environment $\Gamma; \Delta$ (connected by ';') 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 $y : (\mathbf{File}, l_1) \vdash M : \mathbf{bool}$ (which implies that y is a file accessed at l_1 in M) and $y : (\mathbf{File}, l_2), x : \mathbf{bool} \vdash N : \mathbf{bool}$, then we get $y : (\mathbf{File}, l_1; l_2) \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \mathbf{bool}$. 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 $M \triangleq \mathbf{let}\ x = y\ \mathbf{in}\ (\mathbf{fread}^{l_R}(y); \mathbf{fwrite}^{l_W}(x, c))$ where c is a character. If we naively apply the above rule, we get:

$$\frac{\begin{array}{c} y : (\mathbf{File}, l_W) \vdash y : (\mathbf{File}, l_W) \\ y : (\mathbf{File}, l_R), x : (\mathbf{File}, l_W) \vdash \mathbf{fread}^{l_R}(y); \mathbf{fwrite}^{l_W}(x, c) : \mathbf{bool} \end{array}}{(y : (\mathbf{File}, l_W)); (y : (\mathbf{File}, l_R)) \ (= y : (\mathbf{File}, l_W; l_R)) \vdash M : \mathbf{bool}}.$$

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 $y : (\mathbf{File}, l_W)$ occurs not when y is evaluated but when the body of the let-expression $\mathbf{fread}^{l_R}(y); \mathbf{fwrite}^{l_W}(x)$ is evaluated. To solve this problem, we introduce a usage constructor $\diamond U$. Both U and $\diamond U$ mean that the resource must be used according to U, but they differ in the specification about the timing of resource access: If an expression M is to be typed under the assumption that x's usage is U, x must be accessed according to U *now*, when the expression M is evaluated. On the other hand, if x's usage is $\diamond U$, x can be accessed at any time, either when M is evaluated, or when the value of M is used later. Using the constructor $\diamond$, we replace the above inference with:

$$\frac{\begin{array}{c} y : (\mathbf{File}, \diamond l_W) \vdash y : (\mathbf{File}, l_W) \\ y : (\mathbf{File}, l_R), x : (\mathbf{File}, l_W) \vdash \mathbf{fread}^{l_R}(y); \mathbf{fwrite}^{l_W}(x, c) : \mathbf{bool} \end{array}}{y : (\mathbf{File}, \diamond l_W; l_R) \vdash M : \mathbf{bool}}.$$

The premise $y : (\mathbf{File}, \diamond l_W) \vdash y : (\mathbf{File}, l_W)$ reflects the fact that the resource y is accessed only when the value of y is used later (when $\mathbf{fwrite}^{l_W}(x)$ is evaluated). The usage $\diamond l_W; l_R$ in the conclusion means that an access at l_W may occur immediately before an access at l_R occurs, or later after an access at l_R occurs. So, the conclusion implies that y may be accessed at l_W and l_R in any order. (In order to obtain a more accurate usage $l_R; l_W$, we need to keep dependencies between different variables: See Section 7.)

In order to get accurate information about the access order, we also need to have a rule to remove $\diamond$. Suppose that $x : (\mathbf{File}, \diamond l) \vdash M : \tau$ 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:

$$\frac{\Gamma, x : \tau \vdash M : \sigma \quad x \text{ does not escape from } M}{\Gamma, x : \blacklozenge\tau \vdash M : \sigma}.$$

Here, $\blacklozenge$ is a constructor to cancel the $\diamond$-constructor.

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 Article

Section 2 introduces a target language and Section 3 defines the problem of resource usage analysis. After Section 4 presents a type system for resource usage analysis and Section 5 proves its correctness, Section 6 gives a type inference algorithm. Section 7 discusses extensions of the type-based method. Section 8 discusses related work and Section 9 concludes.

2. TARGET LANGUAGE

This section introduces $\lambda^{\mathcal{R}}$, 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 the 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 program execution. We call an element of $\mathcal{L}^{*,\downarrow}$ a *trace*. We write ϵ for the empty sequence, and s_1s_2 for the concatenation of two traces s_1 and s_2. A *trace set*, denoted by the meta-variable Φ, is a subset of $\mathcal{L}^{*,\downarrow}$ that is prefix-closed, that is, $ss' \in \Phi$ implies $s \in \Phi$. $S^\sharp$ denotes the set of all prefixes of elements of S, that is, $\{s \in \mathcal{L}^{*,\downarrow} \mid ss' \in S\}$.

Definition 2.1 (*Terms*). The syntax of $\lambda^{\mathcal{R}}$ terms is given by:

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

Here, we have extended the standard λ-calculus with two constructs: $\mathbf{new}^{\Phi}()$ for creating a new resource and $\mathbf{acc}^{l}(M)$ for accessing resource M. For simplicity, we consider a single kind of resource (hence, the single primitive for resource creation). Also, we assume that access primitives 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 $\lambda r.(\mathbf{let}\ x = \mathbf{acc}^{l}(r)\ \mathbf{in}\ r)$. $\mathbf{fun}(f, x, M)$ 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 $(\mathbf{fun}(f, x, M_2))\ M_1$ (where f is not free in M_2), but we include it to make our type-based analysis in Section 4 more precise (also see Section 7). A formal operational semantics of the language is defined in the next section.

The trace set Φ attached to an occurrence of resource creation primitive represents the programmer's intention on how the 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 the 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 [Gischer 1981; Jędrzejowicz and Szepietowski 2001] context-free grammars, modal logics [Emerson 1990], 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. We often write $M';M$ for $\mathbf{let}\ x = M'\ \mathbf{in}\ M$ when $x \notin \mathbf{FV}(M)$, and write $\lambda x.M$ for $\mathbf{fun}(f, x, M)$ when $f \notin \mathbf{FV}(M)$.

Example 2.2. 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 **acc**.) 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.

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

Here, $\Phi_r = (l_I(l_R + l_W)^* l_F\downarrow)^\sharp$ (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. Alternatively, Φ_r can be a more precise specification $(l_I(l_R l_W)^* l_R l_F\downarrow)^\sharp$. This kind of access pattern (initialized, accessed, and then deallocated) often occurs to various types of resources (e.g., memory, files, Java objects [Freund and Mitchell 1999]).

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 by 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 trace sets.

Definition 3.1.1 (*Heap*). A *heap* H is a function from a finite set of variables to trace sets.

We write $\{x_1 \mapsto \Phi_1, \ldots, x_n \mapsto \Phi_n\}$ (n may be 0) for the heap H such that $dom(H) = \{x_1, \ldots, x_n\}$ and $H(x_i) = \Phi_i$. When $dom(H_1) \cap dom(H_2) = \emptyset$, we

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

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

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

$$(H, \mathcal{E}[\mathbf{fun}(f, x, M)\ v]) \rightsquigarrow (H, \mathcal{E}[[\mathbf{fun}(f, x, M)/f, v/x]M]) \quad \text{(R-App)}$$

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

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

Fig. 1. Reduction rules.

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 Kobayashi [1999], Morrisett et al. [1995], and Turner et al. [1995], 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, using evaluation contexts.

Definition 3.1.2 (*Values, Substitution*). A *value* v is either a variable, $\mathbf{fun}(f, x, M)$, **true**, or **false**. We write $[v_1/x_1, \ldots, v_n/x_n]$ for the standard (simultaneous) capture-avoiding substitution of v_i for x_i.

Definition 3.1.3 (*Evaluation Contexts*). The syntax of evaluation contexts is given by:

$$\mathcal{E} ::= [\,] \mid \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \mid \mathcal{E}\ M \mid v\ \mathcal{E} \mid \mathbf{acc}^{l}(\mathcal{E}) \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ M.$$

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

Definition 3.1.4. 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 the rules in Figure 1. We write $\rightsquigarrow^*$ for the reflexive transitive closure of $\rightsquigarrow$.

Most of the rules are straightforward. In rule R-Acc, 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). Since we do not care about the result of resource access here, it is left unspecified which Boolean value is returned in R-Acc; hence, reduction is nondeterministic. When an ordinary type error like application of a nonfunctional value occurs, the reduction will get stuck.

Example 3.1.5. Let M be the following program, obtained by removing $\mathbf{init}^{l_I}(r)$ from the program in Example 2.2 (let Φ_r be $(l_I(l_R + l_W)^* l_F)^\sharp$):

$$\begin{array}{l}\mathbf{let}\ f = \mathbf{fun}(f, x, \mathbf{if\ read}^{l_R}(x)\ \mathbf{then\ free}^{l_F}(x)\ \mathbf{else}\ (\mathbf{write}^{l_W}(x);\ f\ x))\ \mathbf{in}\\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ f\ r.\end{array}$$

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

$$\begin{array}{ll} & (\{\}, M)\\ \rightsquigarrow^* & (\{z \mapsto (l_I(l_R + l_W)^* l_F \downarrow)^\sharp\}, \mathbf{fun}(f, x, \mathbf{if\ read}^{l_R}(x)\ \mathbf{then}\ \cdots\ \mathbf{else} \cdots)\, z)\\ \rightsquigarrow & (\{z \mapsto (l_I(l_R + l_W)^* l_F \downarrow)^\sharp\},\ \mathbf{if\ read}^{l_R}(z)\ \mathbf{then}\ \cdots\ \mathbf{else} \cdots)\\ \rightsquigarrow & \mathbf{Error}.\end{array}$$

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

Since the problem is undecidable, the resource usage analysis technique developed here is only sound (not 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 3.2.2. The program M in Example 2.2 is resource-safe.

Example 3.2.3. Let M be the following program, obtained from the program in Example 2.2 by replacing $\mathbf{free}^{l_F}(x)$ in the definition of f with **true** (let Φ_r be $(l_I(l_R + l_W)^* l_F)^\sharp$):

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

It is evaluated as follows:

$$\begin{array}{ll} & (\{\}, M)\\ \rightsquigarrow^* & (\{z \mapsto \{(l_R + l_W)^* l_F \downarrow\}^\sharp\}, \mathbf{if\ true\ then\ true}\\ & \qquad\qquad \mathbf{else}\ (\mathbf{write}^{l_W}(z); \mathbf{fun}(f, x, \ldots)\, z))\\ \rightsquigarrow & (\{z \mapsto \{(l_R + l_W)^* l_F \downarrow\}^\sharp\}, \mathbf{true}).\end{array}$$

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 3.2.1 is violated).

According to the second condition of Definition 3.2.1, a resource-safe program must use up all resources before it terminates; For example, the program must close all files of usage $(l_R + l_W)^* l_C$. If a programmer wants to rely on the operating system to close a file, the usage of the file should be specified as $(l_R + l_W)^*(l_C + \epsilon)$ instead of $(l_R + l_W)^* l_C$.

Remark 3.2.4. Alternatively, we can formalize usage analysis as a problem of giving not only a "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 the following sections 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 hinted in Section 1, a main idea is to augment the type of a resource with a usage expression (a usage, in short), which expresses how the resource may be accessed. We first define the syntax and semantics of usages in Section 4.1. We then define types, type environments, and typing rules in Sections 4.2–4.4. Note that programmers need not explicitly declare any usage in their programs: the type inference algorithm described in Section 5 can automatically recover usage information from (untyped) terms.

4.1 Usages

Syntax of Usages. As explained in Section 1, usage expressions defined below are used to describe how each resource can be accessed.

Definition 4.1.1 (*Usages*). The set $\mathcal{U}$ of *usages*, ranged over by U, is defined by:

$$U ::= \mathbf{0} \mid \alpha \mid l \mid U_1 \,\&\, U_2 \mid U_1 \,;U_2 \mid U_1 \otimes U_2 \mid \mu\alpha.U \mid \diamond U \mid \blacklozenge U \mid U_1 \odot U_2$$

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

$\mathbf{0}$ 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. We write $!U$ as a shorthand notation for $\mu\alpha.(\mathbf{0} \,\&\, (U \otimes \alpha))$. As mentioned in Section 1, $\diamond U$ means that the resource may be accessed now or later according to U. So, a resource of usage $\diamond l_1;l_2$ may be accessed either at l_1 and then at l_2, or at l_2 and then at l_1. $\blacklozenge U$ means that the access represented by U must occur *now*. So, for example, $\blacklozenge(\diamond l_1;l_2;\diamond l_3)$ 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, $(l_1 \otimes l_2) \odot U$ is equivalent to $U \otimes U$. The precise meaning of each usage is defined later in this section.

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

Remark 4.1.2. Our usage constructors $\otimes$ and & correspond to multiplicative conjunction and additive conjunction (also denoted by $\otimes$ and &) of linear logic [Girard 1987], respectively. In linear logic, $A \otimes B$ intuitively means that we have A and B at the same time, while $A \& B$ means that we can choose either of A and B, but cannot have both at the same time. This intuition matches the intuition of the usages $U_1 \otimes U_2$ and $U_1 \,\&\, U_2$: $U_1 \otimes U_2$ means that we have both the capability to access a resource according to U_1 and the capability to access a resource according to U_2, while $U_1 \,\&\, U_2$ means that we can choose one from the capability to access a resource according to U_1 and the capability to access a resource according to U_2, but cannot exercise both capabilities.

Example 4.1.3. The usage of a read-only file can be expressed by $\mu\alpha.(\mathbf{0} \,\&\, (l_R;\alpha));l_C$ (or $(!l_R);l_C$), while that of a writable file can be expressed by $\mu\alpha.(\mathbf{0} \,\&\, ((l_R \,\&\, l_W);\alpha));l_C$ (or $!(l_R \,\&\, l_W);l_C$). The usage of a stack is expressed by $!(l_{push};l_{pop})$, and l_{push} and l_{pop} are the labels for the push and pop primitives respectively.

The usage expressions are strictly more expressive than the context-free grammar due to the presence of $\otimes$ and recursion. One may wonder why we do not use a simpler language (like a regular language) for describing usages. There are two reasons for this:

(1) Usage of some resources cannot be specified using a regular expression. For example, consider a stack-like resource, on which the "pop" operation should be performed the same number of times as the "push" operation.
(2) Even if the usage of a resource can be specified using a regular expression (as we have shown in the example of files), the usage of the resource in a *certain part* of the program may not be expressed using a regular expression. For example, consider the following recursive function (where b is some Boolean expression that does not contain any access to x):

$$\mathbf{fun}(f, x, \mathbf{if}\ b\ \mathbf{then\ true\ else}\ (g(x); f(x); h(x)).$$

Function g is first applied to the argument x of the function, and then h is applied the same number of times. In order to perform type reconstruction, we need to be able to assign a *most general* type for each expression. Using regular expressions, however, we cannot assign the most general type to the above function. The type judgment

$$g : (\mathbf{R}, \alpha_g) \to \mathbf{bool}, h : (\mathbf{R}, \alpha_h) \to \mathbf{bool} \vdash \mathbf{fun}(f, x, \ldots) : (\mathbf{R}, \alpha_g^* \alpha_h^*) \to \mathbf{bool}$$

is correct but there are type judgments that express more precise information:

$$\begin{array}{l} g : (\mathbf{R}, \alpha_g) \to \mathbf{bool}, h : (\mathbf{R}, \alpha_h) \to \mathbf{bool} \\ \quad \vdash \mathbf{fun}(f, x, \ldots) : (\mathbf{R}, \epsilon + \alpha_g\alpha_h + \alpha_g\alpha_g^+\alpha_h\alpha_h^+) \to \mathbf{bool} \\ g : (\mathbf{R}, \alpha_g) \to \mathbf{bool}, h : (\mathbf{R}, \alpha_h) \to \mathbf{bool} \\ \quad \vdash \mathbf{fun}(f, x, \ldots) : (\mathbf{R}, \epsilon + \alpha_g\alpha_h + \alpha_g\alpha_g\alpha_h\alpha_h + \alpha_g^2\alpha_g^+\alpha_h^2\alpha_h^+) \to \mathbf{bool} \\ \ldots \end{array}$$

The above example suggests that we need at least a context-free language to express the most general typing. In fact, in our type system, the function is typed as:

$$\begin{array}{l} g : (\mathbf{R}, \alpha_g) \to \mathbf{bool}, h : (\mathbf{R}, \alpha_h) \to \mathbf{bool} \\ \quad \vdash \mathbf{fun}(f, x, \ldots) : (\mathbf{R}, \mu\alpha.(\mathbf{0} \,\&\, (\alpha_g;\alpha;\alpha_h))) \to \mathbf{bool}, \end{array}$$

where the usage $\mu\alpha.(\mathbf{0}\,\&\,(\alpha_g;\alpha;\alpha_h))$ denotes sequences of the form $\alpha_g^n\alpha_h^n$. Moreover, as we have already explained in Section 1, we need the $\diamond$-constructor for expressing information about not only the order between accesses but also the timing of accesses.

A usage constructor $\odot$ is necessary for expressing usage of a resource accessed through the invocation of a function closure. For example, consider a function $\lambda y.\mathbf{read}^{l_R}(x)$. The resource x is read *once each time the function is called*. Therefore, if the function is called n times, the resource x is read n times. More generally, if x is accessed according to U in an expression e, and the function $\lambda y.e$ is called n times, the usage of x is expressed by:

$$\underbrace{U\otimes\cdots\otimes U}_{n}\,.$$

Since we may not be able to statically determine exactly how often each function is called, we express information about how often a function may be called by using usage expressions (but with only a special label 1, as we are only interested in how often a function is called, not in the call sites[3]). For example, the usage of a function that may be called at most twice is expressed by the usage $\mathbf{0}\,\&\,1\,\&\,(1\otimes 1)$. The usage of a resource in a function closure can be computed from the usage of the function closure (expressing how often the function may be called) and the usage of a resource in the function body: If x is accessed according to U in an expression e, and the function $\lambda y.e$ is called according to U', the usage of x is expressed by $U'\odot U$. Intuitively, the usage:

$$(\underbrace{1\otimes\cdots\otimes 1}_{n})\odot U$$

expresses

$$\underbrace{U\otimes\cdots\otimes U}_{n},$$

and the usage:

$$(U_1\,\&\cdots\&\,U_n)\odot U$$

expresses

$$(U_1\odot U)\,\&\cdots\&\,(U_n\odot U).$$

We should note that ordering information in the usage of a function is not as useful as might be expected, to estimate the usage of the resource referred to by a free variable in this function. Suppose, for instance, x is accessed according to U in an expression e. Even if the usage of the function $\lambda y.e$ is given $1\,;1$ (the order between the two calls is known), the usage of x is *not* necessarily $U\,;U$. As it is usually the case for recursive functions, the same (nonrecursive) function may be called twice before the execution of the first call is finished. Thus, we estimate the usage of x to be $U\otimes U$ and define the semantics of usages so that $(1\,;1)\odot U$ is equivalent to $U\otimes U$.

[3]As we will see in Section 4.2, usages including only 1 as a label are attached to function types.

$$U_1 \,\&\, U_2 \preceq U_1 \qquad U_1 \,\&\, U_2 \preceq U_2 \qquad \mu\alpha.U \preceq [\mu\alpha.U/\alpha]U \qquad \frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1\,;U_2 \preceq U_1'\,;U_2'}$$

$$\frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1 \otimes U_2 \preceq U_1' \otimes U_2'} \qquad \frac{U \preceq U'}{\Diamond U \preceq \Diamond U'} \qquad \frac{U \preceq U'}{\blacklozenge U \preceq \blacklozenge U'} \qquad \frac{U_1 \preceq U_1' \quad U_2 \preceq U_2'}{U_1 \odot U_2 \preceq U_1' \odot U_2'}$$

Fig. 2. Relation $U \preceq U'$.

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. In what follows, we assume the meta-variable l ranges over $\mathcal{L} \cup \{1\}$.

We shall define a transition relation $U \xrightarrow{l} U'$, which means that a resource of usage U can be first accessed at l and then accessed according to U'. The transition relation is basically defined in a manner similar to those for process calculi [Milner 1989; Sangiorgi and Walker 2001]. A little complication, however, arises for defining the semantics of usage $U_1;U_2$. A resource of usage $U_1;U_2$ can be used according to U_2 only if U_1 no longer contains accesses that must be performed immediately. So, the usage $l_R;l_W$ should have only the transition sequence:

$$l_R;l_W \xrightarrow{l_R} l_W \xrightarrow{l_W} \mathbf{0}$$

since l_R means that the resource must be read immediately, while the usage $\Diamond l_R;l_W$ should have two transition sequences:

$$\Diamond l_R;l_W \xrightarrow{l_R} l_W \xrightarrow{l_W} \mathbf{0}$$
$$\Diamond l_R;l_W \xrightarrow{l_W} \Diamond l_R \xrightarrow{l_R} \mathbf{0},$$

since $\Diamond l_R$ means that the read access may be delayed. In general, the part U_2 in usage $U_1;U_2$ can be reduced only when all the accesses specified in U_1 are guarded by $\Diamond$. We express this condition by a unary predicate $U_1{}^{\Downarrow}$, defined below.

Before defining the transition relation, we first define auxiliary relations, including $U_1{}^{\Downarrow}$ mentioned above.

Definition 4.1.4. A relation $\preceq$ is the least reflexive and transitive relation on usages that satisfies the rules in Figure 2.

$U_1 \preceq U_2$ holds when U_2 is obtained from U_1 by unfolding some recursive usages ($\mu\alpha.U$) and removing some branches from choices ($U \,\&\, U'$). For example, $l_1;(l_2 \,\&\, l_3) \preceq l_1;\ l_2$ and $\mu\alpha.(\mathbf{0} \,\&\, (U;\alpha)) \preceq \mathbf{0} \,\&\, (U;\ \mu\alpha.(\mathbf{0} \,\&\, (U;\alpha))) \preceq U;\ \mu\alpha.(\mathbf{0} \,\&\, (U;\alpha))$.

Definition 4.1.5. Unary relations $\mathit{void}(\cdot)$, $\cdot^{\downarrow}$ and $\cdot^{\Downarrow}$ are the least relations on usages that satisfy the rules in Figure 3.

Intuitively, $\mathit{void}(U)$ means that the resource cannot be used at all. In other words, $\mathit{void}(U)$ holds if U expresses essentially the same usage as $\mathbf{0}$. For

$void(U)$:

$$void(\mathbf{0}) \qquad \frac{void(U)}{void(\Diamond U)} \qquad \frac{void(U)}{void(\blacklozenge U)} \qquad \frac{void(U)}{void(U \odot U')} \qquad \frac{void(U)}{void(U' \odot U)}$$

$$\frac{\mathbf{op} = \otimes \text{ or } ; \text{ or } \& \qquad void(U_1) \qquad void(U_2)}{void(U_1 \,\mathbf{op}\, U_2)} \qquad \frac{void([\mu\alpha.U/\alpha]U)}{void(\mu\alpha.U)}$$

$U^{\downarrow}$:

$$\frac{U \preceq U' \qquad void(U')}{U^{\downarrow}}$$

$U^{\Downarrow}$:

$$(\Diamond U)^{\Downarrow} \qquad \frac{void(U)}{U^{\Downarrow}} \qquad \frac{U^{\Downarrow}}{(U' \odot U)^{\Downarrow}} \qquad \frac{\mathbf{op} = \otimes \text{ or } ; \text{ or } \& \qquad U_1{}^{\Downarrow} \qquad U_2{}^{\Downarrow}}{(U_1 \,\mathbf{op}\, U_2)^{\Downarrow}} \qquad \frac{([\mu\alpha.U/\alpha]U)^{\Downarrow}}{(\mu\alpha.U)^{\Downarrow}}$$

Fig. 3. Predicates *void*(U), $U^{\downarrow}$, and $U^{\Downarrow}$.

$$l \xrightarrow{l} \mathbf{0} \quad \text{(UR-Zero)} \qquad \frac{U \xrightarrow{l} U'}{\Diamond U \xrightarrow{l} \Diamond U'} \quad \text{(UR-Box)}$$

$$\frac{U_1 \xrightarrow{l} U_1'}{U_1 \otimes U_2 \xrightarrow{l} U_1' \otimes U_2} \quad \text{(UR-ParL)} \qquad \frac{U \xrightarrow{l} U'}{\blacklozenge U \xrightarrow{l} \blacklozenge U'} \quad \text{(UR-Unbox)}$$

$$\frac{U_2 \xrightarrow{l} U_2'}{U_1 \otimes U_2 \xrightarrow{l} U_1 \otimes U_2'} \quad \text{(UR-ParR)} \qquad \frac{U_1 \xrightarrow{l_1} U_1' \qquad U_2 \xrightarrow{l_2} U_2'}{U_1 \odot U_2 \xrightarrow{l_2} U_2' \otimes (U_1' \odot U_2)} \quad \text{(UR-Mult)}$$

$$\frac{U_1 \xrightarrow{l} U_1'}{U_1 \,;\, U_2 \xrightarrow{l} U_1' \,;\, U_2} \quad \text{(UR-SeqL)} \qquad \frac{U \preceq U'' \qquad U'' \xrightarrow{l} U'}{U \xrightarrow{l} U'} \quad \text{(UR-PCong)}$$

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

Fig. 4. Usage reduction rules.

example, *void*$(\mathbf{0} \,\&\, \Diamond\mathbf{0})$ holds. $U^{\downarrow}$ means that some branch of the usage is equivalent to $\mathbf{0}$, and thus the resource need not be accessed before evaluation of the whole term terminates. For example, $(\mathbf{0} \,\&\, l_R)^{\downarrow}$ holds, although *void*$(\mathbf{0} \,\&\, l_R)$ does not hold.

Now we define the transition relation $U \xrightarrow{l} U'$.

Definition 4.1.6. A transition relation $U \xrightarrow{l} U'$ on usages is the least relation closed under the rules in Figure 4.

The rules (UR-ParR) and (UR-SeqR) highlight the difference between $U_1 \otimes U_2$ and $U_1 ; U_2$: (UR-ParR) allows a resource of usage $U_1 \otimes U_2$ to be accessed according to U_2 without any condition, while (UR-SeqR) requires $U^{\Downarrow}$, which specifies that U_1 does not contain any obligation to access the resource immediately, in order for a resource of usage $U_1 ; U_2$ to be accessed according to U_2. As shown in the rules (UR-Box) and (UR-Unbox), the constructors $\diamond$ and $\blacklozenge$ do not directly affect the transition of a usage. Those constructors affects only the side condition $U_1{}^{\Downarrow}$ in the rule (UR-SeqR). The premise $U_1 \xrightarrow{l_1} U_1'$ of the rule (UR-Mult) (actually l_1 is always the same usage 1 in our type system) implies that a resource of $U_1 \odot U_2$ can be used according to $U_2 \otimes (U_1' \odot U_2)$ (recall that $(1 \otimes \cdots \otimes 1) \odot U_2$ intuitively means $U_2 \otimes \cdots \otimes U_2$). The other premise $U_2 \xrightarrow{l_2} U_2'$ means that a resource of U_2 may be used first at l_2 and then U_2'. So, a resource of usage $U_1 \odot U_2$, which subsumes $U_2 \otimes (U_1' \odot U_2)$, may be first used at l_2 and then used according to $U_2' \otimes (U_1' \odot U_2)$, as specified in the conclusion of rule (UR-Mult). Rule (UR-PCong) allows elimination of & and expansion of recursive usages to be performed before the reduction. For example, we can derive $l_1 \,\&\, l_2 \xrightarrow{l_1} \mathbf{0}$ by:

$$\frac{l_1 \,\&\, l_2 \preceq l_1 \quad l_1 \xrightarrow{l_1} \mathbf{0}}{l_1 \,\&\, l_2 \xrightarrow{l_1} \mathbf{0}}.$$

Example 4.1.7. $\diamond l_1; l_2$ has two transition sequences: $\diamond l_1; l_2 \xrightarrow{l_1} \diamond \mathbf{0}; l_2 \xrightarrow{l_2} \diamond \mathbf{0}; \mathbf{0}$ and $\diamond l_1; l_2 \xrightarrow{l_2} \diamond l_1; \mathbf{0} \xrightarrow{l_1} \diamond \mathbf{0}; \mathbf{0}$ 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).)

$\blacklozenge(\diamond l_1; l_2); l_3$ has two transition sequences:

$$\blacklozenge(\diamond l_1; l_2); l_3 \xrightarrow{l_1} \blacklozenge(\diamond \mathbf{0}; l_2); l_3 \xrightarrow{l_2} \blacklozenge(\diamond \mathbf{0}; \mathbf{0}); l_3 \xrightarrow{l_3} \blacklozenge(\diamond \mathbf{0}; \mathbf{0}); \mathbf{0}$$
$$\blacklozenge(\diamond l_1; l_2); l_3 \xrightarrow{l_2} \blacklozenge(\diamond l_1; \mathbf{0}); l_3 \xrightarrow{l_1} \blacklozenge(\diamond \mathbf{0}; \mathbf{0}); l_3 \xrightarrow{l_3} \blacklozenge(\diamond \mathbf{0}; \mathbf{0}); \mathbf{0}$$

$(1;1) \odot (l_1; l_2)$ has the following transition sequences (For the sake of readability, we shall simply write U for $\mathbf{0}; U$):

$$\begin{aligned}(1;1) \odot (l_1; l_2) &\xrightarrow{l_1} l_2 \otimes (1 \odot (l_1; l_2))\\ &\xrightarrow{l_2} \mathbf{0} \otimes (1 \odot (l_1; l_2))\\ &\xrightarrow{l_1} \mathbf{0} \otimes l_2 \otimes (\mathbf{0} \odot (l_1; l_2))\\ &\xrightarrow{l_2} \mathbf{0} \otimes \mathbf{0} \otimes (\mathbf{0} \odot (l_1; l_2)).\end{aligned}$$

$$\begin{aligned}(1;1) \odot (l_1; l_2) &\xrightarrow{l_1} l_2 \otimes (1 \odot (l_1; l_2))\\ &\xrightarrow{l_1} l_2 \otimes l_2 \otimes (\mathbf{0} \odot (l_1; l_2))\\ &\xrightarrow{l_2} \mathbf{0} \otimes l_2 \otimes (\mathbf{0} \odot (l_1; l_2))\\ &\xrightarrow{l_2} \mathbf{0} \otimes \mathbf{0} \otimes (\mathbf{0} \odot (l_1; l_2)).\end{aligned}$$

So, $(1;1) \odot (l_1; l_2)$ has the same transition sequences as $(l_1; l_2) \otimes (l_1; l_2)$.

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

Definition 4.1.8. Let U be a usage. $[\![\, U \,]\!]$ denotes the set:

$$\begin{aligned}&\{l_1 \cdots l_n \mid \exists U_1, \ldots, U_n.(U \xrightarrow{l_1} U_1 \cdots U_{n-1} \xrightarrow{l_n} U_n)\}\\ &\cup \{l_1 \cdots l_n \downarrow \mid \exists U_1, \ldots, U_n.((U \xrightarrow{l_1} U_1 \cdots U_{n-1} \xrightarrow{l_n} U_n) \wedge U_n{}^{\downarrow})\}.\end{aligned}$$

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 4.1.9

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

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.

In order for $U_1 \leq U_2$ to hold, the condition $[\![U_1]\!] \subseteq [\![U_2]\!]$ is not sufficient. For example, $[\![\diamond l]\!] = [\![l]\!] = \{\epsilon, l, l\downarrow\}$ holds, but l should not be considered a subusage of $\diamond l$: Note that l, which says that the resource must be accessed *now*, expresses a more restrictive usage than $\diamond l$, which says that the resource may be accessed at any time. We, therefore, require 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 $C[U] = \mu\alpha.(U;\alpha)$. Let $C = [\,];l'$. Then, $[\![C[l]]\!] = \{\epsilon, l, ll', ll'\downarrow\}$ and $[\![C[\diamond l]]\!] = \{\epsilon, l, ll', l'l, ll'\downarrow, l'l\downarrow\}$, so that usages l and $\diamond l$ can be distinguished.

Using the usage contexts, one could define the subusage relation by: $U_1 \leq U_2$ if and only if $[\![C[U_1]]\!] \subseteq [\![C[U_2]]\!]$ for any usage context C. We, however, impose a stronger condition for the convenience of proving type soundness.

Definition 4.1.10. The *subusage* relation $\leq$ is the largest binary relation such that for any usages U_1 and U_2, 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'.
(3) If $U_2{}^{\downarrow}$, then $U_1{}^{\downarrow}$.

Intuitively, U_1 is a subusage of U_2 if for any context C, every transition step of $C[U_1]$ is simulated by a transition of $C[U_2]$. It is trivial that if $U_1 \leq U_2$ holds, then $[\![C[U_1]]\!] \subseteq [\![C[U_2]]\!]$ holds for any usage context C.

We write $U_1 \cong U_2$ if and only if $U_1 \leq U_2$ and $U_2 \leq U_1$. Some properties of $\leq$ and usage constructors, including reflexivity, transitivity of $\leq$, commutativity and associativity of $\otimes$, etc. are shown in Appendix A.

Example 4.1.11. $U \leq \mu\alpha.\alpha$ holds for any U. $U_1 \,\&\, U_2 \leq U_1$ holds for any U_1 and U_2. $U \cong U \otimes \mathbf{0}$ holds for any U. More laws are given in Appendix A.

Example 4.1.12. $l \leq \mathbf{0}$ does not hold, since $\mathbf{0}^{\downarrow}$ holds but $l^{\downarrow}$ does not hold, which violates the third condition of Definition 4.1.10.

4.2 Types

Now we introduce the syntax of types. As explained above, we associate both resource types and function types with usages.

Definition 4.2.1 (*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, $(\mathbf{bool} \to \mathbf{bool}, 1 \otimes 1)$ is the type of a Boolean function that is called twice. $(\mathbf{R}, U)$ 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 $\mathit{Use}(\mathbf{R}, U) = U$.

We extend the subusage relation to the following subtype relation on types. As usual, τ_1 is a subtype of τ_2, written $\tau_1 \leq \tau_2$, when a value of type τ_1 may be used as a value of type τ_2.

Definition 4.2.2. The *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 4.2.3. Actually, we could relax the above subtype relation by replacing rule (Sub-Fun) with the following rule.

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

The replacement would make our type-based analysis more precise. We did not do so in this article for the sake of simplicity.

4.3 Type Judgments and Type Environments

We consider a type judgment of the form $\Gamma \vdash M : \tau$, where Γ is a type environment, which 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 $x \notin \mathit{dom}(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Δ such that $\mathit{dom}(\Delta) = \mathit{dom}(\Gamma) \cup \{x\}$, $\Delta(x) = \tau$, and $\Delta(y) = \Gamma(y)$ for $y \in \mathit{dom}(\Gamma)$.

A type environment specifies how the resources pointed to by free variables should be accessed. For example, $x : (\mathbf{R}, l_R)$, $y : (\mathbf{R}, l_W)$ specifies that the resource x should be read once, and y should be written once. The type environment $x : (\mathbf{R}, l_R; l_W)$ specifies that the resource x should be first read once and then written once. A type judgment $\Gamma \vdash M : \tau$ means that the expression M obeys the resource usage specified by Γ, and evaluates to a value of type τ. So, $x : (\mathbf{R}, l_R; l_W) \vdash \mathbf{read}^{l_R}(x); \mathbf{write}^{l_W}(x) : \mathbf{bool}$ is a valid judgment, but $x : (\mathbf{R}, l_R; l_W) \vdash \mathbf{write}^{l_W}(x); \mathbf{read}^{l_R}(x) : \mathbf{bool}$ is not.

We introduce operations on type environments so that a complex specification of resource usage may be constructed from simpler specifications. For example, the type environment $\Gamma_1; \Gamma_2$, defined below, specifies that resources should be first accessed according to Γ_1 and then according to Γ_2. As explained in Section 1, these operations are useful for defining typing rules. We also define relations on type environments.

Definition 4.3.1 (*Operations on Types and Type Environments*). 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:

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

Let **op** be a binary usage constructor ';' or '&'. It is extended to operations on types and type environments by:

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

$$\begin{aligned} dom(\Gamma_1\,\mathbf{op}\,\Gamma_2) &= dom(\Gamma_1) \cup dom(\Gamma_2) \\ (\Gamma_1\,\mathbf{op}\,\Gamma_2)(x) &= \begin{cases} \Gamma_1(x)\,\mathbf{op}\,\Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x)\,\mathbf{op}\;\mathbf{0} & \text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2) \\ \mathbf{0}\;\mathbf{op}\;\Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \backslash dom(\Gamma_1). \end{cases} \end{aligned}$$

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

$$\blacklozenge_x \Gamma = \begin{cases} \Gamma & \text{if } x \notin dom(\Gamma) \\ \Gamma', x : (\mathbf{R}, \blacklozenge U) & \text{if } \Gamma = \Gamma', x : (\mathbf{R}, U). \end{cases}$$

Note that if $\Gamma(x) = \mathbf{bool}$ or $\Gamma(x) = (\tau_1 \to \tau_2, U)$, then $\blacklozenge_x \Gamma$ is undefined.

Example 4.3.2. Let Γ be $x : (\mathbf{R}, U)$ and Δ be $x : (\mathbf{R}, U')$, $y : \mathbf{bool}$. Then, $\Diamond\Gamma = \Diamond[\Gamma] = x : \Diamond(\mathbf{R}, U) = x : (\mathbf{R}, \Diamond U)$ and $\Gamma; \Delta = x : ((\mathbf{R}, U); (\mathbf{R}, U'))$, $y : (\mathbf{0};\ \mathbf{bool}) = x : (\mathbf{R}, U; U')$, $y : \mathbf{bool}$.

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 $x \in dom(\Gamma_1) \backslash dom(\Gamma_2)$.

4.4 Typing

Now we introduce typing rules to define the type judgment relation $\Gamma \vdash M : \tau$.

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

$$x : \Diamond\tau \vdash x : \tau \qquad \text{(T-Var)}$$

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

$$\frac{\Gamma, f : (\tau_1 \to \tau_2, U_1), x : \tau_1 \vdash M : \tau_2 \qquad \alpha \text{ fresh}}{(U_2 \odot \mu\alpha.(1 \otimes (U_1 \odot \alpha))) \odot \Diamond\Gamma \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_2)} \qquad \text{(T-Fun)}$$

$$\frac{\Gamma_1 \vdash M_1 : (\tau_1 \to \tau_2, 1) \qquad \Gamma_2 \vdash M_2 : \tau_1}{\Gamma_1 ; \Gamma_2 \vdash M_1\ M_2 : \tau_2} \qquad \text{(T-App)}$$

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

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

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

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

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

Fig. 5. Typing rules.

As mentioned in Section 1, an escape analysis [Blanchet 1998; Hannan 1995] 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 resource referred to by x is not contained in (is unreachable from) the value of M. For example, $(\mathbf{read}^l(x))^{\{x\}}$ is a valid annotation, but $\mathbf{fun}(f, y, \mathbf{read}^l(x))^{\{x\}}$ is not. A simplest escape analysis to check whether M may be annotated as $M^{\{x\}}$ would be to compare the type of M and that of x, as in variants of linear type system [Wadler 1990; Walker and Watkins 2001]: For example if the type of M is **bool**, x cannot escape from M (in the above sense).

Typing rules are shown in Figure 5. The type judgment relation $\Gamma \vdash M : \tau$ is the least relation closed under those rules. In rule (T-Var), the $\Diamond$-constructor is applied to the type of x in the type environment, because x is used only later, not when x is evaluated.

In rule (T-NEW), the conclusion means that $\mathbf{new}^{\Phi}()$ returns a resource that should be used according to U. The premise $[\![U]\!] \subseteq \Phi$ checks that the usage U conforms to the programmer's specification Φ about how the resource created here should be used.

To understand rule (T-FUN) for recursive functions, it would be helpful to first consider the case of a nonrecursive function $\lambda x.M$. The rule for nonrecursive functions would be:

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{U \odot \diamond\Gamma \vdash \lambda x.M : (\tau_1 \to \tau_2, U)}. \qquad \text{(T-ABS)}$$

The premise $\Gamma, x : \tau_1 \vdash M : \tau_2$ says that, *each time* the function body M is evaluated, a resource pointed to by the formal argument x is accessed according to τ_1 and those pointed to by *free variables* in the function $\lambda x.M$ are accessed according to Γ. While the value of x can vary in each function call, those of free variables are determined when the function closure is created and remain the same during its life time. So, if the function $\lambda x.M$ is called according to U, the resources pointed to by free variables are accessed according to $U \odot \diamond\Gamma$. (The constructor $\diamond$ is necessary since the resources are accessed only later when the function is called.) For example, the following is a derivation for the case where the function is called twice:

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{(1 \otimes 1) \odot \diamond\Gamma (\cong \diamond\Gamma \otimes \diamond\Gamma) \vdash \lambda x.M : (\tau_1 \to \tau_2, 1 \otimes 1)}.$$

In the case of a recursive function, we have to carefully count how often the function is called. The type $(\tau_1 \to \tau_2, U_2)$ in the conclusion means that the function is called according to U_2 from the outside of the function, and the type $(\tau_1 \to \tau_2, U_1)$ in the premise means that each time the function is called, it is recursively called according to U_1 inside the function. Therefore, the function is, in total, called according to:

$$\begin{array}{l} U_2 \odot (1 \otimes U_1 \otimes (U_1 \odot U_1) \otimes (U_1 \odot U_1 \odot U_1) \otimes \cdots) \\ (= U_2 \odot \mu\alpha.(1 \otimes (U_1 \odot \alpha)))\cdot \end{array}$$

(As we have already discussed, ordering information between different function calls is not very useful to estimate resource usage, hence $\otimes$ rather than ;). Thus, the type environment for the function is $(U_2 \odot \mu\alpha.(1 \otimes (U_1 \odot \alpha))) \odot \diamond\Gamma$.[4] For example, if the function is called twice from the outside, and if there is no recursive call, the usage of the function is: $(1 \otimes 1) \odot \mu\alpha.(1 \otimes (\mathbf{0} \odot \alpha)) \cong 1 \otimes 1$. If the function is called once from the outside, and if there is zero or one recursive call, the usage of the function is: $1 \odot \mu\alpha.(1 \otimes ((\mathbf{0} \,\&\, 1) \odot \alpha)) \cong \mu\alpha.(1 \otimes (\mathbf{0} \,\&\, \alpha))$, which means that the function may be called at least once.

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,

[4]A similar calculation is performed in linear type systems [Kobayashi 1999; Igarashi and Kobayashi 2000a, 2000b].

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

Fig. 6. An example of type derivation.

the usage of M_1 must be 1. Similarly, in rule (T-Acc), the usage of M must be l because it is accessed at l.[5]

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 $\Gamma(x)$ should happen now, that is, when M is evaluated. The operator $\blacklozenge_x$ is applied to reflect this fact.

Example 4.4.1. A derivation for the type judgment

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

is shown in Figure 6.

5. TYPE SOUNDNESS

The type system in the last section is sound in the sense that every closed well-typed expression of type τ where $Use(\tau) \leq \mathbf{0}$ is resource-safe, provided that the escape analysis is sound. The condition $Use(\tau) \leq \mathbf{0}$ means that resources contained in the result of the evaluation may no longer be accessed. In this section, after stating the type soundness theorem formally in Section 5.1, we prove the theorem using a technique similar to the one used in Kobayashi's quasi-linear type system [Kobayashi 1999]. We first introduce another operational semantics to the target language—the semantics takes into account not only how but also *where* in the expression each resource is used during evaluation. This alternative semantics, defined in Section 5.2, 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 in Section 5.3. Readers who are not interested in proofs may safely skip Sections 5.2–5.4.

5.1 Type Soundness Theorem

To state the type soundness theorem formally, we first extend the operational semantics of the target language to deal with terms of the form $M^{\{x\}}$. The syntax of evaluation contexts is extended by:

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

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

We add the following reduction rule, which make sure that $M^{\{x\}}$ reduces only when the escape analysis is correct (in other words, if the escape analysis were wrong, evaluation would get stuck):

$$\frac{x \notin \mathbf{FV}(v)}{(H, \mathcal{E}[v^{\{x\}}]) \rightsquigarrow (H, \mathcal{E}[v])}. \qquad \text{(R-ECHECK)}$$

The soundness of our type system is stated as follows:

THEOREM 5.1.1 (*Type Soundness*). *If $\emptyset \vdash M : \tau$ and $Use(\tau) \leq \mathbf{0}$, then M is resource-safe.*

5.2 Dynamic Expressions

We extend the target language with $\mathbf{let_R}$-expressions to express "local" usages of resources and introduce dynamic expressions.

Definition 5.2.1. The set of *dynamic expressions*, ranged over by D, is given by the following syntax:

$$\begin{aligned} D ::= {} & \mathbf{let_R}\ x : U \ \mathbf{in}\ D \mid \mathbf{true} \mid \mathbf{false} \mid x \mid \mathbf{fun}(f, x, M) \mid \mathbf{if}\ D_1\ \mathbf{then}\ D_2\ \mathbf{else}\ D_3 \\ & \mid D_1\ D_2 \mid \mathbf{new}^{\Phi}() \mid \mathbf{acc}^{l}(D) \mid \mathbf{let}\ x = D_1\ \mathbf{in}\ D_2 \mid D^{\{x\}}. \end{aligned}$$

The expression of the form $\mathbf{let_R}\ x : U\ \mathbf{in}\ D$ means that the resource allocated at x is used in D and that U represents the resource's usage *local* to D. We often abbreviate $\mathbf{let_R}\ x_1 : U_1\ \mathbf{in} \cdots \mathbf{let_R}\ x_n : U_n\ \mathbf{in}\ D$ to $\mathbf{let_R}\ \tilde{x} : \tilde{U}\ \mathbf{in}\ D$ and $\mathbf{let_R}\ x_1 : (U_1\,; U_1')\ \mathbf{in} \cdots \mathbf{let_R}\ x_n : (U_n\,; U_n')\ \mathbf{in}\ D$ to $\mathbf{let_R}\ \tilde{x} : (\tilde{U}\,; \tilde{U}')\ \mathbf{in}\ D$.

5.2.1 *Operational Semantics of Dynamic Expressions.* An operational semantics of dynamic expressions is defined by the reduction relation $D \xrightarrow{\xi} E$, in which E is either a dynamic expression or **Error**. The label ξ is either ϵ, which corresponds to a reduction step in the standard semantics given in Section 3, or a variable x, which means the usage of the heap value at x is split and *localized* to subexpressions.

As in the standard semantics, the reduction relation is given by using evaluation contexts, whose syntax is given by:

$$\begin{aligned} \mathcal{E}_D \ ::= {} & [\,] \mid \mathbf{let_R}\ x : U\ \mathbf{in}\ \mathcal{E}_D \mid \mathbf{if}\ \mathcal{E}_D\ \mathbf{then}\ D_1\ \mathbf{else}\ D_2 \mid \mathcal{E}_D\ D \\ & \mid (\mathbf{let_R}\ \tilde{x} : \tilde{U}\ \mathbf{in}\ v)\ \mathcal{E}_D \mid \mathbf{acc}^{l}(\mathcal{E}_D) \mid \mathbf{let}\ x = \mathcal{E}_D\ \mathbf{in}\ D \mid \mathcal{E}_D{}^{\{x\}}. \end{aligned}$$

The reduction rules are given in Figures 7 and 8. We write $D \Longrightarrow E$ for $D \xrightarrow{x_1} \cdots \xrightarrow{x_n} \xrightarrow{\epsilon} E$ and write $D \uparrow$ if D always reduces to an error, that is, if $D \Longrightarrow$ **Error** and there is no D' such that $D \Longrightarrow D'$.

A rule R-*Name* of the standard semantics corresponds to the rule RD-*Name*. Unlike the standard semantics, which keeps track of one global heap, when a heap value at x is accessed, it must be in a local heap binding. A heap binding is pushed into subexpressions by rules RD-*Name*PUSH; If there are more than one subexpressions (as in RD-APPPUSH, RD-IFPUSH, and RD-LETPUSH), the usage U is split to two usages. On the other hand, when usages remain after the

$$\frac{[\![\, U \,]\!] \subseteq \Phi \qquad z \text{ fresh}}{\mathcal{E}_{\mathcal{D}}[\mathbf{new}^{\Phi}()] \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ z : U \ \mathbf{in}\ z]} \qquad \text{(RD-New)}$$

$$\mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ x : U \ \mathbf{in}\ D^{\{x\}}] \xrightarrow{x} \mathcal{E}_{\mathcal{D}}[(\mathbf{let}_{\mathbf{R}}\ x : \Diamond U \ \mathbf{in}\ D)^{\{x\}}] \qquad \text{(RD-EChkPush1)}$$

$$\frac{x \neq y}{\mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ x : U \ \mathbf{in}\ D^{\{y\}}] \xrightarrow{x} \mathcal{E}_{\mathcal{D}}[(\mathbf{let}_{\mathbf{R}}\ x : U \ \mathbf{in}\ D)^{\{y\}}]} \qquad \text{(RD-EChkPush2)}$$

$$\frac{y \notin \mathbf{FV}(v) \qquad \mathbf{FV}(v) \subseteq \{\tilde{x}\}}{\mathcal{E}_{\mathcal{D}}[(\mathbf{let}_{\mathbf{R}}\ \tilde{x} : \tilde{U} \ \mathbf{in}\ v)^{\{y\}}] \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ \tilde{x} : \tilde{U} \ \mathbf{in}\ v]} \qquad \text{(RD-ECheck)}$$

$$\frac{U \leq U_1 \,;U_2}{\mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ x : U \ \mathbf{in}\ D_1\ D_2] \xrightarrow{x} \mathcal{E}_{\mathcal{D}}[(\mathbf{let}_{\mathbf{R}}\ x : U_1 \ \mathbf{in}\ D_1)\ (\mathbf{let}_{\mathbf{R}}\ x : U_2 \ \mathbf{in}\ D_2)]} \qquad \text{(RD-AppPush)}$$

$$\begin{array}{l}\mathcal{E}_{\mathcal{D}}[(\mathbf{let}_{\mathbf{R}}\ \tilde{x_1} : \tilde{U_1} \ \mathbf{in}\ \mathbf{let}_{\mathbf{R}}\ \tilde{x_2} : \tilde{U_2} \ \mathbf{in}\ \mathbf{fun}(f,y,M))\ (\mathbf{let}_{\mathbf{R}}\ \tilde{x_1} : \tilde{U_4} \ \mathbf{in}\ \mathbf{let}_{\mathbf{R}}\ \tilde{x_3} : \tilde{U_3} \ \mathbf{in}\ v)] \\ \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ \tilde{x_1} : (\tilde{U_1}\,;\tilde{U_4}) \ \mathbf{in}\ \mathbf{let}_{\mathbf{R}}\ \tilde{x_2} : \tilde{U_2} \ \mathbf{in}\ \mathbf{let}_{\mathbf{R}}\ \tilde{x_3} : \tilde{U_3} \ \mathbf{in}\ [v/y, \mathbf{fun}(f,y,M)/f]M]\end{array} \qquad \text{(RD-App)}$$

$$\mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ x : U \ \mathbf{in}\ \mathbf{acc}^{l}(D)] \xrightarrow{x} \mathcal{E}_{\mathcal{D}}[\mathbf{acc}^{l}(\mathbf{let}_{\mathbf{R}}\ x : U \ \mathbf{in}\ D)] \qquad \text{(RD-AccPush)}$$

$$\frac{U_i \xrightarrow{l} U_i' \qquad U_k' = U_k \text{ for } k \neq i \qquad b = \mathbf{true} \text{ or } \mathbf{false}}{\mathcal{E}_{\mathcal{D}}[\mathbf{acc}^{l}(\mathbf{let}_{\mathbf{R}}\ \tilde{x} : \tilde{U} \ \mathbf{in}\ x_i)] \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let}_{\mathbf{R}}\ \tilde{x} : \tilde{U'} \ \mathbf{in}\ b]} \qquad \text{(RD-Acc)}$$

$$\frac{\neg \exists U. U_i \xrightarrow{l} U}{\mathcal{E}_{\mathcal{D}}[\mathbf{acc}^{l}(\mathbf{let}_{\mathbf{R}}\ \tilde{x} : \tilde{U} \ \mathbf{in}\ x_i)] \xrightarrow{\epsilon} \mathbf{Error}} \qquad \text{(RD-AccErr)}$$

Fig. 7. Dynamic expressions: Reduction rules (1).

evaluation of subexpressions, they are merged for the rest of computation (as in RD-App, RD-IfT, RD-IfF and RD-Let).

For example, a dynamic expression $\mathbf{let}\ x = \mathbf{new}^{\{l_1+l_2\}^\sharp}()\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^{l}(x))$, which is also an expression, can be reduced as follows:

$$\begin{array}{l}\mathbf{let}\ x = \mathbf{new}^{\{l_1+l_2\}^\sharp}()\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^{l}(x)) \\ \quad \xrightarrow{\epsilon}\ \mathbf{let}\ x = \mathbf{let}_{\mathbf{R}}\ z : l_1 \,\&\, l_2\ \mathbf{in}\ z\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^{l}(x)) \\ \quad \xrightarrow{\epsilon}\ \mathbf{let}_{\mathbf{R}}\ z : l_1 \,\&\, l_2\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^{l}(z)) \\ \quad \xrightarrow{z}\ (\mathbf{let}_{\mathbf{R}}\ z : \mathbf{0}\ \mathbf{in}\ \lambda y.y)\ (\mathbf{let}_{\mathbf{R}}\ z : l_1 \,\&\, l_2\ \mathbf{in}\ \mathbf{acc}^{l}(z)) \\ \quad \xrightarrow{\epsilon}\ (\mathbf{let}_{\mathbf{R}}\ z : \mathbf{0}\ \mathbf{in}\ \lambda y.y)\ (\mathbf{let}_{\mathbf{R}}\ z : \mathbf{0}\ \mathbf{in}\ \mathbf{true}) \\ \quad \xrightarrow{\epsilon}\ \mathbf{let}_{\mathbf{R}}\ z : \mathbf{0}\,;\mathbf{0}\ \mathbf{in}\ \mathbf{true}\end{array}$$

Note that this is not the only reduction sequence: in particular, an error may be yielded earlier than expected due to wrong split of resource bindings. For

$$\frac{U \le U_1\,;U_2}{\begin{array}{l}\mathcal{E}_{\mathcal{D}}[\mathbf{let_R}\ x : U\ \mathbf{in\ if}\ D\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2] \\ \xrightarrow{x} \mathcal{E}_{\mathcal{D}}[\mathbf{if}\ (\mathbf{let_R}\ x : U_1\ \mathbf{in}\ D)\ \mathbf{then}\ (\mathbf{let_R}\ x : U_2\ \mathbf{in}\ M_1)\ \mathbf{else}\ (\mathbf{let_R}\ x : U_2\ \mathbf{in}\ M_2)]\end{array}} \qquad \text{(RD-IfPush)}$$

$$\begin{array}{l}\mathcal{E}_{\mathcal{D}}[\mathbf{if}\ (\mathbf{let_R}\ \tilde{x} : \tilde{U_1}\ \mathbf{in\ let_R}\ \tilde{y} : \tilde{U_2}\ \mathbf{in\ true})\ \mathbf{then}\ (\mathbf{let_R}\ \tilde{x} : \tilde{U_3}\ \mathbf{in}\ D_1)\ \mathbf{else}\ D_2] \\ \qquad \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let_R}\ \tilde{x} : (\tilde{U_1}\,;\tilde{U_3})\ \mathbf{in\ let_R}\ \tilde{y} : \tilde{U_2}\ \mathbf{in}\ D_1]\end{array} \qquad \text{(RD-IfT)}$$

$$\begin{array}{l}\mathcal{E}_{\mathcal{D}}[\mathbf{if}\ (\mathbf{let_R}\ \tilde{x} : \tilde{U_1}\ \mathbf{in\ let_R}\ \tilde{y} : \tilde{U_2}\ \mathbf{in\ false})\ \mathbf{then}\ D_1\ \mathbf{else}\ (\mathbf{let_R}\ \tilde{x} : \tilde{U_3}\ \mathbf{in}\ D_2)] \\ \qquad \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let_R}\ \tilde{x} : (\tilde{U_1}\,;\tilde{U_3})\ \mathbf{in\ let_R}\ \tilde{y} : \tilde{U_2}\ \mathbf{in}\ D_2]\end{array} \qquad \text{(RD-IfF)}$$

$$\frac{U \le U_1\,;U_2 \qquad x \ne y}{\begin{array}{l}\mathcal{E}_{\mathcal{D}}[\mathbf{let_R}\ x : U\ \mathbf{in\ let}\ y = D_1\ \mathbf{in}\ D_2] \\ \xrightarrow{x} \mathcal{E}_{\mathcal{D}}[\mathbf{let}\ y = (\mathbf{let_R}\ x : U_1\ \mathbf{in}\ D_1)\ \mathbf{in\ let_R}\ x : U_2\ \mathbf{in}\ D_2]\end{array}} \qquad \text{(RD-LetPush)}$$

$$\begin{array}{l}\mathcal{E}_{\mathcal{D}}[\mathbf{let}\ z = (\mathbf{let_R}\ \tilde{x} : \tilde{U_1}\ \mathbf{in\ let_R}\ \tilde{y} : \tilde{U_3}\ \mathbf{in}\ v)\ \mathbf{in\ let_R}\ \tilde{x} : \tilde{U_2}\ \mathbf{in}\ M] \\ \qquad \xrightarrow{\epsilon} \mathcal{E}_{\mathcal{D}}[\mathbf{let_R}\ \tilde{x} : (\tilde{U_1}\,;\tilde{U_2})\ \mathbf{in\ let_R}\ \tilde{y} : \tilde{U_3}\ \mathbf{in}\ [v/z]M]\end{array} \qquad \text{(RD-Let)}$$

Fig. 8. Dynamic expressions: reduction rules (2).

example, other possible reduction sequences are:

$$\begin{array}{ll}\multicolumn{2}{l}{\mathbf{let}\ x = \mathbf{new}^{\{l_1+l_2\}^\sharp}()\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^l(x))} \\ \Longrightarrow^* & \mathbf{let_R}\ z : l_1 \,\&\, l_2\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^l(z)) \\ \xrightarrow{z} & (\mathbf{let_R}\ z : l_1 \,\&\, l_2\ \mathbf{in}\ \lambda y.y)\ (\mathbf{let_R}\ z : \mathbf{0}\ \mathbf{in\ acc}^l(z)) \\ \xrightarrow{\epsilon} & \mathbf{Error}\end{array}$$

and

$$\begin{array}{ll}\multicolumn{2}{l}{\mathbf{let}\ x = \mathbf{new}^{\{l_1+l_2\}^\sharp}()\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^l(x))} \\ \Longrightarrow^* & \mathbf{let_R}\ z : l_1 \,\&\, l_2\ \mathbf{in}\ (\lambda y.y)\ (\mathbf{acc}^l(z)) \\ \xrightarrow{z} & (\mathbf{let_R}\ z : \mathbf{0}\ \mathbf{in}\ \lambda y.y)\ (\mathbf{let_R}\ z : l_2\ \mathbf{in\ acc}^l(z)) \\ \xrightarrow{\epsilon} & \mathbf{Error}\end{array}$$

As we show below, however, if an expression has an error-free reduction sequence in the original semantics defined in Section 3, there is at least one error-free reduction sequence in this semantics.

5.2.2 *Typing Rules for Dynamic Expressions.* We extend the type system in Section 4 to dynamic expressions by adding the following rules:[6]

$$\frac{\Gamma, x : (\mathbf{R}, U) \vdash D : \tau}{\Gamma \vdash \mathbf{let_R}\ x : U\ \mathbf{in}\ D : \tau}. \qquad \text{(T-Letres)}$$

[6]Strictly speaking, each occurrence of the meta-variable M in the typing rules of Figure 5 (except for T-Fun) should be replaced with D.

$$
\begin{array}{rcll}
(D)^{\natural} & = & ((B)^{\natural}, M) & \text{where } (B, M) = (D)^{\flat} \\
(\mathbf{let_R}\ x : U \ \mathbf{in}\ D)^{\flat} & = & (B \uplus \{x \mapsto U\}, M) & \text{where } (B, M) = (D)^{\flat} \\
(v)^{\flat} & = & (\{\}, v) & \\
(\mathbf{fun}(f, x, M))^{\flat} & = & (\{\}, \mathbf{fun}(f, x, M)) & \\
(\mathbf{new}^{\Phi}())^{\flat} & = & (\{\}, \mathbf{new}^{\Phi}()) & \\
(\mathbf{if}\ D_1\ \mathbf{then}\ D_2\ \mathbf{else}\ D_3)^{\flat} & = & (B_1 \,;\, B_2, \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3) & \\
 & & & \text{where } (B_i, M_i) = (D_i)^{\flat} \text{ for } i = 1 \ldots 3 \\
 & & & \text{and } B_2 = B_3 \\
(D_1\ D_2)^{\flat} & = & (B_1 \,;\, B_2, M_1\ M_2 & \text{where } (B_i, M_i) = (D_i)^{\flat} \text{ for } i = 1, 2 \\
(\mathbf{acc}^{l}(D))^{\flat} & = & (B, \mathbf{acc}^{l}(M)) & \text{where } (B, M) = (D)^{\flat} \\
(\mathbf{let}\ x = D_1\ \mathbf{in}\ D_2)^{\flat} & = & (B_1 \,;\, B_2, \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) & \\
 & & & \text{where } (B_i, M_i) = (H_i)^{\flat} \text{ for } i = 1, 2 \\
(D^{\{y\}})^{\flat} & = & (\blacklozenge_y B, M^{\{y\}}) & \text{where } (B, M) = (D)^{\flat}
\end{array}
$$

where, $B_1 \,;\, B_2$ is defined by:

$$
\begin{array}{l}
B_1 \,;\, \{\} = B_1 \\
(B_1' \uplus \{x \mapsto U_1\}) \,;\, (B_2' \uplus \{x \mapsto U_2\}) = (B_1' \,;\, B_2') \uplus \{x \mapsto (U_1 \,;\, U_2)\}
\end{array}
$$

and $\blacklozenge_x B$ by:

$$
\begin{array}{ll}
dom(\blacklozenge_x B) = dom(B) & \\
(\blacklozenge_x B)(x) = \blacklozenge B(x) & \\
(\blacklozenge_x B)(y) = B(y) & \text{if } y \neq x
\end{array}
$$

and $(B)^{\natural}$ by:

$$
\begin{array}{l}
dom((B)^{\natural}) = dom(B) \\
(B)^{\natural}(x) = [\![\, B(x) \,]\!]
\end{array}
$$

Fig. 9. Translation of dynamic expressions.

5.3 Properties of Dynamic Expressions

5.3.1 *Correspondence between the Two Semantics.* As is stated below in Theorem 5.3.2, the semantics of dynamic expressions is essentially equivalent to the standard one given in Section 3. Intuitively, the theorem states that (1) program execution (in the original semantics) proceeds as the reduction of a corresponding dynamic expression proceeds; and (2) if there exists an error-free reduction in the semantics of dynamic expressions, then so does a corresponding reduction in the standard semantics.

We first give a few definitions to state correspondence formally: First, we define a translation $(\cdot)^{\natural}$ from dynamic expressions to pairs of a heap and an expression in Figure 9. Here, the meta-variable B ranges over a functions from variables to usages; we use notations $\{x_1 \mapsto U_1, \ldots, x_n \mapsto U_n\}$ or $B_1 \uplus B_2$, defined similarly to $\{x_1 \mapsto \Phi_1, \ldots, x_n \mapsto \Phi_n\}$ or $H_1 \uplus H_2$, respectively. We write $\{\}$ for the empty function. For example,

$$((\mathbf{let_R}\ z : \mathbf{0}\ \mathbf{in}\ \lambda y.y)\,(\mathbf{let_R}\ z : l\ \mathbf{in}\ \mathbf{acc}^{l}(z)))^{\natural} = (\{z \mapsto [\![\, \mathbf{0} \,;\, l \,]\!]\}, (\lambda y.y)\ \mathbf{acc}^{l}(z)).$$

Second, we define the relation $\leq$ between pairs of a heap and an expression:

Definition 5.3.1. The binary relation $\leq$ on heaps is defined by: $H_1 \leq H_2$ if and only if (1) $dom(H_1) = dom(H_2)$; and (2) $H_1(x) \supseteq H_2(x)$. We write $(H_1, M_1) \leq (H_2, M_2)$ if $H_1 \leq H_2$ and $M_1 = M_2$.

Then, the correspondence between (H, M) and D is represented by $(H, M) \leq (D)^\natural$.

Note that, by definition of $D \xrightarrow{\xi} D'$, reduction preserves the following invariants, which guarantee $(\cdot)^\natural$ is well defined: if D contains an expression of the form $\mathbf{let}\ x = D'\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x} : \tilde{U}\ \mathbf{in}\ M$, then $(D')^\flat = (B, M')$ and $B = \{\tilde{x} \mapsto \tilde{U}, \tilde{y} \mapsto \tilde{U}'\}$ and $\tilde{y}$ do not appear in M (similarly for $D_1\ D_2$ and $\mathbf{if}\ D_1\ \mathbf{then}\ D_2\ \mathbf{else}\ D_3$). The condition means that a subexpression being evaluated has extra heap bindings, generated during its evaluation.

The theorem below states that (1) reduction in the original semantics proceeds as reduction of a corresponding dynamic expression proceeds; (2) if an error occurs in the original semantics, so does in the second semantics; (3) for a reduction step in the standard semantics, there may or may not exist the corresponding reduction step. Note that, as discussed above, evaluation of dynamic expressions may cause an error even when that in the standard semantics does not, because usages of one resource are wrongly split.

THEOREM 5.3.2

(1) *If* $D \Longrightarrow D'$ *and* $(H, M) \leq (D)^\natural$*, then* $(H, M) \rightsquigarrow (H', M')$ *and* $(H', M') \leq (D')^\natural$ *for some* H' *and* M'.
(2) *If* $(H, M) \rightsquigarrow \mathbf{Error}$ *and* $(H, M) \leq (D)^\natural$*, then* $D \uparrow$.
(3) *If* $(H, M) \rightsquigarrow (H', M')$ *and* $(H, M) \leq (D)^\natural$*, then either* $D \uparrow$ *or there exists* D' *such that* $D \Longrightarrow D'$ *and* $(H', M') \leq (D')^\natural$.

To prove this theorem, we begin with several required lemmas.

LEMMA 5.3.3

(1) *If* $(\mathcal{E}_D[D_0])^\flat = (B, M)$*, then there exist* $\mathcal{E}$*,* M_0*,* B_0 *and* B_1 *such that* $M = \mathcal{E}[M_0]$ *and* $B = B_0; B_1$ *and* $(D_0)^\flat = (B_0, M_0)$*. Moreover,* $(\mathcal{E}_D[D'_0])^\flat = (B', M')$ *implies* $M' = \mathcal{E}[M'_0]$ *and* $B' = B'_0; B_1$ *and* $(D'_0)^\flat = (B'_0, M'_0)$ *for some* M'_0 *and* B'_0.
(2) *Conversely, if* $(B, \mathcal{E}[M_0]) = (D)^\flat$*, then there exist* $\mathcal{E}_D$*,* $\tilde{x}$*,* $\tilde{U}$*,* D_0*,* B_0 *and* B_1 *such that* $D = \mathcal{E}_D[D_0]$ *and* $B = (B_0; B_1) \uplus \{\tilde{x} \mapsto \tilde{U}\}$ *and* $(D_0)^\flat = (B_0, M_0)$*. Moreover,* $(B', \mathcal{E}[M'_0]) = (D')^\flat$ *implies* $D' = \mathcal{E}_D[D'_0]$ *and* $B' = (B'_0; B_1) \uplus \{\tilde{x} \mapsto \tilde{U}\}$ *and* $(D'_0)^\flat = (B'_0, M'_0)$ *for some* D'_0 *and* B'_0.

PROOF. Easy induction on the structure of $\mathcal{E}_D$ and $\mathcal{E}$. □

LEMMA 5.3.4

(1) *If* $D \xrightarrow{x} D'$*, then* $(D)^\natural \leq (D')^\natural$.
(2) *If* $D \xrightarrow{\epsilon} D'$ *and* $(H, M) \leq (D)^\natural$*, then there exist* H' *and* M' *such that* $(H, M) \rightsquigarrow (H', M')$ *and* $(H', M') \leq (D')^\natural$.

PROOF. By case analysis on the rule used to derive $D \xrightarrow{\xi} D'$, using Lemma 5.3.3 (1). □

PROOF OF THEOREM 5.3.2. (1) follows from Lemma 5.3.4. (2) and (3) are easily shown by case analysis on the rule used to derive $(H, M) \rightsquigarrow (H', M')$, using Lemma 5.3.3 (2). □

5.4 Proof of Theorem 5.1.1

Main theorems are Theorem 5.4.1 that a well-typed expression never causes a usage error and Theorem 5.4.2 that reduction of dynamic expressions preserves typing.

THEOREM 5.4.1. *If* $\Gamma \vdash D : \tau$, *then* $D \overset{\epsilon}{\not\longrightarrow} \mathbf{Error}$.

PROOF. Suppose the reduction step is derived from RD-ACCERR. Then, the premise $\neg\exists U.U_i \xrightarrow{l} U$ contradicts the assumption $\Gamma \vdash D : \tau$. □

THEOREM 5.4.2 (SUBJECT REDUCTION). *If* $\Gamma \vdash D : \tau$ *and* $D \xrightarrow{\xi} D'$ *and* $(D')^\natural = (H', M)$, *then there exists* D'' *such that* $D \xrightarrow{\xi} D''$ *and* $(D'')^\natural = (H'', M)$ *and* $\Gamma \vdash D'' : \tau$.

PROOF. See Appendix B. □

The complication of the statement of Theorem 5.4.2 stems from the fact that even a well-typed dynamic expression may reduce to an ill-typed expression, depending on how a usage is split or on how a reduction step changes the usage of the used value. So, the statement says that there is always a *good* reduction step that preserves the well-typedness of the expression. Moreover, D'' must be the same as D' except for type annotations (this is expressed by the phrases "$(D')^\natural = (H', M)$" and "$(D'')^\natural = (H'', M)$"); It is required since RD-ACC makes reduction nondeterministic.

Finally, Theorem 5.1.1 is shown from Theorems 5.3.2, 5.4.1 and 5.4.2 via the following lemma.

LEMMA 5.4.3. *If* $(H_1, M_1) \rightsquigarrow (H_2, M_2)$ *and* $(H_1, M_1) \leq (D_1)^\natural$ *and* $\emptyset \vdash D_1 : \tau$, *then there exists* D_2 *such that* $D_1 \Longrightarrow D_2$ *and* $(H_2, M_2) \leq (D_2)^\natural$ *and* $\emptyset \vdash D_2 : \tau$.

PROOF. By Theorem 5.3.2(3) and Theorem 5.4.1, there exists D_2' such that $D_1 \Longrightarrow D_2'$ and $(H_2, M_2) \leq (D_2')^\natural$. Furthermore, by Theorem 5.4.2 and the definition of $\leq$, there exist D_2'' and H_2' such that $D_1 \Longrightarrow D_2''$ and $\emptyset \vdash D_2'' : \tau$ and $(D_2'')^\natural = (H_2', M_2)$. By Theorem 5.3.2(1), there exist H_2'' such that $(H_1, M_1) \rightsquigarrow (H_2'', M_2)$ and $H_2'' \leq H_2'$. It is easy to show that $(H_1, M_1) \rightsquigarrow (H_2, M_2)$ and $(H_1, M_2) \rightsquigarrow (H_2'', M_2)$ imply $H_2'' = H_2$, thus, $H_2 \leq H_2'$. Letting $D_2 = D_2''$ finishes the proof. □

PROOF OF THEOREM 5.1.1. For the first condition of the resource safety, let (H_1, M_1) be $(\{\}, M)$ and suppose $(H_1, M_1) \rightsquigarrow \cdots \rightsquigarrow (H_n, M_n) \rightsquigarrow \mathbf{Error}$. Let $D_1 = M$. Then, by Lemma 5.4.3, there exist $D_1, \ldots, D_n$ such that $D_i \Longrightarrow D_{i+1}$ and $(H_i, M_i) \leq (D_i)^\natural$ and $\emptyset \vdash D_i : \tau$. By Theorem 5.3.2(2), $D_n \uparrow$ while, by Theorem 5.4.1, $D_n \overset{\epsilon}{\not\longrightarrow} \mathbf{Error}$. Contradiction.

For the second, by a similar argument, it can be shown that there exist $\tilde{x}$ and $\tilde{U}$ such that $M \Longrightarrow^* \mathbf{let_R}\ \tilde{x} : \tilde{U}\ \mathbf{in}\ v$ and $(H, v) \leq (\mathbf{let_R}\ \tilde{x} : \tilde{U}\ \mathbf{in}\ v)^\natural$ and $\emptyset \vdash \mathbf{let_R}\ \tilde{x} : \tilde{U}\ \mathbf{in}\ v : \tau$. By inspection of the type derivation, $U_i \leq \mathbf{0}$ for any i. Then, by definition of $\leq$, we have $U_i{}^\downarrow$, thus $\downarrow \in H(x_i)$. □

6. A TYPE INFERENCE ALGORITHM

Let M be a closed term. By the type soundness theorem (5.1.1), in order to verify that all resources are used correctly in M, it suffices to verify that $\emptyset \vdash M : \tau$ holds for some type τ with $Use(\tau) \leq \mathbf{0}$. 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. For some specific trace sets, however, it is possible to construct an algorithm for checking $[\![\, U \,]\!] \subseteq \Phi$: See Section 6.6.

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 [Kanellakis et al. 1991; Kobayashi 2000a] 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.

6.1 Step 1: Constructing a Template of a Type Derivation Tree

First, we construct 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, an application of the rule (T-VAR) followed by an application of (T-SUB):

$$\frac{\dfrac{}{x : \diamond\tau \vdash x : \tau}\,(\text{T-VAR}) \quad \Gamma \leq x : \diamond\tau}{\Gamma \vdash x : \tau}\,(\text{T-SUB})$$

is replaced by one rule:

$$\frac{\Gamma \leq x : \diamond\tau}{\Gamma \vdash x : \tau}\,(\text{T-VAR}').$$

The set of syntax-directed typing rules is given in Figure 10.

Remark 6.1.1. Each rule in Figure 10 is obtained by combining each rule in Section 4 with the subsumption rule (T-SUB) applied *after* that rule. Alternatively, we can obtain a syntax-directed rule by combining each rule with the subsumption rule (T-SUB) applied *before* that rule. We have chosen the former

$$\frac{c = \mathbf{true} \text{ or } \mathbf{false} \qquad \Gamma \leq \emptyset}{\Gamma \vdash c : \mathbf{bool}} \quad (\text{T-Const}')$$

$$\frac{\Gamma \leq x : \Diamond\tau}{\Gamma \vdash x : \tau} \quad (\text{T-Var}')$$

$$\frac{[\![U]\!] \subseteq \Phi \qquad \Gamma \leq \emptyset}{\Gamma \vdash \mathbf{new}^{\Phi}() : (\mathbf{R}, U)} \quad (\text{T-New}')$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \tau_2 \qquad \alpha \text{ fresh} \\ \Gamma(f) = (\tau_1 \to \tau_2, U_1) \text{ if } f \in \mathit{dom}(\Gamma) \qquad U_1 \leq \mathbf{0} \text{ if } f \notin \mathit{dom}(\Gamma) \\ \tau_1 \leq \Gamma(x) \text{ if } x \in \mathit{dom}(\Gamma) \qquad \mathit{Use}(\tau_1) \leq \mathbf{0} \text{ if } x \notin \mathit{dom}(\Gamma) \\ \Gamma' \leq (U_2 \odot \mu\alpha.(1 \otimes (U_1 \odot \alpha))) \odot \Diamond(\Gamma \backslash \{f, x\})\end{array}}{\Gamma' \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_2)} \quad (\text{T-Fun}')$$

$$\frac{\Gamma_1 \vdash M_1 : (\tau_1 \to \tau_2, 1) \qquad \Gamma_2 \vdash M_2 : \tau_1 \qquad \Gamma \leq \Gamma_1; \Gamma_2 \qquad \tau_2 \leq \tau_2'}{\Gamma \vdash M_1\ M_2 : \tau_2'} \quad (\text{T-App}')$$

$$\frac{\Gamma \vdash M : (\mathbf{R}, U) \qquad U \leq l}{\Gamma \vdash \mathbf{acc}^{l}(M) : \mathbf{bool}} \quad (\text{T-Acc}')$$

$$\frac{\Gamma_1 \vdash M_1 : \mathbf{bool} \qquad \Gamma_2 \vdash M_2 : \tau \qquad \Gamma_3 \vdash M_3 : \tau \qquad \Gamma \leq \Gamma_1; (\Gamma_2 \,\&\, \Gamma_3) \qquad \tau \leq \tau'}{\Gamma \vdash \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : \tau'} \quad (\text{T-If}')$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2 \vdash M_2 : \tau_2 \\ \tau_1 \leq \Gamma_2(x) \text{ if } x \in \mathit{dom}(\Gamma_2) \qquad \mathit{Use}(\tau_1) \leq \mathbf{0} \text{ if } x \notin \mathit{dom}(\Gamma_2) \\ \Gamma \leq \Gamma_1; (\Gamma_2 \backslash \{x\}) \qquad \tau_2 \leq \tau_2'\end{array}}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2'} \quad (\text{T-Let}')$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \tau \\ \Gamma' \leq \blacklozenge_x \Gamma \qquad \tau \leq \tau'\end{array}}{\Gamma' \vdash M^{\{x\}} : \tau'} \quad (\text{T-Now'})$$

Fig. 10. Syntax-directed typing rules.

approach since the type reconstruction algorithm described below becomes a little clearer.

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 (the standard type for 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.

Example 6.1.2. For a term $f\ x$ where the standard type of f is $\mathbf{R} \to \mathbf{bool}$, we prepare the following types and type environments:

$$\begin{aligned}\tau_f &= ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_2) \\ \Gamma_f &= f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_3) \\ \tau_x &= (\mathbf{R}, \alpha_4) \\ \Gamma_x &= x : (\mathbf{R}, \alpha_5) \\ \tau_{f\ x} &= \mathbf{bool} \\ \Gamma_{f\ x} &= f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_6), x : (\mathbf{R}, \alpha_7).\end{aligned}$$

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, for the above term $f\ x$, the template is:

$$\frac{\begin{array}{c} f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_3) \vdash f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_2) \\ x : (\mathbf{R}, \alpha_5) \vdash x : (\mathbf{R}, \alpha_4) \end{array}}{f : ((\mathbf{R}, \alpha_1) \to \mathbf{bool}, \alpha_6), x : (\mathbf{R}, \alpha_7) \vdash f\ x : \mathbf{bool}}.$$

6.2 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. We can extract from each sub-term N the constraint $\mathcal{C}(N)$ given in Figure 11. For example, for a variable x, the syntax-directed rule (T-VAR′) requires that $\Gamma_x \le x : \diamond\tau_x$. By the construction of Γ_x in Step 1, it is guaranteed that $dom(\Gamma_x) = \{x\}$ holds and that $\Gamma_x(x)$ and τ_x are identical except for their outermost usages. We therefore generate the constraint set $\mathcal{C}(x) = \{Use(\Gamma_x(x)) \le \diamond Use(\tau_x)\}$ for the variable x.

Remark 6.2.1. The reason why we compare only the outermost usages above is that in our definition of subtyping, $\tau_1 \le \tau_2$ holds only if τ_1 and τ_2 are identical except for the outermost usages. If we introduce a more general subtyping rule (recall Remark 4.2.3), $\mathcal{C}(x)$ should be replaced with $\{\Gamma_x(x) \le \diamond\tau_x\}$. The resulting constraint set becomes a little more complex, but we can still solve the constraints in a similar manner.

Let $CS = \bigcup\{\mathcal{C}(N) \mid N \text{ is a subterm of } M\}$. 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.

Each constraint in the set CS is one of the following forms:

(1) $\alpha \le U$

(2) $[\![U]\!] \subseteq \Phi$

(3) $\tau_1 = \tau_2$, where all usages in τ_1 and τ_2 are usage variables.

(4) $\tau_1 \le \tau_2$, where all usages in τ_1 and τ_2 are usage variables.

Constraints of the third form (i.e., unification constraints) can be solved by using a standard unification algorithm. Constraints of the fourth form can be reduced to unification constraints and subusage constraints by the following rules.

$$\begin{array}{lcl} CS \cup \{\mathbf{bool} \le \mathbf{bool}\} & \Longrightarrow & CS \\ CS \cup \{(\tau_1 \to \tau_2, \alpha) \le (\tau_1' \to \tau_2', \alpha')\} & \Longrightarrow & CS \cup \{\tau_1 = \tau_1', \tau_2 = \tau_2', \alpha \le \alpha'\} \\ CS \cup \{(\mathbf{R}, \alpha) \le (\mathbf{R}, \alpha')\} & \Longrightarrow & CS \cup \{\alpha \le \alpha'\}\cdot \end{array}$$

We obtain the following set of constraints as a result:

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

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

$$
\begin{array}{l}
\mathcal{C}(c) = \{\tau_c = \mathbf{bool}\} \\
\mathcal{C}(x) = \{\mathit{Use}(\Gamma_x(x)) \leq \Diamond\,\mathit{Use}(\tau_x)\} \\
\mathcal{C}(\mathbf{new}^{\Phi}()) = \{[\![\, \mathit{Use}(\tau_{\mathbf{new}^{\Phi}()}) \,]\!] \subseteq \Phi\} \\
\mathcal{C}(\mathbf{fun}(f,x,M)) = \\
\qquad \{\Gamma_M(f) = (\tau_1 \to \tau_2, \beta) \mid f \in \mathit{dom}(\Gamma_M), \tau_{\mathbf{fun}(f,x,M)} = (\tau_1 \to \tau_2, U)\} \\
\qquad \cup\{\beta \leq \mathbf{0} \mid f \notin \mathit{dom}(\Gamma_M)\} \\
\qquad \cup\{\mathit{domty}(\tau_{\mathbf{fun}(f,x,M)}) \leq \Gamma_M(x) \mid x \in \mathit{dom}(\Gamma_M)\} \\
\qquad \cup\{\mathit{Use}(\mathit{domty}(\tau_{\mathbf{fun}(f,x,M)})) \leq \mathbf{0} \mid x \notin \mathit{dom}(\Gamma_M)\} \\
\qquad \cup\{\mathit{Use}(\Gamma_{\mathbf{fun}(f,x,M)}(y)) \\
\qquad\qquad \leq (\mathit{Use}(\tau_{\mathbf{fun}(f,x,M)}) \odot \mu\alpha.(1 \otimes (\beta \odot \alpha))) \odot \Diamond\,\mathit{Use}(\Gamma_M(y)) \\
\qquad\qquad \mid y \in \mathit{dom}(\Gamma_{\mathbf{fun}(f,x,M)})\} \\
\qquad (\beta \text{ is fresh}) \\
\mathcal{C}(M_1\ M_2) = \\
\qquad \{\mathit{Use}(\Gamma_{M_1\ M_2}(x)) \leq \mathit{Use}((\Gamma_{M_1};\Gamma_{M_2})(x)) \mid x \in \mathit{dom}(\Gamma_{M_1\ M_2})\} \\
\qquad \cup\{\mathit{Use}(\tau_{M_1}) \leq 1\} \\
\qquad \cup\{\mathit{domty}(\tau_{M_1}) = \tau_{M_2}, \mathit{codty}(\tau_{M_1}) \leq \tau_{M_1\ M_2}\} \\
\mathcal{C}(\mathbf{acc}^l(M)) = \\
\qquad \{\mathit{Use}(\tau_M) \leq l\} \\
\qquad \cup\{\Gamma_{\mathbf{acc}^l(M)}(y) = \Gamma_M(y) \mid y \in \mathit{dom}(\Gamma_M)\} \\
\mathcal{C}(\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3) = \\
\qquad \{\tau_{M_1} = \mathbf{bool}, \tau_{M_2} = \tau_{M_3}\} \\
\qquad \cup\{\mathit{Use}(\Gamma_{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3}(y)) \leq \mathit{Use}((\Gamma_{M_1};(\Gamma_{M_2} \,\&\, \Gamma_{M_3}))(y)) \\
\qquad\qquad \mid y \in \mathit{dom}(\Gamma_{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3})\} \\
\qquad \cup\{\tau_{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3} \leq \tau_{M_2}\} \\
\mathcal{C}(\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) = \\
\qquad \{\tau_{M_1} \leq \Gamma_{M_2}(x) \mid x \in \mathit{dom}(\Gamma_{M_2})\} \\
\qquad \cup\{\mathit{Use}(\tau_{M_1}) \leq \mathbf{0} \mid x \notin \mathit{dom}(\Gamma_{M_2})\} \\
\qquad \cup\{\mathit{Use}(\Gamma_{\mathbf{let}\ x=M_1\ \mathbf{in}\ M_2}(y)) \leq \mathit{Use}((\Gamma_{M_1};\Gamma_{M_2})(y)) \\
\qquad\qquad \mid y \in \mathit{dom}(\Gamma_{\mathbf{let}\ x=M_1\ \mathbf{in}\ M_2})\} \\
\qquad \cup\{\tau_{M_2} \leq \tau_{\mathbf{let}\ x=M_1\ \mathbf{in}\ M_2}\} \\
\mathcal{C}(M^{\{x\}}) = \\
\qquad \{\mathit{Use}(\Gamma_{M^{\{x\}}}(x)) \leq \blacklozenge\,\mathit{Use}(\Gamma_M(x))\} \\
\qquad \cup\{\Gamma_{M^{\{x\}}}(y) \leq \Gamma_M(y) \mid y \in \mathit{dom}(\Gamma_M)\backslash\{x\}\} \\
\qquad \cup\{\tau_M \leq \tau_{M^{\{x\}}}\}
\end{array}
$$

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

Fig. 11. Constraints extracted from each sub-term.

Example 6.2.2. From the template of a type derivation given in Example 6.1.2, we obtain the following constraints:

$$\{\alpha_3 \leq \Diamond\alpha_2, \alpha_5 \leq \Diamond\alpha_4, \alpha_6 \leq \alpha_3, \alpha_7 \leq \alpha_5, \alpha_3 \leq 1, (\mathbf{R}, \alpha_1) = (\mathbf{R}, \alpha_4), \mathbf{bool} \leq \mathbf{bool}\}.$$

By reducing the constraints, we obtain the following constraints on usages:

$$\{\alpha_3 \leq \Diamond\alpha_2, \alpha_5 \leq \Diamond\alpha_4, \alpha_6 \leq \alpha_3, \alpha_7 \leq \alpha_5, \alpha_3 \leq 1\}$$

with a substitution $[\alpha_4/\alpha_1]$.

6.3 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 following transformation rule:

$$CS \cup \{\alpha \leq U\} \Longrightarrow [\mu\alpha.U/\alpha]CS.$$

Then, we check whether the remaining set of constraints is satisfied (using the algorithm stated in the third assumption).

6.4 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, respectively) if the latter is sound (complete, respectively). 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, the 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 Kobayashi [2000a]. (If we choose the more general subtyping rule given in Remark 4.2.3, the assumption about the type size cannot be eliminated to guarantee that the algorithm runs in time $O(n^2)$.)

Although our algorithm (excluding an unspecified algorithm for checking constraints of the form $[\![U]\!] \subseteq \Phi$) requires quadratic time in the worst case, we think that the algorithm runs in linear time for ordinary programs. The size of each constraint set $\mathcal{C}(N)$ is linear in the number of free variables in N, and hence it is $O(n)$ in the worst case. For ordinary programs, however, the number of free variables in each subterm can be regarded as a constant, hence, our algorithm typically runs in linear time.

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. For example, consider the following expression:

$$\mathbf{let}\ x = \mathbf{new}^{\Phi}()\ \mathbf{in}\ (\mathbf{read}^{l_R}(x); \mathbf{write}^{l_W}(y); \mathbf{close}^{l_C}(y)).$$

Here, $\Phi = ((l_R + l_W)^* l_C \downarrow)^\sharp$. By carrying out the first and second steps of the algorithm, we obtain the following type judgment and constraints:

$$x : (\mathbf{R}, \alpha_1) \vdash \mathbf{let}\ x = \mathbf{new}^{\Phi}()\ \mathbf{in}\ (\mathbf{read}^{l_R}(x); \mathbf{write}^{l_W}(y); \mathbf{close}^{l_C}(y)) : \mathbf{bool}$$
$$\alpha_1 \leq l_R \qquad \alpha_2 \leq l_W; l_C \qquad [\![\alpha_2]\!] \subseteq \Phi.$$

Since α_2 cannot be constrained by the outside of the expression, we can solve the constraints on α_2, and obtain the following simplified type judgment and constraint:

$$\begin{array}{l} x : (\mathbf{R}, \alpha_1) \vdash \mathbf{let}\ x = \mathbf{new}^{\Phi}()\ \mathbf{in}\ (\mathbf{read}^{l_R}(x); \mathbf{write}^{l_W}(y); \mathbf{close}^{l_C}(y)) : \mathbf{bool} \\ \alpha_1 \leq l_R. \end{array}$$

6.5 Examples

We give examples of our analysis. We often omit annotations on escape information below, but assume that terms of type **bool** are appropriately annotated with escape information (as in $(\mathbf{acc}^{l_I}(r))^{\{r\}}$, $(f\ r)^{\{r\}}$). For readability, usage expressions are often replaced with equivalent but simplified ones: for example, U is substituted for $\blacklozenge\Diamond U$.

Example 6.5.1. Let us consider the program in Example 2.2. The template of type derivation for the program is of the form (unification on some usage variables has been already applied for the sake of readability):

$$\dfrac{\dfrac{\dfrac{\cdots}{f : \tau_f, x : (\mathbf{R}, \alpha_x) \vdash \mathbf{if} \cdots : \mathbf{bool}}}{\emptyset \vdash \mathbf{fun}(f, x, \mathbf{if}\ \cdots) : \tau'_f} \quad \dfrac{\emptyset \vdash \mathbf{new}^{\Phi_r}() : (\mathbf{R}, \alpha_r) \quad \dfrac{\cdots}{f : \tau'_f, r : (\mathbf{R}, \alpha_r) \vdash \mathbf{init}^{l_I}(r); f\ r : \mathbf{bool}}}{f : \tau'_f \vdash \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ (\mathbf{init}^{l_I}(r); f\ r) : \mathbf{bool}}}{\emptyset \vdash \mathbf{let}\ f = \mathbf{fun}(f, x, \mathbf{if}\ \cdots)\ \mathbf{in}\ \mathbf{let}\ r = \mathbf{new}^{\Phi_r}()\ \mathbf{in}\ (\mathbf{init}^{l_I}(r); f\ r) : \mathbf{bool}}$$

Here, $\tau_f = ((\mathbf{R}, \alpha_x) \to \mathbf{bool}, \alpha_f)$ and $\tau'_f = ((\mathbf{R}, \alpha_x) \to \mathbf{bool}, \alpha'_f)$. We get the following constraints on usage variables α_x and α_r:

$$\{\alpha_x \leq l_R; (l_F \mathbin{\&} (l_W; \alpha_x)),\ \alpha_r \leq l_I; \alpha_x,\ [\![\, \alpha_r \,]\!] \subseteq \Phi_r\}$$

The first constraint is obtained from the derivation for $f : \tau_f, x : (\mathbf{R}, \alpha_x) \vdash \mathbf{if} \cdots :$ **bool** and the second constraint is obtained from the derivation for $f : \tau'_f, r :$ $(\mathbf{R}, \alpha_r) \vdash \mathbf{init}^{l_I}(r); f\ r : \mathbf{bool}$. By solving the first two subusage constraints, we get $\alpha_r = l_I; \mu\alpha_x.(l_R; (l_F \mathbin{\&} (l_W; \alpha_x)))$. By substituting the solution for the third constraint, we get

$$[\![\, l_I; \mu\alpha_x.(l_R; (l_F \mathbin{\&} (l_W; \alpha_x))) \,]\!] = (l_I (l_R l_W)^* l_R l_F \downarrow)^\sharp \subseteq \Phi_r.$$

Since $\Phi_r = (l_I (l_R + l_W)^* l_F \downarrow)^\sharp$, we know that the program is well typed.

Example 6.5.2. Let us consider the following program:

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

The usage of r, inferred in a manner similar to the above example, is $\mu\alpha.(l_R; (\mathbf{0} \mathbin{\&} (l_{Push}; \alpha; l_{Pop})))$. 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 [Bigliardi and Laneve 2000], memory management with reference counting [Walker and Watkins 2001] (counter increment corresponds to **push** and decrement to **pop**).

Example 6.5.3. Let us consider the following program:

$$
\begin{array}{l}
\textbf{let } f = \textbf{fun}(f, g, g \textbf{ true}; f\ g) \textbf{ in} \\
\textbf{let } r = \textbf{new}^{\Phi_r}() \textbf{ in} \\
f\ (\lambda x.\textbf{read}^{l_R}(r)).
\end{array}
$$

It first creates a new resource r, and passes to f a function to access the resource. f calls the function repeatedly, forever. The template of type derivation for the program is of the form:

$$
\dfrac{
\dfrac{\dfrac{\cdots}{f : \tau_f, g : \tau_g \vdash g \textbf{ true}; f\ g : \textbf{bool}}}{\emptyset \vdash \textbf{fun}(f, g, g \textbf{ true}; f\ g) : \tau'_f}
\qquad
\dfrac{\emptyset \vdash \textbf{new}^{\Phi_r}() : (\mathbf{R}, \alpha_r) \quad \dfrac{\dfrac{\cdots}{f : \tau'_f, r : (\mathbf{R}, \alpha'_r) \vdash M : \textbf{bool}}}{f : \tau'_f, r : (\mathbf{R}, \alpha_r) \vdash M^{\{r\}} : \textbf{bool}}}{f : \tau'_f \vdash \textbf{let } r = \textbf{new}^{\Phi_r}() \textbf{ in } M^{\{r\}} : \textbf{bool}}
}{\emptyset \vdash \textbf{let } f = \textbf{fun}(f, g, g \textbf{ true}; f\ g) \textbf{ in let } r = \textbf{new}^{\Phi_r}() \textbf{ in } M^{\{r\}} : \textbf{bool}}
$$

Here, $M = f\ (\lambda x.\textbf{read}^{l_R}(r))$, $\tau_g = (\textbf{bool} \to \textbf{bool}, \alpha_g)$, $\tau_f = (\tau_g \to \textbf{bool}, \alpha_f)$, and $\tau'_f = (\tau_g \to \textbf{bool}, \alpha'_f)$. From the template, we get the following constraints on α_g and α_r:

$$\{\alpha_g \leq 1; \alpha_g,\ \alpha_r \leq \blacklozenge\alpha'_r,\ \alpha'_r \leq (\alpha_g \odot \mu\alpha.(1 \otimes (\mathbf{0} \odot \alpha))) \odot \diamond l_R,\ [\![\, \alpha_r \,]\!] \subseteq \Phi_r\}.$$

The first constraint is obtained from the derivation for $f : \tau_f, g : (\mathbf{R}, \tau_g) \vdash g$ **true**; $f\ g$: **bool** and the third constraint is obtained from the derivation of $f : \tau'_f, r : (\mathbf{R}, \alpha_r) \vdash M : \textbf{bool}$.

From the first three constraints, we get:

$$
\begin{array}{rcl}
\alpha_g & = & \mu\alpha.(1; \alpha) \\
\alpha_r & = & \blacklozenge\alpha'_r \\
 & = & \blacklozenge((\alpha_g \odot \mu\alpha.(1 \otimes (\mathbf{0} \odot \alpha))) \odot \diamond l_R) \\
 & \cong & \blacklozenge(\alpha_g \odot \diamond l_R) \\
 & \cong & \mu\alpha.(l_R; \alpha).
\end{array}
$$

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 $\downarrow$.)

6.6 Some Algorithms for Checking $[\![U]\!] \subseteq \Phi$

In the discussion above, we assumed that there exists an algorithm to verify $[\![\, U \,]\!] \subseteq \Phi$. There is obviously no complete algorithm to verify $[\![\, U \,]\!] \subseteq \Phi$ for arbitrary U and Φ, since it subsumes the inclusion problem between context-free languages. For some specific set Φ, however, there is an algorithm to verify $[\![\, U \,]\!] \subseteq \Phi$.

We informally present a sound (but incomplete[7]) algorithm for the case where $\Phi = ((l_R + l_W)^* l_C \downarrow)^\sharp$, which denotes the usage of files. Constraints of the form

[7]It is probably possible to construct a complete algorithm with a little more complication,

$[\![U]\!] \subseteq ((l_R + l_W)^* l_C \downarrow)^\sharp$ can be expanded into the following form:

$$\begin{array}{l} [\![\alpha_1]\!] \subseteq ((l_R + l_W)^* l_C \downarrow)^\sharp \\ \alpha_1 \le F_1(\alpha_1, \ldots, \alpha_n) \\ \cdots \\ \alpha_n \le F_n(\alpha_1, \ldots, \alpha_n) \end{array}$$

Here, each $F_i(\alpha_1, \ldots, \alpha_n)$ consists of usage variables $\alpha_1, \ldots, \alpha_n$, usage constants, and usage constructors other than the recursive usage constructor. Since the goal is to check whether $[\![\alpha_1]\!] \subseteq ((l_R + l_W)^* l_C \downarrow)^\sharp$, we need not obtain the exact value of α_1. So, we solve the subusage constraints over the abstract domain:

$$\{\mu\alpha.\alpha, \mathbf{0}, U_{RW}, \diamond U_{RW}, U_C, U_{Error}\}$$

where $U_{RW} = \mu\alpha.(\mathbf{0} \mathbin{\&} (l_R;\alpha) \mathbin{\&} (l_W;\alpha))$, $U_C = \diamond\mu\alpha.(l_C \mathbin{\&} (l_R;\alpha) \mathbin{\&} (l_W;\alpha))$, and $U_{Error} = \diamond\mu\alpha.(\mathbf{0} \mathbin{\&} (l_C;\alpha) \mathbin{\&} (l_R;\alpha) \mathbin{\&} (l_W;\alpha))$. By abstracting usage constants and constructors in F_i accordingly, we obtain the following abstract version of the constraints:

$$\begin{array}{l} [\![\alpha_1]\!] \subseteq ((l_R + l_W)^* l_C \downarrow)^\sharp \\ \alpha_1 \le F_1^\flat(\alpha_1, \ldots, \alpha_n) \\ \cdots \\ \alpha_n \le F_n^\flat(\alpha_1, \ldots, \alpha_n) \end{array}$$

For example, the constructor $\otimes$ is replaced by the following abstract operation:

$\otimes^\flat$	$\mu\alpha.\alpha$	$\mathbf{0}$	U_{RW}	$\diamond U_{RW}$	U_C	U_{Error}
$\mu\alpha.\alpha$	$\mu\alpha.\alpha$	$\mu\alpha.\alpha$	U_{RW}	$\diamond U_{RW}$	U_C	U_{Error}
$\mathbf{0}$	$\mu\alpha.\alpha$	$\mathbf{0}$	U_{RW}	$\diamond U_{RW}$	U_C	U_{Error}
U_{RW}	U_{RW}	U_{RW}	U_{RW}	$\diamond U_{RW}$	U_{Error}	U_{Error}
$\diamond U_{RW}$	$\diamond U_{RW}$	$\diamond U_{RW}$	$\diamond U_{RW}$	$\diamond U_{RW}$	U_{Error}	U_{Error}
U_C	U_C	U_C	U_{Error}	U_{Error}	U_{Error}	U_{Error}
U_{Error}	U_{Error}	U_{Error}	U_{Error}	U_{Error}	U_{Error}	U_{Error}

Since the abstract domain is a finite semilattice, we can solve the inequalities by using the standard method [Rehof and Mogensen 1999]. $[\![\alpha_1]\!] \subseteq ((l_R + l_W)^* l_C \downarrow)^\sharp$ holds if α_1 is $\mu\alpha.\alpha$ or U_C.

The case where $\Phi_r = (l_I(l_R + l_W)^* l_F \downarrow)^\sharp$ (recall Example 2.2) can be dealt with in a similar manner. For the trace set $\Phi = \{l_{push}^n l_{pop}^n \mid n \ge 0\}^\sharp$, which represents the usage of a stack, we think we can develop a sound algorithm (that may not be complete) to verify $[\![U]\!] \subseteq \Phi$ by modifying the algorithm given by Iwama and Kobayashi [2002].

7. EXTENSIONS

We discuss some extensions to refine our type-based usage analysis.

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

$$\mathbf{let}\ f = \lambda x.(\mathbf{acc}^{l_1}(x); x)\ \mathbf{in}\ (\mathbf{acc}^{l_2}(f\ y); \mathbf{acc}^{l_3}(f\ z)).$$

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 4.2.3).

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

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

The type inferred for x is $(\mathbf{R}, \blacklozenge(\diamond l_2; l_1))$ (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 [Igarashi and Kobayashi 2003]. For example, the type environment of the expression $\mathbf{acc}^{l_1}(x); \mathbf{acc}^{l_2}(y)$ 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: $[x/y](x : (\mathbf{R}, l_1); y : (\mathbf{R}, l_2)) = x : (\mathbf{R}, l_1; l_2)$. With the extension above, we expect that the type inference algorithm and its time complexity do not change so much, although the proof of type soundness becomes more complex.

Combination with Region/Effect Systems. Regions and effects [Birkedal et al. 1996; Tofte and Talpin 1994] are also useful to improve the accuracy of the analysis. Consider a term $(\lambda y.\mathbf{acc}^{l_1}(x))\ \mathbf{acc}^{l_2}(x)$. The best type we can assign to x is $(\mathbf{R}, \diamond l_1; l_2)$, although the term is computationally equivalent to $\mathbf{let}\ y = \mathbf{acc}^{l_2}(x)\ \mathbf{in}\ \mathbf{acc}^{l_1}(x)$. The problem is that rule (T-FUN) loses information that free variables in $\lambda x.M$ are accessed only after the function is applied.

We can better handle this problem using region and effect systems [Birkedal et al. 1996; Tofte and Talpin 1994]. 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 $\lambda y.\mathbf{acc}^{l_1}(x)$ 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.

There is, however, a drawback in region and effect systems. 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. [DeLine and Fähndrich 2001; Tofte and Talpin 1994; Walker et al. 2000], at the cost of complication of type systems.

We have recently studied a combination of our type system with regions and effects to take the best of both worlds [Kobayashi 2003]. The resulting analysis no longer requires a separate escape analysis, because region/effect information subsumes escape information. Development of a type inference algorithm for the new type system is under way.

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 [2001] type system, existential types are required to express similar information.) The rules for constructing and destructing lists can be given as:

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

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

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.

8. RELATED WORK

8.1 General Type Systems for Resource Usage Analysis

The goal of our type system is close to that of DeLine and Fähndrich's type system for programming language Vault [DeLine and Fähndrich 2001, 2002] and Foster et al.'s [2002] type system. We discuss relationship between our type system and them in this section.

Their type systems keep track of the state of a resource and ensure that only valid operations are performed on the resource in each state. For example, let us consider socket libraries. Socket libraries contain various functions to access sockets, but they should be applied in a particular order: the function *bind* should be first called, and then the function *listen* should be called, etc. To enforce such usage of sockets, the following types[8] are assigned in Vault [DeLine and Fähndrich 2001]:

$$\begin{array}{l} \mathit{socket} : \cdots \to (\mathit{sock}, \mathit{raw}) \\ \mathit{bind} : (\mathit{sock}, \mathit{raw}) \to (\mathit{sock}, \mathit{named}) \\ \mathit{listen} : (\mathit{sock}, \mathit{named}) \to (\mathit{sock}, \mathit{listening}) \\ \cdots \end{array}$$

The types specify that the function *socket* creates a new socket in state *raw*, that the function *bind* takes a socket in state *raw* and changes its state to *named*, and that the function *listen* takes a socket in state *named* and changes its state to *listening*. These types enforce that *bind* is first applied to a new socket, and

[8]The notation used here is simplified and is imprecise. For precise descriptions, see Deline and Fähndrich [2001, 2002] and Foster et al. [2002].

then *listen* is applied. In our resource usage analysis, a similar effect can be achieved by assigning to *socket* the following type:

$$socket : \cdots \to (sock, bind; listen; \cdots)$$

The usage *bind*; *listen* specifies that *bind* and *listen* should be applied in this order.

Although the difference above may not look essential, both approaches have both advantages and disadvantages. A disadvantage of our approach is that usage expressions are so expressive that there is no complete algorithm for deciding $[\![U]\!] \subseteq \Phi$ (i.e., whether the inferred usage U conforms to the specification Φ). As we discussed in Section 6.6, however, we think that for a certain class of languages for describing Φ (regular languages, in particular), we can develop an (at least sound) algorithm for checking $[\![U]\!] \subseteq \Phi$. On the other hand, our approach has the following advantages. First, the other type systems [DeLine and Fähndrich 2001, 2002; Foster et al. 2002] cannot deal with resources that can have infinite states (like stacks), but our type system can, as long as the number of operations is finite. For example, the state of a stack can be expressed using a sequence of actions *push* and *pop* in our approach. Second, our approach seems to require less complex type machinery. To see the advantage, consider a resource to which an operation f can be applied at most twice. Then, the resource can be modeled as an automaton with three states q_0, q_1, q_2 such that f can be applied in q_0 and q_1 and the state becomes q_1 and q_2 respectively. Since f can be applied in two states, the following intersection type has to be assigned in the approach of extending types with states of resources:

$$f : ((\mathbf{R}, q_0) \to (\mathbf{R}, q_1)) \wedge ((\mathbf{R}, q_1) \to (\mathbf{R}, q_2)).$$

On the other hand, we just need to assign type $(\mathbf{R}, f; f)$ to a new resource. Third, our approach can easily deal with concurrent access to resources. Let $M_1 \| M_2$ be an expression that evaluates M_1 and M_2 in parallel and returns the result of M_2. Then, the typing rule for the expression is given as:

$$\frac{\Gamma_1 \vdash M_1 : \mathbf{bool} \quad \Gamma_2 \vdash M_2 : \tau_2}{\Gamma_1 \otimes \Gamma_2 \vdash M_1 \| M_2 : \tau_2}$$

Notice here that $\otimes$ is used to combine environments instead of ";". For example, the following type derivation expresses that the file x is read twice.

$$\frac{\begin{array}{c} x : (\mathbf{File}, l_R) \vdash \mathbf{fread}^{l_R}(x) : \mathbf{bool} \\ x : (\mathbf{File}, l_R) \vdash \mathbf{fread}^{l_R}(x) : \mathbf{bool} \end{array}}{x : (\mathbf{File}, l_R \otimes l_R) \vdash \mathbf{fread}^{l_R}(x) \| \mathbf{fread}^{l_R}(x) : \mathbf{bool}}.$$

On the other hand, it is not obvious how to extend the type systems in Foster et al. [2002] and Deline and Fähndrich [2001, 2002] to deal with concurrency. For example, if one wants to check that a file is read exactly twice (as in the example above), the file must be given three states: *not_read*, *read_once*, *read_twice*. Then, in $\mathbf{fread}^{l_R}(x); \mathbf{fread}^{l_R}(x)$, the first $\mathbf{fread}^{l_R}(x)$ would be given a typing which expresses that the file state is changed from *not_read* into *read_once*, and the second one would be given a typing which expresses that the state is changed from *read_once* into *read_twice*. In the case of $\mathbf{fread}^{l_R}(x) \|$

$\mathbf{fread}^{l_R}(x)$, however, $\mathbf{fread}^{l_R}(x)$ cannot be typed since we cannot statically tell which $\mathbf{fread}^{l_R}(x)$ is executed first.

Another technical difference between our type system and their type systems [DeLine and Fähndrich 2001, 2002; Foster et al. 2002] is that our type system uses the ideas of linear types, while their type systems use the ideas of region/effect systems. (The type system of Vault [DeLine and Fähndrich 2002] also uses some idea of linear types, but in a way different from ours.) As we sketched in Section 7, both approaches have advantages and disadvantages: In the region/effect-based approach, the analysis becomes imprecise when multiple resources are represented by the same region. The type system of Vault, therefore, uses complex type machinery such as bounded polymorphism and existential types, so that automatic type inference is not possible. Foster et al.'s [2002] type system basically gives up keeping track of the state of resources that may be aliased to the same region and introduces a special programming construct to partially solve the problem. On the other hand, our analysis based on linear types becomes imprecise when resources are put into a closure (as mentioned in Section 7). We have recently extended our type system with ideas of region/effect systems, but a type inference algorithm for the new type system has not been developed yet [Kobayashi 2003].

The type system of Vault [DeLine and Fähndrich 2001, 2002] requires explicit type annotation, while in our type system and Foster et al.'s [2002] type system, types can be inferred automatically. Annotation of trace sets (Φ) is necessary in our framework, but it is only used to declare valid access sequences. It 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:

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

Here, we assume that the primitives for reading, writing, and closing a file are annotated with l_R, l_W, and l_C, respectively. As for the analysis cost, except for the unspecified algorithm for checking constraints of the form $[\![U]\!] \subseteq \Phi$, the time complexity of our analysis seems comparable to that of Foster et al.'s [2002] analysis the worst-case time complexity of both analyses is $O(n^2)$.

Vault's type system can check dependencies between multiple resources (e.g., the property that a certain set of resources are guarded by a lock), while our present type system cannot. The type system of Vault and Foster et al.'s [2002] type system can deal with pointers, while the target language of our analysis is a purely functional language. We expect that our analysis can be extended to deal with these points by using techniques we developed elsewhere [Igarashi and Kobayashi 2003].

8.2 Other Related Type Systems

Technical ideas of our type-based analysis are similar to the quasi-linear type system [Kobayashi 1999] for memory management and type systems for concurrent processes (especially, those for deadlock-free processes) [Igarashi and Kobayashi 2003; Kobayashi 2000b; Kobayashi et al. 2000; Sumii and Kobayashi 1998]. 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 [Igarashi and Kobayashi 2003; Kobayashi 2000b; Kobayashi et al. 2000; Sumii and Kobayashi 1998]. In those type systems, usage expressions express how each communication channel is used.

The problem of linearity analysis [Gustavsson and Svenningsson 2000; Turner et al. 1995; Wadler 1990; Wansbrough and Peyton Jones 1999] 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 Igarashi and Kobayashi [2000a].

Among previous work on region-based memory management, most closely related would be Walker et al. [2000, 2001] and Grossman et al. [2002]. Given programs explicitly annotated with region operations, their type systems check the safety of the region operations through a type system. (On the other hand, most of other work on region-based memory management [Aiken et al. 1995; Birkedal et al. 1996; Tofte and Talpin 1994] 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 systems.

Freund and Mitchell [1999] proposed a type system for Java bytecode that 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. The successor of our type system [Kobayashi 2003] can, however, deal with that problem.

Flanagan and Abadi [1999a, 1999b] studied a type system that ensures that a certain lock is acquired before a shared resource is accessed. Our present type system cannot be used for that purpose since our type system cannot keep track of dependencies between multiple resources. As we mentioned above, we expect that our type system can be extended to analyze ordering between accesses to different resources by introducing techniques developed for type systems for the π-calculus [Igarashi and Kobayashi 2003].

8.3 Other Methods for Resource Usage Analysis

There are other approaches to verification of similar properties of programs, using dataflow analysis [Das et al. 2002], model checking, and theorem provers [Ball and Rajamani 2002; Henzinger et al. 2002; Flanagan et al. 2002]. One advantage of type-based approaches in general seem to be that modular analyses can be performed using standard techniques for type inference and that there is a standard, syntactic technique for proving soundness (using the subject reduction property). Besides this general difference, Das et al.'s [2002] method seems to suffer from a problem similar to that of the region/effect-based approach explained in Section 7; when different resources may flow into the same argument of a resource access primitive, their analysis seems unable to determine whether the primitive is indeed applied to the resources. On the other hand, their approach has an advantage that it can deal with value-dependent behavior, unlike the type-based approaches [DeLine and Fähndrich 2002; Foster et al. 2002] including ours. For example, consider the following program fragment:

```
if(d){lock(l);}
...
if(d){unlock(l);}.
```

Their analysis [Das et al. 2002] can check that lock primitives are correctly used, while the type-based approaches including ours cannot.

9. 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 the development of general methods for resource usage analysis, we have also presented a type-based method.

Much 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 7. In fact, our current type-based method does not subsume many solutions proposed for individual problems [Freund and Mitchell 1999; Walker et al. 2000]. It is also left for future work to choose a language appropriate to specify valid trace sets (Φ), and design a practical algorithm to check that inferred usages conform to the specification (i.e., $[\![U]\!] \subseteq \Phi$).

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 concurrent languages. A rather different method may be necessary to analyze those languages.

APPENDIX

A. PROPERTIES OF THE SUBUSAGE RELATION

LEMMA A.1. *The relation $\leq$ satisfies the following propositions:*

(1) *$\leq$ is reflexive and transitive,*
(2) *if $U_1 \preceq U_2$, then $U_1 \leq U_2$,*

(3) $\mathbf{0} \cong \mathbf{0} \odot U$,
(4) $U \cong \mathbf{0}\,; U \cong U\,; \mathbf{0} \cong U \otimes \mathbf{0} \cong \mathbf{0} \otimes U$,
(5) $U \cong l \odot U$,
(6) $U_1 \otimes U_2 \cong U_2 \otimes U_1$,
(7) $(U_1 \otimes U_2) \otimes U_3 \cong U_1 \otimes (U_2 \otimes U_3)$,
(8) $U_1 \otimes U_2 \leq U_1\,; U_2$,
(9) *if* $U_1{}^{\Downarrow}$, *then* $U_1\,; U_2 \leq U_1 \otimes U_2$,
(10) *if* $U_1{}^{\Downarrow}$ *and* $U_2{}^{\Downarrow}$, *then* $U_1\,; U_2 \cong U_1 \otimes U_2$,
(11) $(U_1\,; U_2) \otimes (U_3\,; U_4) \leq (U_1 \otimes U_3)\,; (U_2 \otimes U_4)$,
(12) $(U_1 \otimes U_3)\,\&\,(U_2 \otimes U_3) \cong (U_1\,\&\,U_2) \otimes U_3$,
(13) $(U_1 \odot U_2) \odot U_3 \cong U_1 \odot (U_2 \odot U_3)$,
(14) $\diamond(U_1 \odot U_2) \cong U_1 \odot \diamond U_2$,
(15) $U_1 \odot U_2 \cong \diamond U_1 \odot U_2$,
(16) $(U_1 \odot U') \otimes (U_2 \odot U') \cong (U_1 \otimes U_2) \odot U'$,
(17) $(U' \odot U_1) \otimes (U' \odot U_2) \cong U' \odot (U_1 \otimes U_2)$,
(18) $(U_1\,; U_2) \odot U' \leq (U_1 \odot U') \otimes (U_2 \odot U')$,
(19) $\diamond\diamond U \cong \diamond U \leq U$,
(20) $\diamond(U_1 \otimes U_2) \cong \diamond U_1 \otimes \diamond U_2$,
(21) $\diamond(U_1 \odot U_2) \leq \diamond U_1 \odot \diamond U_2$,
(22) $U \leq \blacklozenge U$,
(23) $\blacklozenge U_1 \otimes \blacklozenge U_2 \leq \blacklozenge(U_1 \otimes U_2)$,
(24) *If* $U \leq C[U]$, *then* $U \leq \mu\alpha.C[\alpha]$,
(25) $[\![U]\!] = [\![\diamond U]\!]$, *and*
(26) *If* $U_1 \leq U_2$, *then* $[\![U_2]\!] \subseteq [\![U_1]\!]$.

Proof. (1), (2), (25), and (26) immediately follow from definitions. Proofs of (3)–(24) are similar to each other. We give only a proof of $U \leq \mathbf{0}\,; U$ (a part of (4)) below. Let $\mathcal{S}$ be the following binary relation on usages:

$$\{(C[U_1, \ldots, U_n], C[\mathbf{0}\,; U_1, \ldots, \mathbf{0}\,; U_n]) \\ \mid U_1, \ldots, U_n \in \mathcal{U}, C \text{ is a usage context with } n \text{ holes}\}.$$

The required property $U \leq \mathbf{0}\,; U$ follows if we show $\mathcal{S} \subseteq \leq$. Therefore, it suffices to show that any $(U_1, U_2) \in \mathcal{S}$ satisfies the following three conditions:

(1) $(C[U_1], C[U_2]) \in \mathcal{S}$ for any usage context C;
(2) If $U_2 \xrightarrow{l} U_2'$, then $U_1 \xrightarrow{l} U_1'$ and $(U_1', U_2') \in \mathcal{S}$ for some U_1'.
(3) If $U_2{}^{\downarrow}$, then $U_1{}^{\downarrow}$.

The first and third conditions are trivial. We show the second condition by induction on derivation of $U_2 \xrightarrow{l} U_2'$, with case analysis on the last rule used.

—*Case* (UR-Zero). This case cannot happen.

—*Case* (UR-ParL). In this case, it must be the case that:

$$\begin{aligned} &U_1 = C_1[V_1, \ldots, V_m] \otimes C_2[V_{m+1}, \ldots, V_n] \\ &U_2 = C_1[\mathbf{0} ; V_1, \ldots, \mathbf{0} ; V_m] \otimes C_2[\mathbf{0} ; V_{m+1}, \ldots, \mathbf{0} ; V_n] \\ &C_1[\mathbf{0} ; V_1, \ldots, \mathbf{0} ; V_m] \xrightarrow{l} U'_{21} \\ &U'_2 = U'_{21} \otimes C_2[\mathbf{0} ; V_{m+1}, \ldots, \mathbf{0} ; V_n]. \end{aligned}$$

By the induction hypothesis, there exists U'_{11} such that $C_1[V_1, \ldots, V_m] \xrightarrow{l} U'_{11}$ and $(U'_{11}, U'_{21}) \in \mathcal{S}$. So, $U'_1 = U'_{11} \otimes C_2[\mathbf{0};V_{m+1}, \ldots, \mathbf{0};V_n]$ satisfies the required condition.

—*Case* (UR-ParR). Similar to the case for (UR-ParL).

—*Case* (UR-SeqL). Similar to the case for (UR-ParL).

—*Case* (UR-SeqR). In this case, either of the following conditions holds:

(1) $U_2 = \mathbf{0} ; U_1$, $U_1 \xrightarrow{l} U'$, and $U'_2 = \mathbf{0} ; U'$.

(2) $U_2 = C_1[\mathbf{0};V_1, \ldots, \mathbf{0};V_m];C_2[\mathbf{0};V_{m+1}, \ldots, \mathbf{0};V_n]$, $C_2[\mathbf{0};V_{m+1}, \ldots, \mathbf{0};V_n] \xrightarrow{l} U'_{22}$, and $U'_2 = C_1[\mathbf{0} ; V_1, \ldots, \mathbf{0} ; V_m] ; U'_{22}$.

In the first case, $U'_1 = U'$ satisfies the required condition. A proof for the second case is similar to the case for (UR-ParL).

—*Case* (UR-Box). In this case, it must be the case that:

$$\begin{aligned} &U_1 = \diamond C[V_1, \ldots, V_m] \\ &U_2 = \diamond C[\mathbf{0} ; V_1, \ldots, \mathbf{0} ; V_m] \\ &C[\mathbf{0} ; V_1, \ldots, \mathbf{0} ; V_m] \xrightarrow{l} U''_2 \\ &U'_2 = \diamond U''_2 \end{aligned}$$

By induction hypothesis, there exists U''_1 such that $C[V_1, \ldots, V_m] \xrightarrow{l} U''_1$ and $(U''_1, U''_2) \in \mathcal{S}$. So, $U'_1 = \diamond U''_1$ satisfies the required condition.

—*Case* (UR-Unbox). Similar to the case for (UR-Box).

—*Case* (UR-Mult). Similar to the case for (UR-ParL).

—*Case* (UR-SMult). Similar to the case for (UR-ParL).

—*Case* (UR-PCong). In this case,

$$U_2 = C[\mathbf{0} ; V_1, \ldots, \mathbf{0} ; V_n] \preceq C'[\mathbf{0} ; V'_1, \ldots \mathbf{0} ; V'_n] \xrightarrow{l} U'_2,$$

with $C \preceq C'$ and $V_1 \preceq V'_1, \ldots, V_n \preceq V'_n$. Since $U_1 = C[V_1, \ldots, V_n] \preceq C'[V'_1, \ldots, V'_n]$, the induction hypothesis implies that there exists U'_1 such that

$$U_1 \preceq C'[V'_1, \ldots, V'_n] \xrightarrow{l} U'_1$$

and $(U'_1, U'_2) \in \mathcal{S}$. □

Lemma A.2

(1) *If* $U_4{}^{\Downarrow}$, *then* $((U_1 ; U_2) \odot U_3) \odot U_4 \leq ((U_1 \odot U_3) \odot U_4) ; ((U_2 \odot U_3) \odot U_4)$, *and*

(2) $((U_1 \odot \diamond U_2) \odot U_3) \odot \diamond U_4 \leq U_1 \odot \diamond((U_2 \odot U_3) \odot \diamond U_4)$, *and*

(3) *if* $U_1 \leq \mathbf{1}$, *then* $(U_1 \odot (\mu\alpha.\mathbf{1} \otimes (U_2 \odot \alpha))) \odot U_3 \leq U_3 \otimes ((U_2 \odot (\mu\alpha.\mathbf{1} \otimes (U_2 \odot \alpha))) \odot U_3)$.

Proof

(1) By the following calculation:

$$\begin{array}{ll} ((U_1\,;U_2)\odot U_3)\odot U_4 & \\ \quad\le\ ((U_1\odot U_3)\otimes(U_2\odot U_3))\odot U_4 & \text{(Lemma A.1(18))}\\ \quad\le\ ((U_1\odot U_3)\,;(U_2\odot U_3))\odot U_4 & \text{(Lemma A.1(8))}\\ \quad\le\ ((U_1\odot U_3)\odot U_4)\otimes((U_2\odot U_3)\odot U_4) & \text{(Lemma A.1(18))}\\ \quad\le\ ((U_1\odot U_3)\odot U_4)\,;((U_2\odot U_3)\odot U_4) & \text{(Lemma A.1(10))} \end{array}$$

(2) By the following calculation:

$$\begin{array}{ll} ((U_1\odot\diamond U_2)\odot U_3)\odot\diamond U_4 & \\ \quad\le\ ((U_1\odot U_2)\odot U_3)\odot\diamond U_4 & \text{(Lemma A.1(15) and (14))}\\ \quad\le\ U_1\odot((U_2\odot U_3)\odot\diamond U_4) & \text{(Lemma A.1(13))}\\ \quad\le\ U_1\odot\diamond((U_2\odot U_3)\odot\diamond U_4) & \text{(Lemma A.1(19) and (15))} \end{array}$$

(3) By the following calculation:

$$\begin{array}{ll} (U_1\odot(\mu\alpha.1\otimes(U_2\odot\alpha)))\odot U_3 & \\ \quad\le\ (\mu\alpha.1\otimes(U_2\odot\alpha))\odot U_3 & (U_1\le l \text{ and Lemma A.1(5))}\\ \quad\le\ (1\otimes(U_2\odot\mu\alpha.1\otimes(U_2\odot\alpha)))\odot U_3 & \text{(Lemma A.1(2))}\\ \quad\le\ U_3\otimes((U_2\odot(\mu\alpha.1\otimes(U_2\odot\alpha)))\odot U_3) & \text{(Lemma A.1(18) and (5))} \end{array}$$ □

B. PROOF OF THEOREM 5.4.2

Lemma B.1 (Inversion)

(1) *If* $\Gamma\vdash x:\tau$, *then* $\Gamma\le x:\diamond\tau$.

(2) *If* $\Gamma\vdash \mathbf{true}:\tau$ *or* $\Gamma\vdash\mathbf{false}:\tau$, *then* $\Gamma\le\emptyset$ *and* $\tau=\mathbf{bool}$.

(3) *If* $\Gamma\vdash \mathbf{let_R}\,x:U\ \mathbf{in}\ D:\tau$, *then there exist* Γ' *and* τ' *such that* $\Gamma',x:(\mathbf{R},U)\vdash D:\tau'$ *with* $\Gamma\le\Gamma'$ *and* $\tau'\le\tau$.

(4) *If* $\Gamma\vdash\mathbf{fun}(f,x,M):\tau$, *then there exist* $\alpha,U_1,U_2,\tau_1,\tau_2$, *and* Γ_1 *such that* $\Gamma_1,f:(\tau_1\to\tau_2,U_1),x:\tau_1\vdash M:\tau_2$ *and* $\Gamma\le(U_2\odot(\mu\alpha.(1\otimes U_1\odot\alpha)))\odot\diamond\Gamma_1$ *and* $(\tau_1\to\tau_2,U_2)\le\tau$.

(5) *If* $\Gamma\vdash D_1\,D_2:\tau$, *then there exist* Γ_1,Γ_2,τ_1, *and* τ_2 *such that* $\Gamma_1\vdash D_1:(\tau_1\to\tau_2,1)$ *and* $\Gamma_2\vdash D_2:\tau_1$ *and* $\Gamma\le\Gamma_1;\Gamma_2$ *and* $\tau_2\le\tau$.

(6) *If* $\Gamma\vdash\mathbf{if}\ D_1\ \mathbf{then}\ D_2\ \mathbf{else}\ D_3:\tau$, *then there exist* $\Gamma_1,\Gamma_2,\Gamma_3$, *and* τ_1 *such that* $\Gamma_1\vdash D_1:\mathbf{bool}$ *and* $\Gamma_2\vdash D_2:\tau_1$ *and* $\Gamma_3\vdash D_3:\tau_1$ *and* $\Gamma\le\Gamma_1;(\Gamma_2\,\&\,\Gamma_3)$ *and* $\tau_1\le\tau$.

(7) *If* $\Gamma\vdash\mathbf{new}^{\Phi}():\tau$, *then* $\Gamma\le\emptyset$ *and there exists* U *such that* $[\![\,U\,]\!]\subseteq\Phi$ *and* $(\mathbf{R},U)\le\tau$.

(8) *If* $\Gamma\vdash\mathbf{acc}^l(D):\tau$, *then* $\tau=\mathbf{bool}$ *and there exists* Γ_1 *such that* $\Gamma_1\vdash D:(\mathbf{R},l)$ *and* $\Gamma\le\Gamma_1$.

(9) *If* $\Gamma\vdash\mathbf{let}\ x=D_1\ \mathbf{in}\ D_2:\tau$, *then there exist* Γ_1,Γ_2,τ_1, *and* τ_2 *such that* $\Gamma_1\vdash D_1:\tau_1$ *and* $\Gamma_2,x:\tau_1\vdash D_2:\tau_2$ *and* $\Gamma\le\Gamma_1;\Gamma_2$ *and* $\tau_2\le\tau$.

(10) *If* $\Gamma\vdash D^{\{x\}}:\tau$, *then there exist* Γ_1,τ_1,τ_2 *such that* $\Gamma_1,x:\tau_1\vdash D:\tau_2$ *and* $\Gamma\le\Gamma_1,x:\blacklozenge\tau_1$ *and* $\tau_2\le\tau$.

Proof. Immediate from the fact that a type derivation of $\Gamma\vdash D:\tau$ must end with an application of the rule corresponding to the form of D, followed by zero or more applications of rule T-Sub. □

LEMMA B.2

(1) *If* $\Gamma \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_1 ; U_2)$*, then there exist* Γ_1 *and* Γ_2 *such that* $\Gamma_i \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_i')$ *and* $U_i \leq U_i'$ *for* $i = 1, 2$ *and* $\Gamma \leq \Gamma_1; \Gamma_2$.

(2) *Similarly, if* $\Gamma \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_1 \odot \diamond U_2)$*, then there exists* Γ' *such that* $\Gamma' \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_2')$ *and* $U_2 \leq U_2'$ *and* $\Gamma \leq U_1 \odot \diamond\Gamma'$.

PROOF

(1) By Lemma B.1, $\Gamma \leq (U' \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma'$ and $\Gamma', f : (\tau_1' \to \tau_2', U_f), x : \tau_1' \vdash M : \tau_2'$ and $(\tau_1' \to \tau_2', U') \leq (\tau_1 \to \tau_2, U_1 ; U_2)$. By rules T-FUN and T-SUB, for $i = 1, 2$,
$$(U_i \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma' \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_i).$$
Finally,
$$\Gamma \leq ((U_1 \odot \mu\alpha.(1 \otimes (U_f \odot \alpha)) \odot \diamond\Gamma'); ((U_2 \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma')$$
by Lemma A.2(1).

(2) By Lemma B.1, $\Gamma \leq (U' \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma'$ and $\Gamma', f : (\tau_1' \to \tau_2', U_f), x : \tau_1' \vdash M : \tau_2'$ and $(\tau_1' \to \tau_2', U') \leq (\tau_1 \to \tau_2, U_1 \odot \diamond U_2)$. By rules T-FUN and T-SUB,
$$(U_2 \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma' \vdash \mathbf{fun}(f, x, M) : (\tau_1 \to \tau_2, U_2).$$
Finally,
$$\begin{aligned}\Gamma &\leq ((U_1 \odot \diamond U_2) \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma' \\ &\leq U_1 \odot \diamond((U_2 \odot \mu\alpha.(1 \otimes (U_f \odot \alpha))) \odot \diamond\Gamma')\end{aligned}$$
by Lemma A.2(2). □

LEMMA B.3 (SUBSTITUTION). *If* $\Gamma_1, x : \tau_1 \vdash M : \tau_2$ *and* $\Gamma_2 \vdash v : \tau_1$*, then* $\Gamma_1 \otimes \Gamma_2 \vdash [v/x]M : \tau_2$.

PROOF. We first prove the case where v is **true**, **false**, or $\mathbf{fun}(f, y, M')$ by structural induction on the derivation of $\Gamma_1, x : \tau_1 \vdash M : \tau_2$ with a case analysis on the last rule used. We show a few representative cases below.

Case T-VAR. If $M = x$, then $\Gamma_1 = \emptyset$ and $\tau_1 = \diamond\tau_2$. Since $\diamond\tau_2 \leq \tau_2$ and $\Gamma_2 = \emptyset \otimes \Gamma_2$, by rule T-SUB, $\emptyset \otimes \Gamma_2 \vdash v : \tau_2$. The other case where $M = y \neq x$ is also easy.

Case T-FUN.
$$\begin{aligned}&M = \mathbf{fun}(f, y, M_0)\\ &U = U_2 \odot \mu\alpha.(1 \otimes (U_1 \odot \alpha))\\ &\Gamma_1, x : \tau_1 = U \odot \diamond\Gamma_1', x : U \odot \diamond\tau_1'\\ &\Gamma_1', x : \tau_1', f : (\tau_{21} \to \tau_{22}, U_1), y : \tau_{21} \vdash M_0 : \tau_{22}\\ &\tau_2 = (\tau_{21} \to \tau_{22}, U_2)\end{aligned}$$

By Lemma B.2(2), there exists Γ_2' such that $\Gamma_2' \vdash v : \tau_1'$ and $\tau_1 \leq U \odot \diamond\tau_1'$ and $\Gamma_2 \leq U \odot \diamond\Gamma_2'$. Thus, by the induction hypothesis, we have
$$\Gamma_1' \otimes \Gamma_2', f : (\tau_{21} \to \tau_{22}, U_1), y : \tau_{21} \vdash [v/x]M_0 : \tau_{22}.$$
By rule T-FUN,
$$U \odot \diamond(\Gamma_1' \otimes \Gamma_2') \vdash \mathbf{fun}(f, y, [v/x]M_0) : (\tau_{21} \to \tau_{22}, U_2).$$

By Lemma A.1(17) and (20),

$$(U \odot \diamond\Gamma_1') \otimes (U \odot \diamond\Gamma_2') \leq U \odot \diamond(\Gamma_1' \otimes \Gamma_2')$$

and T-SUB finishes the case.

Case T-APP. $M = M_1\ M_2 \qquad \Gamma_1, x : \tau_1 = \Gamma_{11}; \Gamma_{12}$
$\Gamma_{11} \vdash M_1 : (\tau_{11} \to \tau_2, 1) \qquad \Gamma_{12} \vdash M_2 : \tau_{11}$

Without loss of generality, we can assume $x \in dom(\Gamma_{11}) \cap dom(\Gamma_{12})$ and $\tau_1 = \tau_{11}; \tau_{12}$. By Lemma B.2(1), we have Γ_{21} and Γ_{22} such that

$$\Gamma_{21} \vdash v : \tau_{11} \qquad \Gamma_{22} \vdash v : \tau_{12} \qquad \Gamma_2 \leq \Gamma_{21}; \Gamma_{22}$$

Then, by the induction hypothesis, we have

$$\Gamma_{11} \otimes \Gamma_{21} \vdash [v/x]M_1 : (\tau_{11} \to \tau_2, 1) \qquad \Gamma_{12} \otimes \Gamma_{22} \vdash [v/x]M_2 : \tau_{11}$$

and then by rule T-APP,

$$(\Gamma_{11} \otimes \Gamma_{21}); (\Gamma_{12} \otimes \Gamma_{22}) \vdash [v/x](M_1\ M_2) : \tau_2.$$

Finally, by Lemma A.1(11),

$$(\Gamma_{11}; \Gamma_{12}) \otimes (\Gamma_{21}; \Gamma_{22}) \leq (\Gamma_{11} \otimes \Gamma_{21}); (\Gamma_{12} \otimes \Gamma_{22})$$

and rule T-SUB finishes the case.

Case T-NOW. $M = M_0^{\{y\}} \qquad \Gamma = \blacklozenge_y \Gamma_0 \qquad \Gamma_0 \vdash M_0 : \tau$

Easy. Note that it cannot be the case that $x = y$, since $\blacklozenge_y \Gamma_0$ is well defined and τ_1 should not be $(\mathbf{R}, U)$.

The case where v is a variable, we need to prove a slightly stronger statement. In particular, straightforward induction fails since $\Gamma \vdash x : \tau_1; \tau_2$ does *not* imply the existence of Γ_1 and Γ_2 such that $\Gamma_i \vdash x : \tau_i$ for $i = 1, 2$ and $\Gamma \leq \Gamma_1; \Gamma_2$. Thus, we prove that, if $\Gamma, x : \tau_x \vdash M : \tau$, then $\Gamma \otimes y : \tau_x \vdash [y/x]M : \tau$, by structural induction on the derivation of $\Gamma_1, x : \tau_1 \vdash M : \tau_2$. The proof itself is straightforward. Then, by Lemmas B.1 and A.1(19), $\Gamma_2 \leq y : \diamond\tau_1 \leq y : \tau_1$. Finally, use T-SUB. □

LEMMA B.4. *If $\Gamma_0 \vdash D : \tau_0$ and $\Gamma_0 \vdash D' : \tau_0$, then $\Gamma \vdash \mathcal{E}_D[D] : \tau$ iff $\Gamma \vdash \mathcal{E}_D[D'] : \tau$.*

PROOF. By structural induction on $\mathcal{E}_D$. □

PROOF OF THEOREM 5.4.2. By a case analysis on the reduction rule used. We show a few representative cases below.

Case RD-APPPUSH. $D = \mathcal{E}_D[\mathbf{let_R}\ x : U_x\ \mathbf{in}\ D_1\ D_2]$
$D' = \mathcal{E}_D[(\mathbf{let_R}\ x : U_{x1}\ \mathbf{in}\ D_1)\ (\mathbf{let_R}\ x : U_{x2}\ \mathbf{in}\ D_2)]$
$\xi = x$.

By Lemma B.4, it suffices to show that if $\Gamma_0 \vdash \mathbf{let_R}\ x : U_x\ \mathbf{in}\ D_1\ D_2 : \tau_0$ then there exist U_{x1}' and U_{x2}' such that $\Gamma_0 \vdash (\mathbf{let_R}\ x : U_{x1}'\ \mathbf{in}\ D_1)\ (\mathbf{let_R}\ x : U_{x2}'\ \mathbf{in}\ D_2) : \tau_0$ and $U_x \leq U_{x1}'; U_{x2}'$.

By Lemma B.1,

$$\Gamma_0', x : (\mathbf{R}, U_x) \vdash D_1\ D_2 : \tau_0' \qquad \Gamma_0 \leq \Gamma_0' \qquad \tau_0' \leq \tau_0$$

and

$$\Gamma_1, x : (\mathbf{R}, U_{11}) \vdash D_1 : (\tau_{22} \to \tau_0', 1) \qquad \Gamma_2, x : (\mathbf{R}, U_{12}) \vdash D_2 : \tau_{22}$$
$$\Gamma_0' \leq \Gamma_1 ; \Gamma_2 \qquad U_x \leq U_{11} ; U_{12} \qquad \tau_0' \leq \tau_0.$$

By rule T-LETRES,

$$\Gamma_1 \vdash \mathbf{let_R}\ x : U_{11}\ \mathbf{in}\ D_1 : (\tau_{22} \to \tau_0', 1)$$

and

$$\Gamma_2 \vdash \mathbf{let_R}\ x : U_{22}\ \mathbf{in}\ D_2 : \tau_{22}.$$

Rules T-APP and T-SUB finish the case.

Case RD-APP. $D = \mathcal{E}_D[(\mathbf{let_R}\ \tilde{x_1} : \tilde{U_1}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_2} : \tilde{U_2}\ \mathbf{in}\ \mathbf{fun}(f, x, M))$ $(\mathbf{let_R}\ \tilde{x_1} : \tilde{U_4}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_3} : \tilde{U_3}\ \mathbf{in}\ v)]$
$D' = \mathcal{E}_D[\mathbf{let_R}\ \tilde{x_1} : (\tilde{U_1} ; \tilde{U_4})\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_2} : \tilde{U_2}\ \mathbf{in}$ $\mathbf{let_R}\ \tilde{x_3} : \tilde{U_3}\ \mathbf{in}\ [v/x, \mathbf{fun}(f, x, M)/f]M]$.

By Lemma B.4, it suffices to show that

$$\Gamma_0 \vdash (\mathbf{let_R}\ \tilde{x_1} : \tilde{U_1}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_2} : \tilde{U_2}\ \mathbf{in}\ \mathbf{fun}(f, x, M))$$
$$(\mathbf{let_R}\ \tilde{x_1} : \tilde{U_4}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_3} : \tilde{U_3}\ \mathbf{in}\ v) : \tau_0$$

implies

$$\Gamma_0 \vdash \mathbf{let_R}\ \tilde{x_1} : (\tilde{U_1} ; \tilde{U_4})\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_2} : \tilde{U_2}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_3} : \tilde{U_3}\ \mathbf{in}$$
$$[v/x, \mathbf{fun}(f, x, M)/f]M : \tau_0.$$

By Lemma B.1,

$$\Gamma_1 \vdash \mathbf{let_R}\ \tilde{x_1} : \tilde{U_1}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_2} : \tilde{U_2}\ \mathbf{in}\ \mathbf{fun}(f, x, M) : (\tau_2 \to \tau_0', 1) \tag{1}$$
$$\Gamma_2 \vdash \mathbf{let_R}\ \tilde{x_1} : \tilde{U_4}\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x_3} : \tilde{U_3}\ \mathbf{in}\ v : \tau_2$$
$$\Gamma_0 \leq \Gamma_1 ; \Gamma_2 \qquad \tau_0' \leq \tau_0$$

and, furthermore, by repeating Lemma B.1 it is easy to show that there exist Γ_1' such that

$$\Gamma_1' \vdash \mathbf{fun}(f, x, M) : \tau_f$$
$$\Gamma_1, \tilde{x_1} : (\mathbf{R}, \tilde{U_1}), \tilde{x_2} : (\mathbf{R}, \tilde{U_2}) \leq \Gamma_1'$$
$$\tau_f \leq (\tau_2 \to \tau_0', 1). \tag{2}$$

Similarly,

$$\Gamma_2' \vdash v : \tau_2'' \tag{3}$$
$$\Gamma_2, \tilde{x_1} : (\mathbf{R}, \tilde{U_4}), \tilde{x_3} : (\mathbf{R}, \tilde{U_3}) \leq \Gamma_2'$$
$$\tau_2'' \leq \tau_2. \tag{4}$$

By (1) and Lemma B.1, there exist $\alpha, U_{f1}, U_{f2}, \tau_{f1}, \tau_{f2}$, and Γ_1'' such that

$$\Gamma_1'', f : (\tau_{f1} \to \tau_{f2}, U_{f1}), x : \tau_{f1} \vdash M : \tau_{f2} \tag{5}$$
$$\Gamma_1' \leq (U_{f2} \odot \mu\alpha.(1 \otimes (U_{f1} \odot \alpha))) \odot \diamond\Gamma_1'' \tag{6}$$
$$(\tau_{f1} \to \tau_{f2}, U_{f2}) \leq \tau_f. \tag{7}$$

By (2), (4), and (7), $\tau_2'' \leq \tau_{f1}$ and $\tau_{f2} \leq \tau_0$ and $U_{f2} \leq 1$. By (5) and rule T-FUN,

$$U_{f1} \odot (\mu\alpha.(1 \otimes (U_{f1} \odot \alpha))) \odot \diamond\Gamma_1'' \vdash \mathbf{fun}(f, x, M) : (\tau_{f1} \to \tau_{f2}, U_{f1}). \qquad (8)$$

By (5), (8), (3), and Lemma B.3,

$$\Gamma_1'' \otimes (U_{f1} \odot \mu\alpha.(1 \otimes (U_{f1} \odot \alpha)) \odot \diamond\Gamma_1'') \otimes \Gamma_2' \vdash [\mathbf{fun}(f, x, M)/f, v/x]M : \tau_{f2}.$$

Then, by Lemma A.2(3) and Lemma A.1(19),

$$\begin{aligned}(U_{f2} \odot \mu\alpha.(1 \otimes U_{f1} \odot \alpha)) \odot \diamond\Gamma_1'' &\leq \diamond\Gamma_1'' \otimes (U_{f1} \odot \mu\alpha.(1 \otimes (U_{f1} \odot \alpha)) \odot \diamond\Gamma_1'') \\ &\leq \Gamma_1'' \otimes (U_{f1} \odot \mu\alpha.(1 \otimes (U_{f1} \odot \alpha)) \odot \diamond\Gamma_1'')\end{aligned}$$

and thus

$$\Gamma_1' \otimes \Gamma_2' \vdash [\mathbf{fun}(f, x, M)/f, v/x]M : \tau_{f2}.$$

From (6), for any i, there exists U_{1i}' such that $U_{1i} \leq U_{1i}'$ and $U_{1i}'^{\Downarrow}$. (If $x_{1i} \notin dom(\Gamma_1'')$, take $\mathbf{0}$ for U_{1i}'.) Then, by Lemma A.1(9), $U_{1i}' \,; U_{4i} \leq U_{1i}' \otimes U_{4i}$, and so $U_{1i} \,; U_{4i} \leq U_{1i}' \otimes U_{4i}$. Thus, we have

$$(\Gamma_1 \otimes \Gamma_2), \tilde{x}_1 : (\mathbf{R}, \tilde{U}_1; \tilde{U}_4), \tilde{x}_2 : (\mathbf{R}, \tilde{U}_2), \tilde{x}_3 : (\mathbf{R}, \tilde{U}_3) \vdash [\mathbf{fun}(f, x, M)/f, v/x]M : \tau_{f2}.$$

By rules T-LETRES and T-SUB, we have

$$\begin{aligned}\Gamma_0 \vdash \mathbf{let_R}\ \tilde{x}_1 : (\tilde{U}_1 \,; \tilde{U}_4)\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x}_2 : \tilde{U}_2\ \mathbf{in}\ \mathbf{let_R}\ \tilde{x}_3 : \tilde{U}_3\ \mathbf{in}& \\ [\mathbf{fun}(f, x, M)/f, v/x]M : \tau_0,&\end{aligned}$$

finishing the case. □

ACKNOWLEDGMENTS

We would like to thank Manuel Fähndrich, Haruo Hosoya, Jakob Rehof, Tatsuro Sekiguchi, and Eijiro Sumii for discussions and comments. Comments from anonymous reviewers helped us improve the presentation of the article.

REFERENCES

AIKEN, A., FÄHNDRICH, M., AND LEVIEN, R. 1995. Improving region-based analysis of higher-order languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 174–185.

BALL, T. AND RAJAMANI, S. K. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 1–3.

BIGLIARDI, G. AND LANEVE, C. 2000. A type system for JVM threads. In *Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)* (Montreal, Que., Canada). ACM, New York.

BIRKEDAL, L., TOFTE, M., AND VEJLSTRUP, M. 1996. From region inference to von Neumann machines via region representation inference. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 171–183.

BLANCHET, B. 1998. Escape analysis: Correctness, proof, implementation and experimental results. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 25–37.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. Path-sensitive program verification in polynomial time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.

DeLine, R. and Fähndrich, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 59–69.

DeLine, R. and Fähndrich, M. 2002. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science Volume B*, J. V. Leeuwen, Ed. The MIT press/Elsevier, Chapter 16, 995–1072.

Flanagan, C. and Abadi, M. 1999a. Object types against races. In *CONCUR'99*. Lecture Notes in Computer Science, vol. 1664. Springer-Verlag, New York, 288–303.

Flanagan, C. and Abadi, M. 1999b. Types for safe locking. In *Proceedings of ESOP 1999*. Lecture Notes in Computer Science, vol. 1576. Springer-Verlag, New York, 91–108.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 234–245.

Foster, J. S., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York.

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

Girard, J.-Y. 1987. Linear logic. *Theoret. Comput. Sci. 50*, 1–102.

Gischer, J. 1981. Shuffle languages, Petri nets, and context-sensitive grammars. *Commun. ACM 24*, 9, 597–605.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. 2002. Region-based memory management in cyclone. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 282–293.

Gustavsson, J. and Svenningsson, J. 2000. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of IFL'00, Implementation of Functional Languages*. Lecture Notes in Computer Science, vol. 2011. Springer-Verlag, New York, 140–157.

Hannan, J. 1995. A type-based analysis for stack allocation in functional languages. In *Proceedings of SAS'95*. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, New York, 172–188.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 58–70.

Igarashi, A. and Kobayashi, N. 2000a. Garbage collection based on a linear type system. In *Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)* (Montreal, Que., Canada) ACM, New York. (Published as Technical Report CMU-CS-00-161, Carnegie Mellon University, Pittsburgh, PA.)

Igarashi, A. and Kobayashi, N. 2000b. Type reconstruction for linear pi-calculus with I/O subtyping. *Inf. Comput. 161*, 1–44.

Igarashi, A. and Kobayashi, N. 2003. A generic type system for the pi-calculus. *Theoret. Comput. Sci. 311*, 1–3 (Jan.), 121–163.

Iwama, F. and Kobayashi, N. 2002. A new type system for JVM lock primitives. In *Proceedings of ASIA-PEPM'02*. ACM Press. Available at `http://www.kb.cs.titech.ac.jp/~kobayasi/publications.html`.

Jędrzejowicz, J. and Szepietowski, A. 2001. Shuffle languages are in P. *Theoret. Comput. Sci. 250*, 1-2, 31–53.

Kanellakis, P. C., Mairson, H. G., and Mitchell, J. C. 1991. Unification and ML type reconstruction. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. D. Plotkin, Eds. The MIT Press, Cambridge, Mass., 444–478.

Kobayashi, N. 1999. Quasi-linear types. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 29–42.

Kobayashi, N. 2000a. Type-based useless variable elimination. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, 84–93.

KOBAYASHI, N. 2000b. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *Proceedings of IFIP International Conference on Theoretical Computer Science (TCS2000)*. Lecture Notes in Computer Science, vol. 1872. Springer-Verlag, New York, 365–389. (Invited Talk.)

KOBAYASHI, N. 2003. Time regions and effects for resource usage analysis. In *Proceedings of ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'03)*. ACM, New York, 50–61.

KOBAYASHI, N., SAITO, S., AND SUMII, E. 2000. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*. Lecture Notes in Computer Science, vol. 1877. Springer-Verlag, New York, 489–503.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N.J.

MORRISETT, G., FELLEISEN, M., AND HARPER, R. 1995. Abstract models of memory management. In *Proceedings of Functional Programming Languages and Computer Architecture*. 66–76.

NIELSON, F., NIELSON, H. R., AND HANKIN, C. 1999. *Principles of Program Analysis*. Springer-Verlag, New York,

REHOF, J. AND MOGENSEN, T. 1999. Tractable constraints in finite semilattices. *Sci. Comput. Prog. 35*, 2, 191–221.

SANGIORGI, D. AND WALKER, D. 2001. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge, Mass.

SUMII, E. AND KOBAYASHI, N. 1998. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*. ENTCS, vol. 16(3). 55–77.

TOFTE, M. AND TALPIN, J.-P. 1994. Implementation of the call-by-value lambda-calculus using a stack of regions. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 188–201.

TURNER, D. N., WADLER, P., AND MOSSIN, C. 1995. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture* (San Diego, Califf.), 1–11.

WADLER, P. 1990. Linear types can change the world! In *Programming Concepts and Methods*. North Holland, Amsterdam, The Netherland.

WALKER, D., CRARY, K., AND MORRISETT, J. G. 2000. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst. 22*, 4, 701–771.

WALKER, D. AND WATKINS, K. 2001. On linear types and regions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM, New York.

WANSBROUGH, K. AND PEYTON JONES, S. L. 1999. Once upon a polymorphic type. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 15–28.

Received May 2002; revised April 2003; accepted February 2004