Time Regions and Effects for Resource Usage Analysis

Naoki Kobayashi
Tokyo Institute of Technology
email:kobayasi@cs.titech.ac.jp

ABSTRACT

Various resources such as files and memory are associated with certain protocols about how they should be accessed. For example, a memory cell that has been allocated should be eventually deallocated, and after the deallocation, the cell should no longer be accessed. Igarashi and Kobayashi recently proposed a general type-based method to check whether a program follows such resource access policies, but their analysis was not precise enough for certain programs. In this paper, we refine their type-based analysis by introducing a new notion of time regions. The resulting analysis combines the merits of two major previous approaches to type-based analysis of resource usage – linear-type-based and effect-based approaches.

Categories and Subject Descriptors

F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Theory, Verification

Keywords

Types, Effects, Program Analysis, Resource Usage

1. INTRODUCTION

Various resources and library functions are often associated with certain protocols about how they should be accessed or called. For example, a memory cell that has been allocated should be eventually deallocated, and after the deallocation, the cell should not be read or updated. A file that has been opened should be eventually closed. A lock should be acquired before a shared resource is accessed. After the lock has been acquired, it should be eventually released.

To ensure correct usage of resources, a number of type systems have been proposed for specific types of resources [1, 4, 29, 31, 3, 10, 11, 18, 14]. Igarashi and Kobayashi [17] generalized the problem of checking correct usage of resources as a resource usage analysis problem and proposed a type-based program analysis to solve the generalized problem. DeLine and Fähndrich [7, 8] and Foster, Terauchi, and Aiken [13] have independently tackled similar problems and presented different type systems to solve the problems. Those methods can be used to check usage of various types of resources in a uniform manner.

Unfortunately, however, none of those type systems seems to be satisfactorily precise and clear enough: They are imprecise for certain programs, and/or require rather complex program annotations to guide the analyses. Full proofs of soundness of the analyses are either very complex [17] or do not exist [13, 8]. To overcome this problem, the present paper proposes a new type system for resource usage analysis and proves its soundness. It combines some of the ideas of the previous type systems, and introduces a new notion called *time regions*. The resulting type system is more precise at least than Igarashi and Kobayashi's analysis [17], and also seems to capture the essence of the special programming constructs introduced in other type systems [13, 8].

In the rest of this section, we first review the resource usage analysis problem [17], and discuss two major approaches to solving the problem – linear-type-based approach [17, 18] and effect-based approach [1, 29, 13, 10, 7] – and drawbacks of them. We then explain ideas of our new type system.

1.1 Previous approaches to resource usage analysis

The resource usage analysis problem [17] is a problem of statically checking whether a program accesses each resource in an order intended by the programmer. For example, let us write ρ for a read operation on a file (pointer), W for a write operation, and C for a close operation. Then, we can specify a policy that a file should be eventually closed and should not be accessed afterwards by a regular expression $(R+W)^*C$, which denotes the set of valid access orders. The usage of a read-only file can be expressed by R^*C.[1] The purpose of the resource usage analysis is to check whether a program conforms to this kind of specification for each resource.

1.1.1 *Linear-type-based approach*

Igarashi and Kobayashi [17] proposed a type-based method for resource usage analysis for a λ-calculus extended

[1] We assume that the "open" operation is implicitly performed when the file (pointer) is allocated.

TLDI'03, January 18, 2003, New Orleans, Louisiana, USA.

with primitives to create and access abstract resources. The basic idea is to annotate a resource type with information about the resource access order. For example, a read-only file is given a type $(\mathbf{File}, R^*C)$, which expresses not only that it is a file but also that it can be first read an arbitrary number of times and then it should be closed. (Actually, they used a more expressive language called *usage expressions* to express information about the access order instead of regular expressions.)

The usual typing rules were extended to reflect the extra information about resource usage. For example, an ordinary typing rule for let-expressions:

$$\frac{\Gamma \triangleright M : \tau_1 \qquad \Gamma, x : \tau_1 \triangleright N : \tau_2}{\Gamma \triangleright \mathbf{let}\ x = M\ \mathbf{in}\ N : \tau_2}$$

was replaced by:

$$\frac{\Gamma \triangleright M : \tau_1 \qquad \Delta, x : \tau_1 \triangleright N : \tau_2}{\Gamma; \Delta \triangleright \mathbf{let}\ x = M\ \mathbf{in}\ N : \tau_2}$$

Here, $\Gamma; \Delta$ denotes the type environment obtained by combining the usage part of each resource type in Γ and Δ with a sequential composition operator. For example, if $\Gamma = x : (\mathbf{File}, R^*)$ and $\Delta = x : (\mathbf{File}, C)$, then $\Gamma; \Delta = x : (\mathbf{File}, R^*C)$. By performing type reconstruction based on the extended type system, one can estimate the set of possible access sequences for each resource.

Igarashi and Kobayashi's analysis is unfortunately not precise enough for certain programs. To understand the problem, consider the following program M_0:

$$\mathbf{let}\ f = \lambda z.\mathbf{close}(x)\ \mathbf{in}\ (\mathbf{read}(x); f(\,)).$$

In the call-by-value semantics, x is first read and then closed. Igarashi and Kobayashi's type system cannot, however, derive such a property. The type derivation for the above program M_0 is:

$$\frac{\begin{array}{c} x : (\mathbf{File}, \Diamond C) \triangleright \lambda z.\mathbf{close}(x) : \mathbf{bool} \to \mathbf{bool} \\ x : (\mathbf{File}, R), f : \mathbf{bool} \to \mathbf{bool} \triangleright \mathbf{read}(x); f(\,) : \mathbf{bool} \end{array}}{x : (\mathbf{File}, (\Diamond C); R) \triangleright M_0 : \mathbf{bool}}$$

Here, $\Diamond C$ in the sub-derivation $x : (\mathbf{File}, \Diamond C) \triangleright \lambda z.\mathbf{close}(x) : \mathbf{bool} \to \mathbf{bool}$ means that x is accessed at some time in the future. The conclusion implies that x is closed once and read once, but we do not know the order in which x is closed and read. (Notice that without the $\Diamond$-operator, we would obtain a wrong conclusion: $x : (\mathbf{File}, C; R) \triangleright M_0 : \mathbf{bool}$.) The $\Diamond$-operator is applied at each place where a closure is created, and hence this kind of loss of information occurs in many places. Similar loss of information occurs also when resources are stored in reference cells.

1.1.2 Effect-based approach

Another major approach [29, 10, 13] is to use an effect system [28]. The idea of an effect system is to associate each term with not only its type but an effect, which describes the side effects that may occur when the term is evaluated. A function type $\tau_1 \to \tau_2$ is replaced by $\tau_1 \xrightarrow{\varphi} \tau_2$ where the effect φ expresses latent effects that may occur when the function is called. A type judgment is of the form $\Gamma \triangleright M : \tau \mid \varphi$, which means that M has type τ and effect φ under the type environment Γ. In the context of resource usage analysis, we can regard an effect as a set of possible resource access sequences.

Let us write $(\mathbf{File}, \rho)$ for the type of a file that belongs to a set ρ of files. (ρ is often called a *region* [28, 30, 13]; It was originally introduced to represent abstract memory space.) Then, the expression $\lambda z.\mathbf{close}(x)$ is typed as follows:

$$\frac{\dfrac{\cdots}{x : (\mathbf{File}, \rho), z : \mathbf{bool} \triangleright \mathbf{close}(x) : \mathbf{bool} \mid \{\rho : C\}}}{x : (\mathbf{File}, \rho) \triangleright \lambda z.\mathbf{close}(x) : \mathbf{bool} \xrightarrow{\{\rho:C\}} \mathbf{bool} \mid \emptyset}$$

The conclusion implies that $\lambda z.\mathbf{close}(x)$ is a function of type $\mathbf{bool} \xrightarrow{\{\rho:C\}} \mathbf{bool}$ and that no resource access occurs when the term is evaluated but a file in region ρ is closed when the function is invoked. So, unlike the corresponding judgment $x : (\mathbf{File}, \Diamond C) \triangleright \lambda z.\mathbf{close}(x) : \mathbf{bool} \to \mathbf{bool}$ in the linear-type-based approach, the type judgment above precisely expresses *when* x is closed.

A problem occurs, however, when multiple resources are aliased to the same region. Consider the following judgment:

$$x : (\mathbf{File}, \rho), y : (\mathbf{File}, \rho) \triangleright M : \mathbf{bool} \mid \{\rho : C; \rho : C\}$$

The judgment implies that files in region ρ is closed twice in total, but we do not know whether each of x and y is closed once, or either x or y is closed twice. Aliasing of different resources to the same region (in the above case, region ρ refers to different resources x and y) is common: It happens when a monomorphic function is applied to different resources or when an aggregate data structure is used. For example, let l be a list of files and *CloseAll* be a function that takes a list as an argument and closes all the files in the list. Then, *CloseAll*(l) would be typed as:

$$l : (\mathbf{File}, \rho)\ \mathbf{list} \triangleright \mathit{CloseAll}(l) : \mathbf{bool} \mid (\rho : C)^*$$

where the effect $(\rho : C)^*$ means that files in region ρ is repeatedly closed. The judgment does not tell us whether each element of the list is closed once or not. For this program, the following judgment in the linear-type-based approach is better:

$$l : (\mathbf{File}, C)\ \mathbf{list} \triangleright \mathit{CloseAll}(l) : \mathbf{bool}$$

since it says that each element of l is closed once.

Because of the problem above, the effect-based-approach can keep track of the resource access order only when a region refers to a single resource (see [13], for example). Foster et al. [13] and DeLine and Fähndrich [8] introduced special constructs to avoid this problem with a help of programming annotations, but their solutions do not seem to be general and simple enough. For example, those solutions do not seem to work for concurrent programs. (More discussion on this point is found in Section 4.)

1.2 Our Approach

As discussed in the previous section, both the linear-type-based approach and the effect-based approach have advantages and disadvantages: the former is good at analyzing how often each resource is used, but not at analyzing when each resource is accessed, while the latter is good at analyzing when each resource is accessed, but not at analyzing how often each resource is accessed.

To take the best of both worlds, we introduce a new concept called *time regions*. While a region in type-effect systems [28, 29] is originally introduced to abstract a set of memory cells, a time region abstracts a set of time periods. Recall the judgment $x : (\mathbf{File}, \Diamond C) \triangleright \lambda z.\mathbf{close}(x) : \mathbf{bool} \to$

bool. We replace the type $(\mathbf{File}, \Diamond C)$ with a new type $(\mathbf{File}, \Diamond_\rho C)$, which means not only that the file is closed just once, but also that the close operation is performed *in a time period of region* ρ. The judgment is replaced by:

$$x : (\mathbf{File}, \Diamond_\rho C) \triangleright \lambda z.\mathbf{close}(x) : \mathbf{bool} \xrightarrow{\rho} \mathbf{bool} \mid \emptyset$$

The judgment means that if x is a file that should be closed once in a time period of region ρ, then $\lambda z.\mathbf{close}(x)$ can be used as a function from booleans to booleans and that a time period of region ρ arrives when the function is called.

Let us consider another example. In the judgment:

$$l : (\mathbf{File}, \Diamond_\rho C)\ \mathbf{list} \triangleright \mathit{CloseAll}(l) : \mathbf{bool} \mid \rho^*,$$

$l : (\mathbf{File}, \Diamond_\rho C)$ **list** implies that each element of l is closed once and that the close operation is performed in a period of region ρ. The effect ρ^* implies that time periods of region ρ arrive successively. So, the judgment means that each element of l is closed once when the expression is evaluated.

In this manner, we can analyze how often each resource is accessed as precisely as in the linear-type-based approach and we can also analyze the resource access order as precisely as in the effect-based approach. Based on this idea, we formalize a new type system for resource usage analysis and prove its soundness. The main focus of this paper is to obtain a sound, general type system for resource usage analysis, and many practical issues such as a type inference algorithm (i.e., an algorithm for the analysis) are left for future work; we believe that the algorithm can be obtained in a manner to similar to the algorithm for Igarashi and Kobayashi's type system [17].

1.3 The Rest of This Paper

Section 2 introduces the syntax of the target language of our resource usage analysis. Section 3 presents our new type system for resource usage analysis, and shows its soundness. Section 4 discusses related work and Section 5 concludes. Because of the space restriction, some of the detailed definitions and proofs are omitted. They are found in the full version of this paper, available from `http://www.kb.cs.titech.ac.jp/~kobayasi/publications.html`. The full version also briefly discusses type inference issues and how to extend our type system to deal with lists, exceptions, and pointers.

2. LANGUAGE

This section introduces a call-by-value λ-calculus extended with primitives for creating and accessing resources, as a target language for our resource usage analysis. It is basically the same as the target language of our previous resource usage analysis [17], except that some simplification has been made.

2.1 Syntax

We assume that there is a countably infinite set $\mathcal{L}$ of labels, ranged over by meta-variable l. A label expresses the kind of a resource access operation (like R, W, C in Section 1). We write $\mathcal{L}^*$ for the set of finite sequences of labels, and write $\mathcal{L}^{*,\downarrow}$ for the set $\mathcal{L}^* \cup \{s\downarrow \mid s \in \mathcal{L}^*\}$. The special symbol '$\downarrow$' is used to denote the termination of a program. We call an element of $\mathcal{L}^{*,\downarrow}$ a *trace*. We write ϵ for the empty sequence, and $s_1 s_2$ for the concatenation of two sequences s_1 and s_2.

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

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

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

$\mathbf{fix}(f, x, M)$ denotes a recursive function f such that $f(x) = M$. When f does not appear free in M, we write $\lambda x.M$ for $\mathbf{fix}(f, x, M)$. An expression $\mathbf{new}^\Phi()$ creates a new resource. (For the sake of simplicity, we consider a single kind of resource here.) A trace set Φ denotes the programmer's intention on how the resource should be accessed afterwards. 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 until some time during evaluation. For example, for a read-only file, Φ would be the prefix-closure of the set $R^*C\downarrow$. We do not fix a particular way to specify trace sets Φ. They could be specified in various ways, using regular expressions, context-free grammars, etc.

An expression $\mathbf{acc}_l(M)$ evaluates M to a resource, and accesses it with the resource access primitive $\mathbf{acc}_l$. For the sake of simplicity, we assume that every access primitive always returns **true** or **false** in a non-deterministic manner. We often write **read**, **write**, and **close** for $\mathbf{acc}_R$, $\mathbf{acc}_W$, and $\mathbf{acc}_C$ below.

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 **let** $x = M_1$ **in** M_2 or $M_1; M_2$ (when $x \notin \mathbf{FV}(M_2)$) for $(\lambda x.M_2)M_1$;

Example 1. Consider the four expressions in Figure 1. Suppose that Φ denotes the set $(R^*C\downarrow)^\sharp$, which means that the resource can be read an arbitrary number of times and then it should be closed before the program terminates. Example A defines a recursive function that takes a resource x as an argument, reads it several times and closes it, and then applies the function to a new resource. Example B creates a new resource and a closure to close the resource, and then reads the resource and calls the closure. Both expressions conform to the resource access policy Φ. On the other hand, Examples C and D violates the resource access policy: Example C tries to update a read-only resource, and Example D forgets to close a resource in the else-clause. The purpose of our resource usage analysis is to accept valid programs like A and B, and to reject invalid programs like C and D. Igarashi and Kobayashi's resource usage analysis [16] is so conservative that it rejects not only C and D but also B. Our analysis presented in this paper accepts both A and B, and rejects C and D.

2.2 Operational Semantics

The operational semantics of the language is defined in the usual manner, except that each run-time state keeps track of how each resource can be used afterwards and that a program is aborted with an error if it performs an invalid resource access. A run-time state is expressed as a dynamic expression introduced below.

Example A:

$$\textbf{let } f = \textbf{fix}(f, x, \textbf{if read}(x) \textbf{ then } f(x) \textbf{ else close}(x)) \textbf{ in } f(\textbf{new}^{\Phi}())$$

Example B:

$$\textbf{let } x = \textbf{new}^{\Phi}() \textbf{ in let } \mathit{finish} = \lambda y.\textbf{close}(x) \textbf{ in } (\textbf{read}(x); \mathit{finish}(\textbf{true}))$$

Example C:

$$\textbf{let } x = \textbf{new}^{\Phi}() \textbf{ in write}(x)$$

Example D:

$$\textbf{let } x = \textbf{new}^{\Phi}() \textbf{ in if read}(x) \textbf{ then close}(x) \textbf{ else true}$$

Figure 1: Examples of Expressions

Definition 2 (dynamic expressions). *The set $\mathcal{D}$ of dynamic expressions, ranged over by D, is defined by the following syntax.*

$$\begin{array}{rcl} D\textit{(dynamic expressions)} & ::= & \textbf{let } H \textbf{ in } M \\ H\textit{(heaps)} & ::= & x_1 = h_1; \cdots ; x_n = h_n \\ h\textit{(heap values)} & ::= & \textbf{new}^{\Phi}() \mid \textbf{fix}(f, x, M) \end{array}$$

We write ϵ for the empty heap. We assume that $x_1, \ldots, x_n$ in $x_1 = h_1; \cdots ; x_n = h_n$ are distinct from each other. When $H = (x_1 = h_1; \cdots ; x_n = h_n)$, we write $H(x_i)$ for h_i and write $dom(H)$ for the set $\{x_1, \ldots, x_n\}$.

Definition 3 (evaluation context).

$$\begin{array}{l} \mathcal{E} ::= [\,] \mid \textbf{acc}_l(\mathcal{E}) \mid \mathcal{E}M \mid v\mathcal{E} \mid \textbf{if } \mathcal{E} \textbf{ then } M_1 \textbf{ else } M_2 \\ v ::= x \mid \textbf{true} \mid \textbf{false} \end{array}$$

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

Definition 4. *We write Φ^{-l} for the set $\{s \mid ls \in \Phi\}$. Suppose $H = (x_1 = h_1; \cdots ; x_n = h_n)$. If $h_i = \textbf{new}^{\Phi}()$ and $\Phi^{-l} \neq \emptyset$, then $H - (x_i, l)$ is defined to be the heap $x_1 = h'_1; \cdots ; x_n = h'_n$ where $h'_i = \textbf{new}^{\Phi^{-l}}()$ and $h'_j = h_j$ for each $j \neq i$.*

The reduction relation $\rightsquigarrow \subseteq \mathcal{D} \times (\mathcal{D} \cup \{\textbf{Error}\})$ is the least relation closed under the rules in Figure 2. In the figure, $[v/x]M$ expresses an expression obtained by replacing every free occurrence of x in M with v. **Error** means that a resource access error has occurred.

3. TYPE SYSTEM

This section introduces a type system for resource usage analysis, which ensures that no well-typed expression violates the resource access policy. We assume that there is a countably infinite set of region variables (or simply called regions), ranged over by ρ.

3.1 Usages

As explained in Section 1, we annotate each resource type with a *usage*, which denotes how a resource can be accessed.

Definition 5 (usages, effects). *The set of* usages *is defined by the following syntax:*

$$U ::= \mathbf{0} \mid l \mid U_1 \,;\, U_2 \mid U_1 \otimes U_2 \mid\ !U \mid \Diamond_\rho U \mid \overline{\rho} \mid \exists\rho.U$$

When a usage does not contain a term of the form $l (\in \mathcal{L})$, the usage is also called an effect.

We use a meta-variable φ for an effect. The prefix $\exists\rho.$ binds region ρ. We write $\textbf{Fregs}(U)$ for the set of free (i.e., not bound) regions. We give higher precedence to unary operators than binary operators. We also give higher precedence to $\otimes$ than to ;.

Intuitive meaning of usages is summarized in Table 1. The usage $!U$ describes a resource that can be accessed according to one of $\mathbf{0}$, U, $U \otimes U$, $U \otimes U \otimes U$, ...

The meaning of a usage can be understood using a time chart. The usage $(W \otimes R) \,;\, C$ expresses the access order shown in Figure 3. The figure indicates that a resource is written and read once during the first time period, and then closed during the second period. Similarly, the usage $(R \,;\, R) \otimes W$ indicates that a resource is read once during each of the first and second period, and that in parallel to the read accesses, a write access may occur. So, the set of possible access sequences is $\{RRW, RWR, WRR\}$.

The time chart for the usage $\Diamond_{\rho_1} C; \Diamond_{\rho_2} R; \overline{\rho_2}; W; \overline{\rho_1}$ in Figure 3. means (1) in the first and second periods, close and read accesses are delayed until periods belonging to regions ρ_1 and ρ_2 respectively, (2) the third period belongs to region ρ_2 (so that the delayed read access may occur in this period), (3) in the fourth period, a write access occurs, and (4) the fifth period belongs to region ρ_1 (so that the delayed close access may occur in this period). Since there may be other time periods belonging to region ρ_1 and ρ_2, we cannot conclude that $\Diamond_{\rho_1} C; \Diamond_{\rho_2} R; \overline{\rho_2}; W; \overline{\rho_1}$ expresses the access order RWC (for example, it may be a part of the usage $(\Diamond_{\rho_1} C; \Diamond_{\rho_2} R; \overline{\rho_2}; W; \overline{\rho_1}) \otimes \overline{\rho_1}$, in which case the close access may occur at any time. $\exists\rho.U$ ensures that a period of region ρ does not occur outside U. So, $\exists\rho_1.\exists\rho_2.(\Diamond_{\rho_1} C; \Diamond_{\rho_2} R; \overline{\rho_2}; W; \overline{\rho_1})$ means that RWC is the only possible access sequence.

Note that only a single occurrence of $\Diamond_\rho U$ is activated in a single time period specified by $\overline{\rho}$. So, $\Diamond_{\rho_1} R \otimes \Diamond_{\rho_1} C \otimes \Diamond_{\rho_2} W \otimes (\overline{\rho_1}; \overline{\rho_2}; \overline{\rho_1})$ represents access sequences RWC and CWR but not RCW. We write $U_1 \,\&\, U_2$ for $\exists\rho.(!\Diamond_\rho U_1 \otimes !\Diamond_\rho U_2 \otimes \overline{\rho})$. A resource of the usage $U_1 \,\&\, U_2$ can be accessed according to either U_1 or U_2. Similarly, we can express a recursive usage $\mu\alpha.U$ such that $\mu\alpha.U = [\mu\alpha.U/\alpha]U$ by $\exists\rho.(!\Diamond_\rho[\overline{\rho}/\alpha]U \otimes \overline{\rho})$.

The formal semantics of usages is defined using the labeled transition semantics shown in Figure 4. The unary relation $U^{\Downarrow}$, defined as the least relation closed under the rules in

$$\textbf{let } H \textbf{ in } \mathcal{E}[h] \rightsquigarrow \textbf{let } H; x = h \textbf{ in } \mathcal{E}[x] \qquad (x \text{ fresh}) \tag{R-Alloc}$$

$$\frac{b \in \{\textbf{true}, \textbf{false}\} \qquad H' = H - (x, l)}{\textbf{let } H \textbf{ in } \mathcal{E}[\textbf{acc}_l(x)] \rightsquigarrow \textbf{let } H' \textbf{ in } \mathcal{E}[b]} \tag{R-Acc}$$

$$\frac{H - (x, l) \text{ is undefined}}{\textbf{let } H \textbf{ in } \mathcal{E}[\textbf{acc}_l(x)] \rightsquigarrow \textbf{Error}} \tag{R-AccError}$$

$$\frac{H(x) = \textbf{fix}(f, z, M)}{\textbf{let } H \textbf{ in } \mathcal{E}[xv] \rightsquigarrow \textbf{let } H \textbf{ in } \mathcal{E}[[x/f][v/z]M]} \tag{R-App}$$

$$\textbf{let } H \textbf{ in } \mathcal{E}[\textbf{if true then } M_1 \textbf{ else } M_2] \rightsquigarrow \textbf{let } H \textbf{ in } \mathcal{E}[M_1] \tag{R-IfT}$$

$$\textbf{let } H \textbf{ in } \mathcal{E}[\textbf{if false then } M_1 \textbf{ else } M_2] \rightsquigarrow \textbf{let } H \textbf{ in } \mathcal{E}[M_2] \tag{R-IfF}$$

Figure 2: Operational semantics

Usages	Interpretation
$\mathbf{0}$	Cannot be accessed at all
l	Accessed once through the primitive $\mathbf{acc}_l$
$U_1 ; U_2$	Sequential access (i.e., accessed first according to U_1 and then according to U_2)
$U_1 \otimes U_2$	Interleaving (i.e., accessed according to U_1 and U_2 in an interleaving manner
$!U$	Accessed according to U an arbitrary number of times in an interleaving manner
$\Diamond_\rho U$	Accessed according to U later, in a time period of region ρ
$\overline{\rho}$	Accessed according to the access pattern associated (by $\Diamond_\rho U$) with time region r
$\exists \rho . U$	Accessed according to U, which may contain a new time region ρ

Table 1: Meaning of Usage Expressions

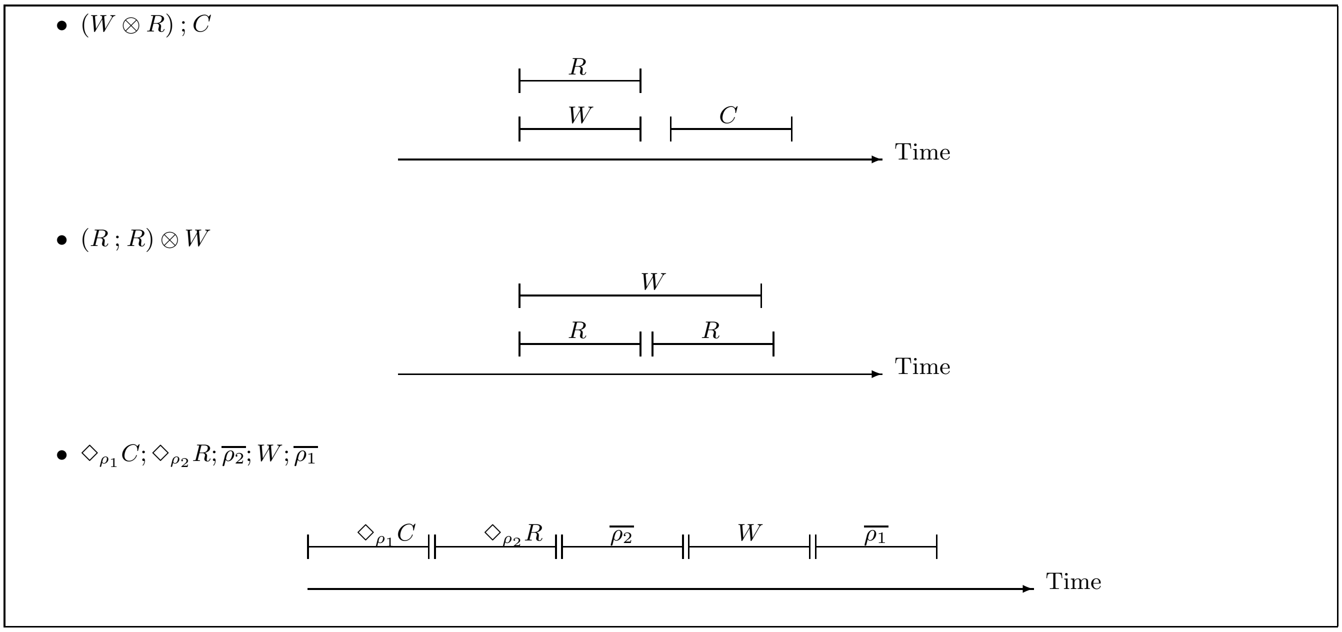

Figure 3: The access orders represented by $(W \otimes R) ; C$,$(R ; R) \otimes W$, and $\Diamond_{\rho_1} C; \Diamond_{\rho_2} R; \overline{\rho_2}; W; \overline{\rho_1}$

the figure, means that U contains only delayed access. The binary relation $U \preceq U'$ is defined as the least reflexive and transitive relation closed under the rules in the figure. Here, E_U in the figure ranges over the set of usage reduction contexts defined by the following syntax.

$$E_U \quad ::= \quad [\,] \mid E_U \,;U \mid E_U \otimes U$$

The relation $\preceq$ is introduced to simplify the transition rules; It corresponds to structural relations used to define reduction semantics of processes [24, 26, 21].

The transition relation $U \xrightarrow{l} U'$ is defined as the least relation closed under the rules in Figure 4. Here, $active(U)$ denotes the set $\{\rho \mid \overline{\rho} \text{ appears in } U\}$, and $delayed(U)$ denotes the set $\{\rho \mid \Diamond_\rho U' \text{ appears in } U\}$. $U \xrightarrow{l} U'$ means that a resource of usage U can be first accessed by the primitive $\mathbf{acc}_l$ and then accessed according to U'. The transition semantics is similar to that of process calculi (e.g., CCS [23]); $\Diamond_\rho$ and $\overline{\rho}$ can be interpreted as co-actions, and $\otimes$ and ; can be interpreted as parallel composition and sequential composition respectively. Note, however, that $\Diamond_\rho$ and $\overline{\rho}$ should not be confused with actions of CCS: $(\Diamond_\rho U_1) \otimes (\overline{\rho}; U_2) \preceq U_1\,; U_2$ holds (so that U_2 is activated only after U_1 becomes stable) while $\rho.P_1|\overline{\rho}.P_2$ is reduced to $P_1|P_2$ in CCS (so that P_2 does not wait for P_1 to finish execution).

We write $[\![\, U \,]\!]$ for the trace set:

$$\begin{array}{l}\{l_1 \cdots l_n \mid U^c \xrightarrow{l_1} \cdots \xrightarrow{l_n} U'\} \\ \cup\{l_1 \cdots l_n \downarrow \mid U^c \xrightarrow{l_1} \cdots \xrightarrow{l_n} U' \wedge U' \preceq \mathbf{0}\}\end{array}$$

where $U^c = U \otimes !\overline{\rho_1} \otimes \cdots \otimes !\overline{\rho_n} \otimes !\Diamond_{\rho_1}\mathbf{0} \otimes \cdots \otimes !\Diamond_{\rho_n}\mathbf{0}$ for $\{\rho_1, \ldots, \rho_n\} = \mathbf{Fregs}(U)$. The trace set $[\![\, U \,]\!]$ expresses the set of possible access sequences denoted by U. When gathering the access sequences, we assume that the environment can perform any access to free regions; that assumption is expressed by the part $!\overline{\rho_1} \otimes \cdots \otimes !\overline{\rho_n} \otimes !\Diamond_{\rho_1}\mathbf{0} \otimes \cdots \otimes !\Diamond_{\rho_n}\mathbf{0}$ of U^c.

Example 2. $U = \exists\rho_1.\exists\rho_2.(\Diamond_{\rho_1}C; \Diamond_{\rho_2}R; \overline{\rho_2}; W; \overline{\rho_1})$ is reduced as follows:

$$\begin{array}{rcl} U & \xrightarrow{R} & \exists\rho_1.\exists\rho_2.(\Diamond_{\rho_1}C \otimes (W; \overline{\rho_1})) \\ & \xrightarrow{W} & \exists\rho_1.\exists\rho_2.(\Diamond_{\rho_1}C \otimes \overline{\rho_1}) \\ & \xrightarrow{C} & \mathbf{0} \end{array}$$

So, $[\![\, U \,]\!] = \{\epsilon, R, RW, RWC, RWC\downarrow\}$

3.2 Types and type environments

Definition 6 (types, type environments). *The sets of* types *and* type environments*, ranged over by τ and Γ respectively, are defined by the following syntax.*

$$\begin{array}{lcl} \tau & ::= & \mathbf{bool} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid (\mathbf{R}, U) \\ \Gamma & ::= & \mathbf{0} \mid x : \tau \mid \Gamma_1\,; \Gamma_2 \mid \Gamma_1 \otimes \Gamma_2 \mid !\Gamma \mid \Diamond_\rho \Gamma \mid \exists\rho.\Gamma \mid \overline{\rho} \end{array}$$

The effect φ in a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ describes the time periods that occur when the function is called. For example, $\mathbf{bool} \xrightarrow{\overline{\rho_1}; \overline{\rho_2}} \mathbf{bool}$ means that time periods in regions ρ_1 and ρ_2 arrive in this order when a function of this type is called. $(\mathbf{R}, U)$ is the type of a resource that should be accessed according to U.

A type environment is usually defined as a set of type bindings of the form $x\!:\!\tau$. Instead, we define a type environment as a term obtained by composing type bindings with sequential composition (;), parallel composition ($\otimes$), etc. $\mathbf{0}$ expresses the empty type environment (which means that no variables can be accessed). The meaning of the other constructors for type environments is analogous to that of the corresponding usage constructors. The constructors are overloaded so that an effect is not only a usage but also a type environment.

Using a type environment, we can express the order between accesses to different variables. For example, $(x : (\mathbf{R}, R))\,; (y : (\mathbf{R}, W))$ means that x and y are both resources and that x should be first read and then y should be written. A similar notion of type environments was introduced in type systems for process calculi [16].

Types τ and τ' are compatible, written $\tau \sim \tau'$, if τ and τ' are identical or if $\tau = (\mathbf{R}, U)$ and $\tau' = (\mathbf{R}, U')$ for some U and U'. A type environment Γ is *well-formed* if, for each x, $\tau \sim \tau'$ holds whenever whenever Γ contains $x : \tau$ and $x : \tau'$. For example, $x : (\mathbf{R}, U_1)\,; x : (\mathbf{R}, U_2)$ is well-formed but $(x : (\mathbf{R}, U)) \otimes (x : \tau_1 \xrightarrow{\varphi} \tau_2)$ is not. We consider only well-formed type environments below.

We write $\mathbf{Fregs}(\tau)$ and $\mathbf{Fregs}(\Gamma)$ for the set of free region variables appearing in τ and Γ respectively. $\mathbf{IFregs}(\tau)$, the set of free *inner* region variables (free region variables excluding those appearing in outermost usages), is defined by:

$$\begin{array}{l} \mathbf{IFregs}(\tau) = \begin{cases} \emptyset & \text{if } \tau \text{ is a resource type } (\mathbf{R}, U) \\ \mathbf{Fregs}(\tau) & \text{otherwise} \end{cases} \\ \mathbf{IFregs}(\mathbf{0}) = \mathbf{IFregs}(\overline{\rho}) = \emptyset \\ \mathbf{IFregs}(x : \tau) = \mathbf{IFregs}(\tau) \\ \mathbf{IFregs}(\Gamma_1\,; \Gamma_2) = \mathbf{IFregs}(\Gamma_1) \cup \mathbf{IFregs}(\Gamma_2) \\ \mathbf{IFregs}(\Gamma_1 \otimes \Gamma_2) = \mathbf{IFregs}(\Gamma_1) \cup \mathbf{IFregs}(\Gamma_2) \\ \mathbf{IFregs}(!\Gamma) = \mathbf{IFregs}(\Diamond_\rho \Gamma) = \mathbf{IFregs}(\Gamma) \\ \mathbf{IFregs}(\exists\rho.\Gamma) = \mathbf{IFregs}(\Gamma)\backslash\{\rho\} \end{array}$$

3.3 Subtyping

This section introduces a subtyping relation $\tau_1 \leq \tau_2$: As usual, it means that a value of type τ_1 can be used as a value of type τ_2.

We first define a sub-usage relation $U_1 \leq U_2$, which means that U_1 describes a more liberal usage of resources than U_2, so that a resource of usage U_1 can be accessed according to U_2. A *usage context* is a term obtained from a usage by replacing one of its sub-terms with a hole $[\,]$. If C_U is a usage context, we write $C_U[U]$ for a usage obtained by replacing $[\,]$ in C_U with U.

Definition 7 (sub-usage). *The sub-usage relation $U_1 \leq U_2$ is defined by:*

$$U_1 \leq U_2 \Leftrightarrow \forall C_U.([\![\, C_U[U_1] \,]\!] \supseteq [\![\, C_U[U_2] \,]\!])$$

For example, $U_1 \otimes U_2 \leq U_1\,; U_2$ holds for any U_1 and U_2. Both $\exists\rho.(\Diamond_\rho U\,; \overline{\rho}) \leq U$ and $U \leq \exists\rho.(\Diamond_\rho U\,; \overline{\rho})$ hold.

Definition 8 (subtyping). *The subtyping relation $\tau_1 \leq \tau_2$ holds if $\tau_1 = \tau_2$ or if $\tau_1 = (\mathbf{R}, U_1)$ and $\tau_2 = (\mathbf{R}, U_2)$ with $U_1 \leq U_2$ for some U_1, U_2.*

For the sake of simplicity, function types are defined to be related only if they are identical.

Similarly, a sub-type-environment relation $\Gamma \leq \Gamma'$ means that Γ expresses more liberal usage of variables than Γ'. For example, $(x : \tau \otimes y : \tau') \leq (x : \tau\,; y : \tau')$ holds. The definition is given in the full version.

Unary relation $\cdot^{\Downarrow}$ on usages:

$$\mathbf{0}^{\Downarrow}$$

$$\frac{U_1{}^{\Downarrow} \qquad U_2{}^{\Downarrow}}{(U_1\,;U_2)^{\Downarrow}}$$

$$(\Diamond_\rho U)^{\Downarrow}$$

$$\frac{U^{\Downarrow}}{(\exists\rho.U)^{\Downarrow}}$$

$$\frac{U_1{}^{\Downarrow} \qquad U_2{}^{\Downarrow}}{(U_1 \otimes U_2)^{\Downarrow}}$$

Relation $U \preceq U'$:

$$U \otimes \mathbf{0} \preceq U$$

$$!U \preceq U \otimes !U$$

$$\exists\rho.\mathbf{0} \preceq \mathbf{0}$$

$$\frac{U_1{}^{\Downarrow}}{U_1\,;U_2 \preceq U_1 \otimes U_2}$$

$$E_U[\exists\rho.U] \preceq \exists\rho.E_U[U] \quad (\rho \notin \mathbf{Fregs}(E_U))$$

$$\frac{U \preceq U'}{E_U[U] \preceq E_U[U']}$$

$$U_1 \otimes U_2 \preceq U_2 \otimes U_1$$

$$\Diamond_\rho U_1 \otimes E_U[\overline{\rho}] \preceq E_U[U_1]$$

$$(U_1 \otimes U_2) \otimes U_3 \preceq U_1 \otimes (U_2 \otimes U_3)$$

$$\frac{U \preceq U'}{\exists\rho.U \preceq \exists\rho.U'}$$

$$!U \preceq \mathbf{0}$$

Relation $U \xrightarrow{l} U'$:

$$E_U[l] \xrightarrow{l} E_U[\mathbf{0}] \qquad \text{(UR-Lab)}$$

$$\frac{U \xrightarrow{l} U' \qquad \rho \in \mathit{active}(U') \Leftrightarrow \rho \in \mathit{delayed}(U')}{\exists\rho.U \xrightarrow{l} \exists\rho.U'} \qquad \text{(UR-New)}$$

$$\frac{U_1 \preceq U_1' \qquad U_1' \xrightarrow{l} U_2' \qquad U_2' \preceq U_2}{U_1 \xrightarrow{l} U_2} \qquad \text{(UR-PCong)}$$

Figure 4: Labeled transition semantics of usages

3.4 Typing

A type judgment is of the form $\Gamma \triangleright M : \tau$. It means that M evaluates to a value of type τ, and that the resources bound to free variables are accessed according to Γ. For example, $x{:}(\mathbf{R}, R);y{:}(\mathbf{R}, W);x{:}(\mathbf{R}, C) \triangleright M : \mathbf{bool}$ implies that during evaluation of M, x is first read, y is written, and then x is closed. An effect of M explained in Section 1 is merged into a type environment. For example, the judgment

$$l : (\mathbf{File}, \Diamond_\rho C)\ \mathbf{list} \triangleright \mathit{CloseAll}(l) : \mathbf{bool} \mid \rho^*$$

in Section 1 is actually represented by:

$$(l : (\mathbf{File}, \Diamond_\rho C)\ \mathbf{list})\ ;\ !\overline{\rho} \triangleright \mathit{CloseAll}(l) : \mathbf{bool}$$

in our type system (if we have list types).

The relation $\Gamma \triangleright M : \tau$ is defined as the least relation closed under the rules in Figure 5. In the figure, we write "Γ_1, Γ_2" for $\Gamma_1 \otimes \Gamma_2$ if $dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$.

We explain the main rules below.

Rule (T-NEW)**:** U expresses how the new resource can be used. The premise $[\![U]\!] \subseteq \Phi$ ensures that U expresses only valid access sequences. $\mathit{Effect}(U)$ denotes the effect obtained by replacing every occurrence of $l \in \mathcal{L}$ in U with $\mathbf{0}$. Note that $\mathit{Effect}(U)$ cannot be replaced by $\mathbf{0}$: For example, if U is $\overline{\rho} \otimes l$, we have to keep in the type environment information about the occurrence of time region ρ.

Rule (T-ACC)**:** The premise implies that M evaluates to a resource and that resources are accessed according to Γ during the evaluation. $\Gamma;\overline{\rho}$ indicates that after resources are accessed according to Γ, a time period of region ρ arrives so that the delayed access represented by $\Diamond_\rho l$ occurs in that period.

Rule (T-FIX)**:** The premise implies that each time the function body M is evaluated, resources are accessed according to $\Gamma \otimes \varphi$. Since the function may be called many times, resources may be accessed according to $!\Gamma$ in total. $\Gamma^{\Downarrow}$ means that $U^{\Downarrow}$ holds for the usage U obtained by replacing each occurrence of $x : \tau$ in Γ with $\mathit{Use}(\tau)$, where $\mathit{Use}(\mathbf{bool}) = \mathit{Use}(\tau_1 \xrightarrow{\varphi} \tau_2) = \mathbf{0}$ and $\mathit{Use}((\mathbf{R}, U)) = U$. It ensures that Γ expresses only delayed access (since the resources are accessed only when the function is called).

Rule (T-APP)**:** $\Diamond_\rho \tau$ is defined to be $(\mathbf{R}, \Diamond_\rho U)$ if $\tau = (\mathbf{R}, U)$ and $\Diamond_\rho \tau = \tau$ otherwise. $\overline{\rho}^\tau$ represents $\overline{\rho}$ if τ is a resource type and $\mathbf{0}$ otherwise. The premises imply that resources are accessed according to Γ_1 and Γ_2 when M and N are evaluated. The type environment in the conclusion takes into account the evaluation order that M is first evaluated, N is evaluated next, and then the function is called. $\Diamond_\rho$ in the right premise is necessary to capture the fact that the value of N is used according to τ_2 only later when the function is called.

Rule (T-REG)**:** This rule allows us to make a time region invisible from the outside. The condition $\rho \notin \mathbf{Fregs}(\tau) \cup \mathbf{IFregs}(\Gamma)$ makes sure that time periods of the region ρ appears only when the expression M is evaluated. This rule is necessary to keep time regions distinct as far as possible. For example, consider the case where a term $\lambda x.\mathbf{close}(x)$ is applied to two resources y and z. Without using (T-REG), we obtain:

$$\dfrac{\dfrac{\cdots}{x : (\mathbf{R}, \Diamond_\rho C)\ ;\ \overline{\rho} \triangleright \mathbf{close}(x) : \mathbf{bool}}}{\mathbf{0} \triangleright \lambda x.\mathbf{close}(x) : (\mathbf{R}, \Diamond_\rho C) \xrightarrow{\overline{\rho}} \mathbf{bool}}$$

Since the function type mentions region ρ explicitly, both of the usages of y and z must be $\Diamond_\rho C$, so that we cannot distinguish between the timing when y is closed and the timing when z is closed. If we use (T-REG), we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{x : (\mathbf{R}, \Diamond_\rho C)\ ;\ \overline{\rho} \triangleright \mathbf{close}(x) : \mathbf{bool}}}{\exists \rho.(x : (\mathbf{R}, \Diamond_\rho C)\ ;\ \overline{\rho}) \triangleright \mathbf{close}(x) : \mathbf{bool}}\ (\text{T-REG})}{x : (\mathbf{R}, C) \triangleright \mathbf{close}(x) : \mathbf{bool}}\ (\text{T-SUB})}{\mathbf{0} \triangleright \lambda x.\mathbf{close}(x) : (\mathbf{R}, C) \xrightarrow{\mathbf{0}} \mathbf{bool}}\ (\text{T-FIX})$$

So, we can avoid merging the time regions in which function arguments are accessed. (T-REG) is similar to the rule for the letregion construct in ordinary region-effect systems [29].

We give some examples of type derivations below. We often use the fact $\Diamond_\rho \Gamma \otimes \overline{\rho} \leq \Gamma$.

Example 3. The following is a derivation for $(\lambda x.N)M$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, x : \tau_1 \triangleright N : \tau_2 \quad \Gamma_2 \triangleright M : \Diamond_{\rho_2}\tau_1}{\cdots}}{(!\Diamond_\rho \Gamma_1)\ ;\ \Gamma_2\ ;\ (\overline{\rho_2}^{\tau_1} \otimes \overline{\rho}) \triangleright (\lambda x.N)M : \tau_2}}{\exists \rho.(!\Diamond_\rho \Gamma_1\ ;\ \Gamma_2\ ;\ (\overline{\rho_2}^{\tau_1} \otimes \overline{\rho})) \triangleright (\lambda x.N)M : \tau_2}\ (\text{T-REG})}{\Gamma_2\ ;\ (\overline{\rho_2}^{\tau_1} \otimes \Gamma_1) \triangleright (\lambda x.N)M : \tau_2}\ (\text{T-SUB})$$

So, we have the following derived rule for let-expressions:

$$\dfrac{\Gamma_1 \triangleright M : \Diamond_\rho \tau_1 \qquad \Gamma_2, x : \tau_1 \triangleright N : \tau_2}{\Gamma_1; (\overline{\rho}^{\tau_1} \otimes \Gamma_2) \triangleright \mathbf{let}\ x = M\ \mathbf{in}\ N}\ (\text{T-LET})$$

Example 4. Consider $\mathbf{let}\ x = y\ \mathbf{in}\ \mathbf{read}(x); \mathbf{close}(y)$. We have $x : (\mathbf{R}, R); y : (\mathbf{R}, C) \triangleright \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}$. Let $\Gamma_1 = y : (\mathbf{R}, \Diamond_\rho \Diamond_{\rho_1} R)$ and $\Gamma_2 = \overline{\rho_1}; y : (\mathbf{R}, C)$. Then, we can derive $y : (\mathbf{R}, R; C) \triangleright \mathbf{let}\ x = y\ \mathbf{in}\ \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}$ using the derivation in Figure 6. On the other hand, Igarashi and Kobayashi's type system [17] cannot keep track of the order between accesses through different variables, and hence it cannot guarantee that y is read before being closed. Being able to analyze this kind of program is necessary for analyzing object initialization in JVML [14].

Example 5. Example B in Figure 1 is well-typed under $\mathbf{0}$. A part of the derivation is is shown in Figure 6.

Example 6. Let us consider the following expression M:

$$M = \mathbf{fix}(f, y, \mathbf{read}(x); \mathbf{write}(x); f(\mathbf{true}))(\mathbf{true}).$$

M repeatedly reads and writes file x. M is typed as shown in Figure 6. The usage of x is reduced as follows:

$$\begin{array}{ll} & \exists \rho.(!\Diamond_\rho(R; W; \overline{\rho}); \overline{\rho}) \\ \preceq & \exists \rho.(!\Diamond_\rho(R; W; \overline{\rho}) \otimes \overline{\rho}) \\ \preceq & \exists \rho.(\Diamond_\rho(R; W; \overline{\rho}) \otimes !\Diamond_\rho(R; W; \overline{\rho}) \otimes \overline{\rho}) \\ \preceq & \exists \rho.(!\Diamond_\rho(R; W; \overline{\rho}) \otimes (R; W; \overline{\rho})) \\ \xrightarrow{R} & \exists \rho.(!\Diamond_\rho(R; W; \overline{\rho}) \otimes (W; \overline{\rho})) \\ \xrightarrow{W} & \exists \rho.(!\Diamond_\rho(R; W; \overline{\rho}) \otimes \overline{\rho}) \\ \xrightarrow{R} & \cdots \end{array}$$

Therefore, we know that the usage of x denotes the access sequence $RWRWRW\cdots$. (As a by-product, since the usage of x denotess only this infinite sequence, we know that the program does not terminate.)

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

$$\dfrac{c \in \{\mathbf{true}, \mathbf{false}\}}{\mathbf{0} \triangleright c : \mathbf{bool}} \qquad (\text{T-Const})$$

$$\dfrac{[\![\, U \,]\!] \subseteq \Phi \qquad U^{\Downarrow}}{\mathit{Effect}(U) \triangleright \mathbf{new}^{\Phi}() : (\mathbf{R}, U)} \qquad (\text{T-New})$$

$$\dfrac{\Gamma \triangleright M : (\mathbf{R}, \Diamond_{\rho} l)}{\Gamma ; \overline{\rho} \triangleright \mathbf{acc}_l(M) : \mathbf{bool}} \qquad (\text{T-Acc})$$

$$\dfrac{\Gamma \otimes \varphi, f : \tau_1 \xrightarrow{\varphi} \tau_2, x : \tau_1 \triangleright M : \tau_2 \qquad \Gamma^{\Downarrow}}{!\Gamma \triangleright \mathbf{fix}(f, x, M) : \tau_1 \xrightarrow{\varphi} \tau_2} \qquad (\text{T-Fix})$$

$$\dfrac{\Gamma_1 \triangleright M : \tau_1 \xrightarrow{\varphi} \tau_2 \qquad \Gamma_2 \triangleright N : \Diamond_{\rho} \tau_1}{\Gamma_1 ; \Gamma_2 ; (\overline{\rho}^{\tau_1} \otimes \varphi) \triangleright MN : \tau_2} \qquad (\text{T-App})$$

$$\dfrac{\Gamma_1 \triangleright M_1 : \mathbf{bool} \qquad \Gamma_2 \triangleright M_2 : \tau \qquad \Gamma_2 \triangleright M_3 : \tau}{\Gamma_1 ; \Gamma_2 \triangleright \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : \tau} \qquad (\text{T-If})$$

$$\dfrac{\Gamma \triangleright M : \tau \qquad \rho \notin \mathbf{Fregs}(\tau) \cup \mathbf{IFregs}(\Gamma)}{\exists \rho. \Gamma \triangleright M : \tau} \qquad (\text{T-Reg})$$

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

Figure 5: Typing rules

Example 4

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \triangleright y : (\mathbf{R}, \Diamond_{\rho} \Diamond_{\rho_1} R) \qquad \dfrac{x : (\mathbf{R}, R); y : (\mathbf{R}, C) \triangleright \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}}{\Gamma_2, x : (\mathbf{R}, \Diamond_{\rho_1} R) \triangleright \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}}\ (\text{T-Sub})}{\Gamma_1 ; (\overline{\rho} \otimes \Gamma_2) \triangleright \mathbf{let}\ x = y\ \mathbf{in}\ \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}}\ (\text{T-Let})}{\exists \rho. \exists \rho_1. (\Gamma_1 ; (\overline{\rho} \otimes \Gamma_2)) \triangleright \mathbf{let}\ x = y\ \mathbf{in}\ \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}}\ (\text{T-Reg})}{y : (\mathbf{R}, R; C) \triangleright \mathbf{let}\ x = y\ \mathbf{in}\ \mathbf{read}(x); \mathbf{close}(y) : \mathbf{bool}}\ (\text{T-Sub})$$

Example 5

$$\dfrac{\dfrac{\dfrac{\cdots}{!x : (\mathbf{R}, \Diamond_{\rho} C) \triangleright \lambda y.\mathbf{close}(x) : \mathbf{bool} \xrightarrow{\overline{\rho}} \mathbf{bool}} \qquad \dfrac{\cdots}{x : (\mathbf{R}, R); \mathit{finish} : \mathbf{bool} \xrightarrow{\overline{\rho}} \mathbf{bool}; \overline{\rho} \triangleright \mathbf{read}(x); \mathit{finish}(\mathbf{true}) : \mathbf{bool}}}{\exists \rho. (x : (\mathbf{R}, !\Diamond_{\rho} C); x : (\mathbf{R}, R); \overline{\rho}) \triangleright \mathbf{let}\ \mathit{finish} = \lambda y.\mathbf{close}(x)\ \mathbf{in}\ (\mathbf{read}(x); \mathit{finish}(\mathbf{true})) : \mathbf{bool}}}{x : (\mathbf{R}, R\,; C) \triangleright \mathbf{let}\ \mathit{finish} = \lambda y.\mathbf{close}(x)\ \mathbf{in}\ (\mathbf{read}(x); \mathit{finish}(\mathbf{true})) : \mathbf{bool}}$$

Example 6

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{x : (\mathbf{File}, R; W); f : \mathbf{bool} \xrightarrow{\overline{\rho}} \mathbf{bool}; (\overline{\rho} \otimes y : \mathbf{bool}) \triangleright \mathbf{read}(x); \mathbf{write}(x); f(\mathbf{true}) : \mathbf{bool}}}{x : (\mathbf{File}, \Diamond_{\rho}(R; W; \overline{\rho})) \otimes \overline{\rho}, f : \mathbf{bool} \xrightarrow{\overline{\rho}} \mathbf{bool}, y : \mathbf{bool} \triangleright \mathbf{read}(x); \mathbf{write}(x); f(\mathbf{true}) : \mathbf{bool}}\ (\text{T-Sub})}{x : (\mathbf{File}, !\Diamond_{\rho}(R; W; \overline{\rho})) \triangleright \mathbf{fix}(f, y, \mathbf{read}(x); \mathbf{write}(x); f(\mathbf{true})) : \mathbf{bool} \xrightarrow{\overline{\rho}} \mathbf{bool}}\ (\text{T-Fix}) \qquad \mathbf{0} \triangleright \mathbf{true} : \mathbf{bool}}{x : (\mathbf{File}, !\Diamond_{\rho}(R; W; \overline{\rho})); \mathbf{0}; \overline{\rho} \triangleright M : \mathbf{bool}}}{x : (\mathbf{File}, \exists \rho. (!\Diamond_{\rho}(R; W; \overline{\rho}); \overline{\rho})) \triangleright M : \mathbf{bool}}\ (\text{T-Reg})$$

Figure 6: Type derivation trees for Examples 4–6

3.5 Type Soundness

Our type system is sound in the sense that no well-typed program causes a resource access error. Because of the space restriction, we show only main theorems. Proofs are given in the full version of this paper.

In order to prove the soundness, we introduce a type judgment relation $\Gamma \triangleright D : \tau$ for dynamic expressions. Typing rules are given in Appendix A.

We write $\tau^{\downarrow}$ if τ is **bool**, a function type, or a resource type $(\mathbf{R}, \mathbf{0})$. The following theorem says that a well-typed program does not cause a resource access error immediately.

THEOREM 1 (LACK OF IMMEDIATE ERROR). *(i) If* $\varphi \triangleright D : \tau$, *then* $D \not\rightsquigarrow \mathbf{Error}$.

(ii) Suppose $\varphi \triangleright \mathbf{let}\ H\ \mathbf{in}\ v : \tau$. *If* $\tau^{\downarrow}$ *and* $H(x) = \mathbf{new}^{\Phi}()$, *then* $\downarrow \in \Phi$.

The following theorem guarantees that well-typedness is preserved during reduction.

THEOREM 2 (SUBJECT REDUCTION). *If* $\varphi \triangleright D : \tau$ *and* $D \rightsquigarrow D'$, *then* $\varphi \triangleright D' : \tau$.

Using the above theorems, we obtain the following main theorem.

THEOREM 3 (TYPE SOUNDNESS). *If* $\varphi \triangleright M : \tau$, *then the following conditions hold.*

(i) $\mathbf{let}\ \epsilon\ \mathbf{in}\ M \not\rightsquigarrow^{*} \mathbf{Error}$.

(ii) Suppose $\mathbf{let}\ \epsilon\ \mathbf{in}\ M \rightsquigarrow^{*} \mathbf{let}\ H\ \mathbf{in}\ v$. *If* $\tau^{\downarrow}$ *and* $H(x) = \mathbf{new}^{\Phi}()$, *then* $\downarrow \in \Phi$.

Property (ii) means that when a well-typed program terminates, all necessary operations have been performed to each resource. For example, if a resource with the access policy $(R^{*}C\downarrow)^{\sharp}$ is created, it is guaranteed that the resource has been closed when the program terminates.

4. RELATED WORK

Our type system is a refinement of Igarashi and Kobayashi's type system for resource usage analysis [17]: We have replaced $\Diamond U$, which describes a resource that is accessed according to U *at some time* in the future, with $\Diamond_r U$, which carries more precise information that the resource is accessed according to U at some time *in region* r. This refinement is necessary for analyzing closures containing resources (as in Example B in Figure 1) and pointers (though we have not yet extended the type system to deal with pointers). We have looked into the source code of a file synchronizer Unison [25] and found several places where resources are stored in closures as in Example B in Figure 1. Also, our type system seems to be precise enough to analyze object initialization in JVML [14], while Igarashi and Kobayashi's type system is not (recall Example 4).

Another advantage of our new type system is that while the typing rules of our type system are more complex, the proof of type soundness is rather simpler than that for Igarashi and Kobayashi's type system [17]: This is because Igarashi and Kobayashi's type system is highly dependent on the syntactic structure of programs (for example, $(\lambda x.N)M$ and $\mathbf{let}\ x = M\ \mathbf{in}\ N$ cannot be identified even without let-polymorphism), and as a result, a special, complex operational semantics has to be defined to prove subject reduction.

Foster et al.'s type system [13] and DeLine and Fähndrich's type system [7] for Vault can keep track of the state of each resource and ensure that only valid operations are applied at each state. Foster et al.'s type system [13] can be basically considered variants of the effect-based approach explained in Section 1. Thus, as discussed in Section 1, the state of resources cannot be kept track of when they are aliased to the same region. To partially solve this problem, a special programming construct "restrict" has been introduced [13]. For example, the **restrict** construct is used in this way:

$$\mathbf{restrict}\ l = hd(\mathit{locklist})\ \mathbf{in}\ (\mathit{lock}(l); \mathit{unlock}(l))$$

Here, *locklist* is a list of locks. Without the **restrict** construct, their analysis cannot verify the above code since all the elements of the list are aliased to the same region in their analysis. To check the above code, their type system temporarily allocates a fresh region for l to keep track of l's state, and checks that the region does not escape to the outside. So, the underlying mechanism for the rule for the **restrict** construct is similar to that of the rule (T-REG) in our type system. In our type system, the above program can be treated as an ordinary let-expression (which is again a derived form) $\mathbf{let}\ l = hd(\mathit{locklist})\ \mathbf{in}\ (\mathit{lock}(l); \mathit{unlock}(l))$. The type $(\mathbf{Lock}, (\mathit{Lock}; \mathit{Unlock})\ \&\ \mathbf{0})\ \mathbf{list}$ would be assigned to *locklist* (recall that $U_1\ \&\ U_2$ is a derived form: see Section 3.1). In Vault [7], *keys* roughly correspond to regions. Vault avoids the aliasing of multiple resources to the same key using existential types, and can therefore associate a different key with each element of the above list. A construct similar to **restrict** (called "focus") is, however, necessary to access an element of the list. To make a clearer and fairer comparison, we need to extend our target language with imperative features.

Another difference between our type system and the type systems in [13, 7, 8] is that our type system keeps track of resource access sequences while the others keep track of resource states. Although both approaches are almost dual, our approach is slightly more general: In our approach, we can deal with resources that can have infinite states (like stacks and JVM locks [18]) and 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:

$$\frac{\Gamma_1 \triangleright M_1 : \mathbf{bool} \qquad \Gamma_2 \triangleright M_2 : \tau_2}{\Gamma_1 \otimes \Gamma_2 \triangleright M_1 \,||\, M_2 : \tau_2}$$

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.

$$\frac{\begin{array}{c} x : (\mathbf{File}, R) \triangleright \mathbf{read}(x) : \mathbf{bool} \\ x : (\mathbf{File}, R) \triangleright \mathbf{read}(x) : \mathbf{bool} \end{array}}{x : (\mathbf{File}, R \otimes R) \triangleright \mathbf{read}(x) \,||\, \mathbf{read}(x) : \mathbf{bool}}$$

On the other hand, it is not obvious how to extend the type systems in [13, 7, 8] 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 $\mathbf{read}(x); \mathbf{read}(x)$, the first $\mathbf{read}(x)$ 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 $\mathbf{read}(x) \,||\, \mathbf{read}(x)$, however, $\mathbf{read}(x)$ cannot be typed since we cannot statically tell which $\mathbf{read}(x)$ is executed first. (In Vault [7], it seems possible for a programmer to explicitly split a key for reading the file twice into two keys for reading the file once and distribute them to the two processes. It is, however, the programmer's responsibility to write such operations for correctly splitting keys.)

Many of the ideas of our type system (especially, usages and non-standard representation of type environments in our type system) was inspired from our previous work on type systems for concurrent processes [16, 20, 21, 27]. The notion of time regions in this paper is, however, original and seems useful also for refining the type systems for concurrent processes.

Walker [32] has exploited a different combination of ideas of linear types and region-effect systems. To analyze memory usage, he used regions to abstract memory space and then used linearity to control management of regions. On the other hand, our type system uses both linearity and region-effects to control resource usage: linearity controls how often each access primitive is applied to each resource and (time) region-effects control the timing when order access primitives are applied.

There are other approaches to verification of similar properties of programs, using dataflow analysis [6], model checking, and theorem provers [2, 15, 9]. Advantages of type-based approaches in general seem to be that modular analyses can be performed using standard techniques for type inference, that there is a separation of concerns between what programs are accepted (which is expressed by typing rules) and how programs are analyzed (which is expressed by a type inference algorithm), and that there is a standard technique for proving soundness (using the subject reduction property). On the other hand, a disadvantage is that value-dependent behavior of programs cannot be analyzed in type-based approaches (unless we use a general form of dependent types, in which case automatic analysis is hopeless). Besides this general difference, Das et al.'s method [6] seems to suffer from a problem similar to that of the effect-based approach explained in Section 1; when different resources may flow into the same argument of a resource access primitive, their analysis cannot determine whether the primitive is indeed applied to the resources. Extended Static Checking [9] requires the user's annotation of pre/post conditions of procedures and loop invariants, although some of them can be automatically inferred [12].

5. CONCLUSION

We have developed a new type system for resource usage analysis and proved its soundness. It combines ideas of linear type systems and effect systems, and seems to be more general than previous type systems developed for a similar purpose. We have two kinds of applications of the type system in mind. One is to use our type system to develop a general tool for resource usage analysis that can deal with arbitrary kinds of resources. As we discuss below, we admit that a lot of work is left to be done for this purpose. The other application is to use our type system as a theoretical basis on which various program analysis tools for specific kinds of resources can be developed and compared in a disciplined manner. In fact, we have already developed a clean type system [18] for analysis of JVM lock primitives along this line of research.

It is possible to further refine our type system in several ways. Introducing let-polymorphism seems to be especially effective.

A lot of work is left to be done to develop a general tool for resource usage analysis based on our type system. First of all, we need to develop a concrete algorithm for resource usage analysis (i.e., a type inference algorithm). As briefly discussed in the full version of this paper, a type inference algorithm can be developed in a manner similar to that for Igarashi and Kobayashi's type system (possibly by imposing some restriction), but some sub-algorithms remain left open. In particular, we need to develop an algorithm (that is necessarily incomplete but works well in practice) for checking $[\![\, U \,]\!] \subseteq \Phi$ for an inferred usage U and a resource usage policy Φ. The problem of developing such an algorithm has been left open since Igarashi and Kobayashi's work. For a specific resource usage policy and a restricted form of usages, we have already developed such an algorithm [18].

There seems to be some similarity between our usages and some logics – especially, extensions of linear logic with modal operators [19, 22] and process logics [5]. It would be interesting to study formal connections between them.

Acknowledgment

We would like to thank Tatsuo Mizukami, Eijiro Sumii and anonymous referees for useful comments on an earlier version of this paper.

6. REFERENCES

[1] A. Aiken, M. Fähndrich, and R. Levien. Improving region-based analysis of higher-order languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1995.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

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

[4] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1996.

[5] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[6] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

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

[8] R. DeLine and M. Fähndrich. Adoption and focus:

Practical linear types for imperative programming. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.
[9] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, 1998.
[10] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.
[11] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP 1999*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, 1999.
[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 191–202, 2002.
[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.
[14] S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 58–70, 2002.
[16] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 128–141, January 2001.
[17] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.
[18] F. Iwama and N. Kobayashi. A new type system for JVM lock primitives. In *Proceedings of ASIA-PEPM'02*. ACM Press, 2002. Available at `http://www.kb.cs.titech.ac.jp/~kobayasi/publications.html`.
[19] M. Kanovich and T. Ito. Temporal linear logic specifications for concurrent processes. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 48–57, 1997.
[20] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.
[21] N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, August 2000. The full version is available as technical report TR00-01, Dept. Info. Sci., Univ. Tokyo.
[22] N. Kobayashi, T. Shimizu, and A. Yonezawa. Distributed concurrent linear logic programming. *Theoretical Computer Science*, 227(2):185–220, 1999.
[23] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[24] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
[25] B. C. Pierce, T. Jim, and J. Vouillon. UNISON: A portable, cross-platform file synchronizer, 2001. `http://www.cis.upenn.edu/~bcpierce/unison`.
[26] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
[27] E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, volume 16(3) of *ENTCS*, pages 55–77, 1998.
[28] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
[29] M. Tofte and J.-P. Talpin. 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*, pages 188–201, 1994.
[30] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
[31] D. Walker, K. Crary, and J. G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.
[32] D. Walker and K. Watkins. On linear types and regions. In *Proceedings of International Conference on Functional Programming*, 2001.

APPENDIX

A. TYPING RULES FOR DYNAMIC EXPRESSIONS

We write $\Gamma \setminus x$ for the type environment obtained by replacing each occurrence of $x : \tau$ in Γ with $\mathit{Effect}(\tau)$, where $\mathit{Effect}(\tau)$ is defined by: $\mathit{Effect}(\mathbf{bool}) = \mathit{Effect}(\tau_1 \xrightarrow{\varphi} \tau_2) = \mathbf{0}$ and $\mathit{Effect}(\mathbf{R}, U) = \mathit{Effect}(U)$.

$$\frac{\Gamma \triangleright M : \tau}{\Gamma \triangleright \mathbf{let}\ \epsilon\ \mathbf{in}\ M : \tau} \quad \text{(DT-EMPTY)}$$

$$\frac{\Gamma \triangleright \mathbf{let}\ H\ \mathbf{in}\ M : \tau \qquad \Gamma(x) = (\mathbf{R}, U) \qquad [\![U]\!] \subseteq \Phi}{\Gamma \setminus x \triangleright \mathbf{let}\ x = \mathbf{new}^{\Phi}(); H\ \mathbf{in}\ M : \tau} \quad \text{(DT-ALLOCR)}$$

$$\frac{\begin{array}{c}\Gamma_1 \triangleright \mathbf{fix}(f, y, M) : \tau_1 \xrightarrow{\varphi} \tau_2 \\ \Gamma_2, x : \tau_1 \xrightarrow{\varphi} \tau_2 \triangleright \mathbf{let}\ H\ \mathbf{in}\ N : \tau\end{array}}{\Gamma_1 \otimes \Gamma_2 \triangleright \mathbf{let}\ x = \mathbf{fix}(f, y, M); H\ \mathbf{in}\ N : \tau} \quad \text{(DT-ALLOCF)}$$

$$\frac{\Gamma \triangleright D : \tau \qquad \rho \notin \mathbf{Fregs}(\tau) \cup \mathbf{IFregs}(\Gamma)}{\exists \rho . \Gamma \triangleright D : \tau} \quad \text{(DT-REG)}$$

$$\frac{\Gamma' \triangleright D : \tau \qquad \Gamma \leq \Gamma'}{\Gamma \triangleright D : \tau} \quad \text{(DT-SUB)}$$