Resource Usage Analysis for a Functional Language with Exceptions

Futoshi Iwama
Tohoku University
iwama@kb.ecei.tohoku.ac.jp

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

Abstract

Igarashi and Kobayashi have proposed a general type system for checking whether resources such as files and memory are accessed in a valid manner. Their type system is, however, for call-by-value λ-calculus with resource primitives, and does not deal with non-functional primitives such as exceptions and pointers. We extend their type system to deal with exception primitives and prove soundness of the type system. Dealing with exception primitives is especially important in practice, since many resource access primitives may raise exceptions. The extension is non-trivial: While Igarashi and Kobayashi's type system is based on linear types, our new type system is a combination of linear types and effect systems. We also report on a prototype analyzer based on the new type system.

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 Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Verification.

Keywords Effect System, Exception, Resource Usage Analysis, Type Inference, Type System.

1. Introduction

Background There has recently been growing interest in methods for verifying that resources (such as files, memories or network channels) used in a program are accessed in a valid manner [1, 3, 8, 10, 22, 24]. For example, Tofte et al.'s type system [22] can ensure that certain memories are not accessed after their deallocation and Freund and Mitchell's type system [8] can ensure that objects are used only after their initialization is finished. Igarashi and Kobayashi [10] have coined the term "resource usage analysis" for such general verification problems and proposed a type-based method for resource usage analysis for call-by-value λ-calculus with resource primitives.

The idea of their type system is to augment types of resources with information about in which order resources are accessed. For example, the type of read-only files is given by $(\mathbf{File}, R^*; C)$ and the type of read-write files is given by $(\mathbf{File}, (R + W)^*; C)$, where the order of operations on files is represented by regular expressions (the concatenation is given by ';') with R, W and C as labels for read, write and close operations, respectively (in their type system, actually, a more expressive language called *usage expressions* is used to represent the access order).

Typing rules are designed by taking the evaluation order into account. For example, the usual typing rule for **let** expressions:

$$\frac{\Gamma \vdash M : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash N : \sigma_2}{\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma_2}$$

is replaced by:

$$\frac{\Gamma \vdash M : \sigma_1 \qquad \Delta, x : \sigma_1 \vdash N : \sigma_2}{\Gamma; \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma_2}$$

Here, $\Gamma; \Delta$ denotes the type environment obtained by composing the usage expression of Γ and Δ by ';'. For example, if $\Gamma = x : (\mathbf{File}, R^*)$, which intuitively means the resource x is read several times during the evaluation of M, and if $\Delta = x : (\mathbf{File}, C)$, which means x is closed (once) during the evaluation of N, then $\Gamma; \Delta = x : (\mathbf{File}, R^*; C)$, which means x is read several times *and then* closed. The new typing rule reflects the fact that M is evaluated first and then N is in $\mathbf{let}\ x = M\ \mathbf{in}\ N$.

In this way, Igarashi and Kobayashi's type system can keep track of the order of accesses to resources. However, the target language is pure call-by-value λ-calculus only with resource primitives. So, it is not clear that the method can be extended to practical programming languages, which are usually equipped with non-functional primitives such as exceptions and references.

Our Contributions In this paper, we extend Igarashi and Kobayashi's type system to deal with exception primitives. This extension is very important in practice because access primitives in practical programming languages may raise exceptions such as *End_of_File*. Our technical contributions are summarized as follows:

- Design and formalization of a type system for resource usage analysis for a functional language with an exception handling mechanism;
- Proof of soundness of the type system;
- Development of a type reconstruction algorithm; and
- Implementation of a prototype resource usage analyzer.

PEPM '06 2006, Charleston, South Carolina, USA.

Although the exception mechanism here is much simpler than that of ML or Java (there is only one exception constructor, which does not carry values), the extension is already non-trivial. While the type system of Igarashi and Kobayashi [10] is based on linear types, our new type system is based on a combination of linear type and effect systems [21]. In fact, even for the problem of just checking that certain values are used only once (which can be considered an instance of the resource usage analysis problem), previous linear type systems [10, 13, 16, 23] are insufficient; For example, they cannot judge that x is used once in the following program (suppose that `use` is a function that uses its argument just once):

```
let   f () = use x
in    try (if b then f() else raise E)
      with E -> use x
```

Key Ideas in the Type System First, we keep track of information on exceptions, as well as that on access sequences, by usage expressions. For this purpose, we extend usage expressions by introducing the special label E and the constructor $;_E$. E means that a resource is not accessed any more because an exception is raised; for example, the usage $R; E$ means a resource is first read and then an exception is raised. The usage $U_1;_E U_2$, which corresponds to exception handling, means that a resource is used according to U_1 and if an exception is raised, then it is used according to U_2. Thus, for example, the usage $(R; E);_E C$ is equivalent to $R; C$. Now, for example, `try (read(x);raise) with E -> close(x)` is typed as follows:

$$x : (\mathbf{File}, (R; E);_E C) \vdash \begin{array}{ll}\texttt{try} & \texttt{read(x);raise} \\ \texttt{with} & \texttt{E -> close(x)}\end{array} : \mathbf{bool}.$$

Notice that $;_E$ corresponds to the use of `try`. The type judgment above tells us that, during the evaluation of M, the file x is first read and then closed. (Here, we assume access primitives themselves do not raise exceptions; we can model access primitives that may raise exceptions by combining them with conditional expressions.)

Unfortunately, adding the above usage constructors is not sufficient to obtain the accuracy required for practical programs using exceptions. For example, consider the following program M:

$$\mathbf{let}\ f = \lambda x.\mathbf{raise}\ E\ \mathbf{in}\ \mathbf{try}\ \mathbf{read}(x); f()\ \mathbf{with}\ E \to \mathbf{close}(x)$$

If we naively extend Igarashi and Kobayashi's type system with the new usage constructors above, we would obtain the following judgment:

$$x : (\mathbf{File}, \Diamond E; (R;_E C)) \vdash M : \mathbf{bool}$$

Here, $\Diamond E$ means that an exception may be raised once at *any* time; so the judgment does not tell us that an exception is raised inside the body of the try-expression. To solve this problem, we exploit the idea of effect systems [21] to keep track of more accurate information on *when* exceptions are raised. As a result, a type judgment becomes of the form:

$$\Gamma \parallel \varphi \vdash M : \sigma,$$

which means that, during the evaluation of term M, each resource is accessed according to (usages in) Γ and exceptions are raised according to effect φ.

Rest of This Paper In Section 2, we introduce our target language formally. In Section 3, we present our type system for resource usage analysis. Section 4 shows the correctness of the type system. Section 5 describes a type inference algorithm. Section 6 reports on implementation and experiments. We discuss other possible methods for dealing with the exception mechanism in Section 7. After discussing related work in Section 8, we conclude in Section 9.

2. Target Language $\lambda^{\mathcal{R}}_{E}$

This section introduces the target language $\lambda^{\mathcal{R}}_{E}$ of our analysis. $\lambda^{\mathcal{R}}_{E}$ is a call-by-value λ-calculus extended with resources and exceptions.

We assume a finite set $\mathcal{A}$ of *access-labels*, ranged over by a. An access label denotes the kind of access to a resource; We shall use access labels I, R, W and C for initialization, read, write, and close operations respectively.

A *trace* is a sequence consisting of access labels and the special label $\downarrow$. Formally, the set $\mathcal{A}^{*,\downarrow}$ of traces is defined by:

$$\mathcal{A}^{*,\downarrow} = \mathcal{A}^* \cup \{s \downarrow \mid s \in \mathcal{A}^*\}$$

Here, $\mathcal{A}^*$ is the set of finite sequences of elements of $\mathcal{A}$. The additional symbol $\downarrow$ expresses that the evaluation terminates normally or abruptly with an exception.

A trace expresses how a resource has been accessed at a certain point of the execution of a program. A trace $a_1 a_2 \ldots a_k$ means that the resource is accessed by each operation a_i in the order $a_1, a_2, \ldots, a_k$. A trace $a_1 a_2 \ldots a_k \downarrow$ means that the evaluation has terminated after the resource is accessed in the order $a_1, a_2, \ldots, a_k$. For example, a trace $RRC\downarrow$ means that the resource has been read twice, closed, and then the evaluation has terminated.

A *trace set* is a set of traces that is closed under the prefix operation. We write $S^{\#}$ for the set of prefixes of elements of S. A set S of traces is called a *trace set* if $S^{\#} = S$. We use the metavariable Φ for a trace set.

Consider a regular expression $(RW\downarrow)$, then $(RW\downarrow)^{\#} = \{\epsilon, R, RW, RW\downarrow\}$ is a trace set. We can consider a trace set Φ as a specification of each resource, which requires the resource is accessed only according to a trace in the set Φ.

Example 2.1. Let us consider the trace set $\Phi = (IR^*C\downarrow)^{\#}$ and program: $\mathbf{init}(x); \mathbf{read}(x); \mathbf{close}(x)$. The program initializes, reads and closes the resource x. This program satisfies the specification Φ. On the other hand, neither $\mathbf{read}(x); \mathbf{close}(x)$ nor $\mathbf{init}(x); \mathbf{read}(x)$ satisfies Φ, since the resource x can be accessed according to $RC\downarrow$ or $IR\downarrow$ but neither trace is contained in Φ. The meaning of $\downarrow$ is a little tricky; Let us consider the program: $\mathbf{init}(x); \mathbf{read}(x); \mathit{loop\_infinitely}$. It initializes and reads the resource x, and then goes into an infinite loop. This program *does* satisfy the specification Φ. Actually, the resource x is accessed according to a trace in the set $\{IR\}^{\#} \subset \Phi$. □

DEFINITION 2.1 (Terms, Values).

$$\begin{array}{rcl} M\ \textit{(terms)} & ::= & v \mid M_1 M_2 \mid \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \\ & \mid & \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \\ & \mid & \mathbf{new}^{\Phi}() \mid \mathbf{acc}^{a}(M) \mid M^{\{x\}} \\ & \mid & \mathbf{try}\ M_1\ \mathbf{with}\ M_2 \mid \mathbf{raise} \\ v\ \textit{(values)} & ::= & \mathbf{true} \mid \mathbf{false} \mid x \mid \mathbf{fun}(f, x, M) \end{array}$$

The first two lines show standard constructs for the λ-calculus. $\mathbf{fun}(f, x, M)$ is a recursive function (which is often defined by $f(x) = M$). We write $\lambda x.M$ when f is not free in M. The primitives for resources are the same as those of $\lambda^{\mathcal{R}}$ [10]. $\mathbf{new}^{\Phi}()$ is the primitive for creating a resource. The trace set Φ specifies how the resource should be accessed afterwards. In this paper, we often use a regular expression to specify a trace set. $\mathbf{acc}^{a}(M)$ is the primitive for accessing the resource M with an operation specified by a. We often write $\mathbf{init}(M), \mathbf{read}(M), \mathbf{write}(M)$ and $\mathbf{close}(M)$ for $\mathbf{acc}^{I}(M), \mathbf{acc}^{R}(M), \mathbf{acc}^{W}(M)$ and $\mathbf{acc}^{C}(M)$, respectively. For the sake of simplicity, we assume that the resource access primitive returns a boolean in a non-deterministic manner.

$M^{\{x\}}$ is the same as M, except that the evaluation gets stuck if the resource bound to x escapes from M. The escape information

is used to refine the accuracy of the analysis. A separate escape analysis is assumed, which checks that x does not escape from M in $M^{\{x\}}$. The term $\mathbf{try}\ M_1\ \mathbf{with}\ M_2$ first evaluates M_1. If an exception is raised, then M_2 is evaluated; Otherwise, the value of M_1 is returned. The term $\mathbf{raise}$ raises an exception. For the sake of simplicity, we consider a single kind of exception, so $\mathbf{raise}$ takes no argument. We write $M_1; M_2$ for $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ if variable x is not free in M_2.

Example 2.2. Consider the following terms:

$$M_1 \triangleq \mathbf{if}\ \mathbf{init}(x)^{\{x\}}\ \mathbf{then}\ (\mathbf{write}(x)^{\{x\}}; \mathbf{close}(x)^{\{x\}})\ \mathbf{else}\ \mathbf{raise}$$
$$M_2 \triangleq \mathbf{try}\ M_1\ \mathbf{with}\ \mathbf{close}(x)^{\{x\}}$$
$$M \triangleq \mathbf{let}\ x = \mathbf{new}^{(I(W)^*C\downarrow)^{\#}}()\ \mathbf{in}\ M_2$$

M_1 first initializes x. If the initialization returns $\mathbf{true}$, then it writes and closes x; otherwise an exception is raised. The term M_2 closes x when an exception is raised by M_1. Therefore, the resource x is closed no matter whether $\mathbf{init}(x)$ returns $\mathbf{true}$ or $\mathbf{false}$. The term M creates a resource x and evaluates M_2. The trace set $(I(W)^*C\ \downarrow)^{\#}$ specifies that x should be first initialized, that it can be written an arbitrary number of times after that, and that it must be closed once before the program terminates. The term M_2 obeys that specification, so that M should be accepted as a good program. □

Example 2.3. The following is a fragment of a typical OCaml program accessing files.

```
try  while (true)
       do  write_char(y,read_char(x))  done
with  End_of_File -> close(x);close(y)
```

It copies a character from x to y until the end-of-file exception is raised, when the exception handler is executed and x and y are closed.

The above program can be modeled in our language as the following term M.

$$M \triangleq \mathbf{try}\ \mathbf{fun}(f, z, M_1)\mathbf{true}\ \mathbf{with}\ (\mathbf{close}(x); \mathbf{close}(y))$$
$$M_1 \triangleq (\mathbf{if}\ \mathbf{read}(x)\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{raise}); \mathbf{write}(y); f\ \mathbf{true}$$

Note that the library function call `read_char(x)`, which may raise an exception, has been modeled by $\mathbf{if}\ \mathbf{read}(x)\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{raise}$. The value returned by the function `read_char` is ignored in the translation, since we are only concerned with the order of resource accesses. □

2.1 Operational semantics of $\lambda_E^{\mathcal{R}}$

An operational semantics of $\lambda_E^{\mathcal{R}}$ is given by reduction of pairs of a term and a *heap*, used to record the states of resources. The state of a resource only captures what access sequence is allowed for the resource; Resource-specific values such as the contents of a file are not modeled.

DEFINITION 2.2 (Heap). *A* heap H *is a finite mapping from variables to trace sets.*

We write $\{x_1 \mapsto \Phi_1, \ldots, x_n \mapsto \Phi_n\}$ $(n \geq 0)$ for the heap H such that $dom(H) = \{x_1, \ldots, x_n\}$ and $H(x_i) = \Phi_i$. It expresses that each variable x_i refers to a resource that should be used according to one of the traces of Φ_i. When $dom(H_1) \cap dom(H_2) = \emptyset$, we write $H_1 \uplus H_2$ for the heap H such that $dom(H) = dom(H_1) \cup dom(H_2)$ and $H(x) = H_i(x)$ for x in $dom(H_i)$.

DEFINITION 2.3 (Evaluation Contexts). Evaluation contexts, *ranged over by* $\mathcal{E}$, *and* evaluation contexts without **try**, *ranged over by* $\mathcal{E}^{\overline{try}}$, *are defined by the following syntax:*

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

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

$\mathcal{E}$ is an ordinary call-by-value evaluation context; $\mathcal{E}^{\overline{try}}$ is one without exception handlers and used to identify an innermost exception handler. We write $\mathcal{E}[M]$ and $\mathcal{E}^{\overline{try}}[M]$ for the expressions obtained by replacing $[\,]$ in $\mathcal{E}$ and $\mathcal{E}^{\overline{try}}$ respectively with M.

We write $[M_1/x_1, \ldots, M_n/x_n]$ for the capture-avoiding simultaneous substitution of M_i for x_i and Φ^{-a} for $\{s \mid as \in \Phi\}$. $\mathbf{FV}(M)$ denotes the set of free variables in M.

DEFINITION 2.4 (Reduction Relation). *The relation* $(H, M) \rightsquigarrow P$, *where* P *is either a pair* (H', M') *or* **Error**, *is the least relation closed under the rules in Figure 1. We write* $\rightsquigarrow^*$ *for the reflexive and transitive closure of* $\rightsquigarrow$.

The rule R-NEW is for fresh resource allocation. The rules R-ACC and R-ACCERR express an access to a resource. The rule R-ACC is for successful resource access: the trace set Φ^{-a} after the access is obtained by removing the label a at the head of a trace (if the trace begins with a; the traces not beginning with a are discarded). If no trace begins with a (i.e., $\Phi^{-a} = \emptyset$), then the resource access results in an **Error**. We do not care about the result of resource access here, it is left unspecified which boolean values are returned in R-ACC, so reduction $\rightsquigarrow$ is nondeterministic. The rule R-TRYRAI is for exception handling. A term of the form $\mathbf{try}\ \mathcal{E}^{\overline{try}}[\mathbf{raise}]\ \mathbf{with}\ M$ represents the execution state in which an exception is being raised and the innermost handler is M, and so reduces to M.

Note that by representing the resource states using a heap, we can correctly model the case where resources are aliased (i.e., the case where the same resource is referred to by multiple variables and functions). For example, the term M (where $\Phi = (R^*C\downarrow)^{\sharp}$):

$$\mathbf{let}\ x = \mathbf{new}^{\Phi}()\ \mathbf{in}\ \mathbf{let}\ y = x\ \mathbf{in}\ \mathbf{read}(y); \mathbf{close}(x)$$

is reduced as follows.

$$\begin{array}{l} (\emptyset, M) \\ \rightsquigarrow (\{z \mapsto \Phi\}, \mathbf{let}\ x = z\ \mathbf{in}\ \mathbf{let}\ y = x\ \mathbf{in}\ \mathbf{read}(y); \mathbf{close}(x)) \\ \rightsquigarrow (\{z \mapsto \Phi\}, \mathbf{let}\ y = z\ \mathbf{in}\ \mathbf{read}(y); \mathbf{close}(z)) \\ \rightsquigarrow (\{z \mapsto \Phi\}, \mathbf{read}(z); \mathbf{close}(z)) \\ \rightsquigarrow (\{z \mapsto \Phi\}, \mathbf{close}(z)) \\ \rightsquigarrow (\{z \mapsto \{\epsilon, \downarrow\}\}, \mathbf{true}) \end{array}$$

3. Type System

In this section, we present a type system to guarantee that all accesses to resources in a well-typed term obey the specification (given by the trace set Φ attached to each resource creation $\mathbf{new}^{\Phi}()$).

3.1 Usages

Usage expressions (in short, usages) describe in which order and by which operations a resource can be accessed. As mentioned above, we express information about exceptions also with usages. We therefore add new constructors E and $U_1\ ;_E\ U_2$ to the usages given in [10].

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

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

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

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

$$(H, \mathcal{E}[\mathbf{let}\ x = v\ \mathbf{in}\ M]) \rightsquigarrow (H, \mathcal{E}[[v/x]M]) \quad \text{(R-LET)}$$

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

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

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

$$(H, \mathcal{E}[\mathbf{try}\ v\ \mathbf{with}\ M]) \rightsquigarrow (H, \mathcal{E}[v]) \quad \text{(R-TRY)}$$

$$(H, \mathcal{E}[\mathbf{try}\ \mathcal{E}^{\overline{try}}[\mathbf{raise}]\ \mathbf{with}\ M]) \rightsquigarrow (H, \mathcal{E}[M]) \quad \text{(R-TRYRAI)}$$

Figure 1. Operational semantics of $\lambda_E^{\mathcal{R}}$

Syntax of Usages.

Let the set $\mathcal{L}$ of *labels*, ranged over by l, be $\mathcal{A} \cup \{1, \tau, E\}$. The label 1 is a special label used to count the number of function applications; the labels τ denotes exception handling (which is an unobservable, internal action, hence τ) and E denotes a raised exception respectively.

DEFINITION 3.1 (Usages). *The set $\mathcal{U}$ of* usages*, ranged over by U, is defined by:*

$$\begin{array}{rcl} U & ::= & \mathbf{0} \mid l \mid \alpha \mid U_1; U_2 \mid U_1 \otimes U_2 \mid U_1 \& U_2 \mid \Diamond U \mid \blacklozenge U \\ & \mid & \mu\alpha.U \mid U_1;_E U_2 \end{array}$$

We assume that the unary usage constructors $\Diamond$ and $\blacklozenge$ bind tighter than the binary constructors ($\&, ;, \otimes$ and $;_E$).

We briefly explain informal meaning of usage constructors; see also Igarashi and Kobayashi [10]. The usage $\mathbf{0}$ means that a resource cannot be accessed at all. The usage $l \in \mathcal{L}$ means that a resource is accessed by an access primitive labeled with l (if $l \in \mathcal{A}$), or that an event corresponding to l occurs: especially, $E \in \mathcal{L}$ means that a resource is not accessed later due to a raise of an exception. α is the usage variable, bound in the form of $\mu\alpha.U$, which denotes a recursive usage that satisfies $\alpha = U$. $U_1; U_2$ means that a resource is first accessed according to U_1 and then according to U_2. $U_1 \otimes U_2$ means that a resource is accessed according to a sequence obtained by interleaving U_1 and U_2. $U_1 \& U_2$ means that a resource is accessed according to either U_1 or U_2. $U_1;_E U_2$ means a resource is accessed according to U_1 and, if an exception is raised during the execution, then the resource is accessed according to U_2. For example, $((R \& E); W);_E C$ is equivalent to $(R; W) \& C$. $\Diamond U$ means that some of the resource access expressed by U may be delayed. For example, $(\Diamond l_1); l_2$ expresses access order either $l_1; l_2$ or $l_2; l_1$. $\blacklozenge U$, which cancels $\Diamond$, means that the access represented by U must occur *now*. So, $(\blacklozenge \Diamond U_1); U_2$ is equivalent to $U_1; U_2$.

$$\begin{array}{c} \Diamond \mathbf{0} \equiv \mathbf{0} \quad \blacklozenge \mathbf{0} \equiv \mathbf{0} \\ \mathbf{0} \otimes U \equiv U \quad \mathbf{0}; U \equiv U \quad U; \mathbf{0} \equiv U \quad \mathbf{0};_E U \equiv \mathbf{0} \\ U_1 \otimes U_2 \equiv U_2 \otimes U_1 \quad U_1 \& U_2 \equiv U_2 \& U_1 \\ \Diamond U_1 \otimes \Diamond U_2 \equiv \Diamond(U_1 \otimes U_2) \quad \Diamond U_1;_E U_2 \preceq \Diamond(U_1;_E U_2) \\ \Diamond U_1; U_2 \equiv \Diamond U_1 \otimes U_2 \\ U_1 \& U_2 \preceq U_1 \quad \mu\alpha.U \equiv [\mu\alpha.U/\alpha]U \end{array}$$

Figure 2. Structural pre-congruence on usages

$$l \xrightarrow{l} \mathbf{0} \quad \frac{U \xrightarrow{l} U'}{\Diamond U \xrightarrow{l} \Diamond U'} \quad \frac{U \xrightarrow{l} U'}{\blacklozenge U \xrightarrow{l} \blacklozenge U'}$$

$$\frac{U_1 \xrightarrow{l} U_1'}{U_1 \otimes U_2 \xrightarrow{l} U_1' \otimes U_2} \quad \frac{U_1 \xrightarrow{l} U_1'}{U_1; U_2 \xrightarrow{l} U_1'; U_2}$$

$$\frac{U_1 \xrightarrow{l} U_1' \quad l \neq E}{U_1;_E U_2 \xrightarrow{l} U_1';_E U_2} \quad \frac{U_1 \xrightarrow{E} U'}{U_1;_E U_2 \xrightarrow{\tau} U_2}$$

$$\frac{U_1 \preceq U_1' \quad U_1' \xrightarrow{l} U_2' \quad U_2' \preceq U_2}{U_1 \xrightarrow{l} U_2}$$

Figure 3. Usage transition rules

Example 3.1. The accesses to x in M_2 of Example 2.2 is expressed by the usage $I; ((W; C) \& E);_E C$. Similarly, the accesses to x in M of Example 2.3 is expressed by the usage $\mu\alpha.(R; (\mathbf{0} \& E); \alpha);_E C$. □

In what follows, we write $U \backslash E$ for $U;_E \mathbf{0}$, which cancels exceptions in U. For example, $((l_1; l_2) \otimes E) \backslash E$ is equivalent to $\mathbf{0} \& l_1 \& (l_1; l_2)$.

Semantics of Usages and Subusage Relation.

We give the formal semantics of usages via a labeled transition system. Then, we define the subusage relation $U_1 \leq U_2$, which means that the access order U_1 is more general than U_2.

We define the transition relation of the form $U \xrightarrow{l} U'$, which means that a resource of usage U can be first accessed by l and then accessed according to U'.

DEFINITION 3.2. *The binary relation $U \preceq U'$ on usages are the least pre-congruence that satisfies the rules in Figure 2, where $U \equiv U'$ means $U \preceq U'$ and $U' \preceq U$.*

For example, $(\Diamond U_1 \& U_2); U_3 \preceq \Diamond U_1 \otimes U_3$ holds.

Now, we give the transition rules on usages.

DEFINITION 3.3 (Transition Relation on Usages). *The transition relation $U \xrightarrow{l} U'$ on usages is the least relation closed under the rules in Figure 3. The multi-step transition relation $U \xrightarrow{t} U'$, where $t \in (\mathcal{A} \cup \{1, \tau\})^*$, is defined inductively as follows.*

$$\xrightarrow{t} \stackrel{def}{=} \begin{cases} \preceq & \text{if } t = \epsilon \\ \xrightarrow{l} \xrightarrow{t'} & \text{if } t = lt' \end{cases}$$

Example 3.2. Usage $R;(C\&E);_E C$ has the following transition sequences:

$$(R;(C\&E));_E C \xrightarrow{R} (C\&E);_E C \xrightarrow{C} \mathbf{0}$$
$$(R;(C\&E));_E C \xrightarrow{R} (C\&E);_E C \xrightarrow{\tau} C \xrightarrow{C} \mathbf{0}$$

□

By using the labeled transition system, we define the trace set expressed by a usage.

DEFINITION 3.4. *Let U be a usage. The trace set $[\![U]\!]$ is defined by*

$$[\![U]\!] = \{\hat{t} \mid \exists U'.(U \xrightarrow{t} U')\} \cup \{\hat{t}\downarrow \mid U \xrightarrow{t} \mathbf{0}\} \cup \{\hat{t}\downarrow \mid \exists U'.(U \xrightarrow{t} \xrightarrow{E} U')\}$$

Here, $\hat{t} \in \mathcal{A}^*$ is the label sequence obtained by removing all the occurrences of τ from t.

Example 3.3. $[\![\mu\alpha.\alpha]\!] = \{\epsilon\}$, $[\![\mathbf{0}]\!] = \{\epsilon, \downarrow\}$, $[\![\Diamond l_1; l_2]\!] = \{l_1 l_2\downarrow, l_2 l_1\downarrow\}^{\#}$, $[\![\blacklozenge\Diamond l_1; l_2]\!] = \{l_1 l_2\downarrow\}^{\#}$, $[\![(R\&E); C]\!] = \{RC\downarrow\}^{\#}$ and $[\![((R\&E); C);_E C]\!] = \{RC\downarrow, C\downarrow\}^{\#}$. □

We write $U_1 \Longrightarrow U_2$ when $U_1 \xrightarrow{\tau\cdots\tau} U_2$ (where $\tau\cdots\tau$ is a possibly empty sequence of τ). We also write $\overset{l}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{l}\Longrightarrow$.

The subusage relation $U_1 \leq U_2$ is defined as a weak simulation relation closed under usage contexts. A *usage context*, written $\mathcal{C}$, 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 $\mathcal{C}$. We write $\mathcal{C}[U]$ for the usage obtained by replacing $[\,]$ with U. For example, if $\mathcal{C} = \mu\alpha.([\,]\,;\alpha)$, then $\mathcal{C}[U] = \mu\alpha.(U\,;\alpha)$.

DEFINITION 3.5 (Subusage relation). *$U_1 \leq U_2$ is the largest relation that satisfies the following conditions:*

(1) $\mathcal{C}[U_1] \leq \mathcal{C}[U_2]$ for any usage context $\mathcal{C}$,
(2) If $U_2 \xrightarrow{l} U_2'$ and $l \in \mathcal{A} \cup \{1\}$, then $U_1 \overset{l}{\Longrightarrow} U_1'$ and $U_1' \leq U_2'$ for some U_1',
(3) If $U_2 \xrightarrow{\tau} U_2'$, then $U_1 \Longrightarrow U_1'$ and $U_1' \leq U_2'$ for some U_1',
(4) If $U_2 \xrightarrow{\epsilon} \mathbf{0}$, then $U_1 \Longrightarrow \mathbf{0}$;
(5) If $U_2 \xrightarrow{E} U_2'$, then $U_1 \overset{E}{\Longrightarrow} U_1'$ for some U_1'.

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

Example 3.4. $E;_E C \cong C$ and $(C\&E)\setminus E \cong C\&\mathbf{0}$ hold. □

Note that, if $U_1 \leq U_2$, then $[\![\mathcal{C}[U_1]]\!] \supseteq [\![\mathcal{C}[U_2]]\!]$ for any $\mathcal{C}$—in particular, $[\![U_1]\!] \supseteq [\![U_2]\!]$. Moreover, $U \leq \mu\alpha.\alpha$ and $\Diamond U \leq U$ hold for any usage U. $U_1 \preceq U_2$ implies $U_1 \leq U_2$ and $U_1 \equiv U_2$ implies $U_1 \cong U_2$.

3.2 Effects and Types

We proceed to the definitions of effects and types.

An *effect* expresses the termination behavior of an evaluation. Intuitively, the effect $\mathbf{0}$ means that evaluation can terminate normally; E that evaluation can abort with an exception; $E^?$ that evaluation can terminate normally or abort; and, finally, $\top$ that evaluation cannot terminate.

DEFINITION 3.6 (Effects and Subeffect Relation). *The set of* effects, *ranged over by φ, is $\{E^?, \mathbf{0}, E, \top\}$. The subeffect relation $\sqsubseteq$ is the partial order given by $E^? \sqsubseteq \mathbf{0} \sqsubseteq \top$ and $E^? \sqsubseteq E \sqsubseteq \top$.*

Note that non-termination is denoted by $\top$, which is the greatest element of the order $\sqsubseteq$, as opposed to the common practice in denotational semantics, where non-termination denotes the least element. Viewing effects as the set of *termination capabilities* that a program can exercise, we define the order so that lower elements have more capabilities, similarly to the subusage relation.

Effects can be considered usages that do not include access labels. We write $(\varphi)^{use}$ for the usage corresponding to φ, defined by:

$$(E^?)^{use} = \mathbf{0}\&E \quad (\mathbf{0})^{use} = \mathbf{0} \quad (E)^{use} = E \quad (\top)^{use} = \mu\alpha.\alpha.$$

We define some operations on effects, which correspond to usage constructors of the same symbols.

DEFINITION 3.7 (Operations on effects). *The operations on effects $\varphi_1 \mathbf{op} \varphi_2$ are defined (where $\mathbf{op}$ is either ;, &,$\otimes$ or $;_E$) by the following tables (the leftmost columns correspond to φ_1 and the topmost rows φ_2) :*

;	$E^?$	$\mathbf{0}$	E	$\top$
$E^?$	$E^?$	$E^?$	E	E
$\mathbf{0}$	$E^?$	$\mathbf{0}$	E	$\top$
E	E	E	E	E
$\top$	$\top$	$\top$	$\top$	$\top$

&	$E^?$	$\mathbf{0}$	E	$\top$
$E^?$	$E^?$	$E^?$	$E^?$	$E^?$
$\mathbf{0}$	$E^?$	$\mathbf{0}$	$E^?$	$\mathbf{0}$
E	$E^?$	$E^?$	E	E
$\top$	$E^?$	$\mathbf{0}$	E	$\top$

$\otimes$	$E^?$	$\mathbf{0}$	E	$\top$
$E^?$	$E^?$	$E^?$	E	E
$\mathbf{0}$	$E^?$	$\mathbf{0}$	E	$\top$
E	E	E	E	E
$\top$	E	$\top$	E	$\top$

$;_E$	$E^?$	$\mathbf{0}$	E	$\top$
$E^?$	$E^?$	$\mathbf{0}$	$E^?$	$\mathbf{0}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
E	$E^?$	$\mathbf{0}$	E	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Example 3.5. $E;\mathbf{0} = E$ and $E\&\mathbf{0} = E^?$ and $E^?;_E \mathbf{0} = \mathbf{0}$. □

DEFINITION 3.8 (Types). *The set of* types, *ranged over by σ, is given by the following syntax:*

$$\sigma ::= \mathbf{bool} \mid (\delta_1 \xrightarrow{\varphi} \delta_2, U) \mid (\mathbf{R}, U)$$
$$\delta ::= \mathbf{bool} \mid (\delta_1 \xrightarrow{\varphi} \delta_2, U_0) \mid (\mathbf{R}, U_0)$$

Here, U_0 ranges over the set of usages that satisfy $U_0 \cong U_0\setminus E$.

$\mathbf{bool}$ is the type of boolean values. $(\delta_1 \xrightarrow{\varphi} \delta_2, U)$ is the type of functions that take a value of type δ_1 as an argument and return a value of type δ_2 and that, during the execution of the body, may raise an exception according to φ. U describes how a function is accessed (i.e., called). $(\mathbf{R}, U)$ is the type of resources that are accessed according to U.

For example, a function of the type $((\mathbf{R}, (R\&\mathbf{0}); C) \xrightarrow{E^?} \mathbf{bool}, 1; 1)$ takes a resource as an argument, closes the resource after a possible read, and may raise an exception during the function call. Moreover, the usage $1; 1$ states that the function is called twice.

We write the (outermost) usage of σ under effect φ by:

$Use_\varphi(\mathbf{bool}) = (\varphi)^{use}$, $Use_\varphi((\sigma_1 \xrightarrow{\varphi'} \sigma_2, U)) = U$ and $Use_\varphi((\mathbf{R}, U)) = U$.

The subusage relation defined in Section 3.1 is extended to the subtype relation $\sigma_1 \leq \sigma_2$ below. It means that a value of type σ_1 may be used as a value of type σ_2.

DEFINITION 3.9 (Subtype relation). *$\sigma_1 \leq \sigma_2$ is the least relation closed under the following rules:*

$$\mathbf{bool} \leq \mathbf{bool} \qquad \frac{U \leq U' \qquad \varphi' \sqsubseteq \varphi}{(\sigma_1 \xrightarrow{\varphi} \sigma_2, U) \leq (\sigma_1 \xrightarrow{\varphi'} \sigma_2, U')} \qquad \frac{U \leq U'}{(\mathbf{R}, U) \leq (\mathbf{R}, U')}$$

3.3 Type Judgment

A type judgment is of the form $\Gamma \parallel \varphi \vdash M : \delta$, read "term M is given type δ under type environment Γ and effect φ" where Γ is a finite mapping from variables to types. The intended meaning of $\Gamma \parallel \varphi \vdash M : \delta$ is that (1) the term M is evaluated to a

value of type δ, if the evaluation terminates, and (2) during the evaluation, each free variable x in M are used according to type $\Gamma(x)$ and an exception may be raised according to effect φ. The meaning of the judgment is tricky when $\Diamond$ appears in Γ [10]: If a usage in $\Gamma(x)$ is guarded by $\Diamond$, the access represented by the usage may be postponed until the value of M is used; otherwise the access cannot be postponed. For example, $x : (\mathbf{R}, R; C) \parallel \mathbf{0} \vdash \mathbf{read}(x); \mathbf{close}(x) : \mathbf{bool}$ and $x : (\mathbf{R}, R; \Diamond C) \parallel \mathbf{0} \vdash \mathbf{read}(x); x : (\mathbf{R}, C)$ are valid judgments, while $x : (\mathbf{R}, R; C) \parallel \mathbf{0} \vdash \mathbf{read}(x); x : (\mathbf{R}, C)$ is invalid. (Precisely speaking, $\mathbf{read}(x)$ and $\mathbf{close}(x)$ must be annotated with $\cdot^{\{x\}}$ in our type system.)

We write $\emptyset$ for the empty type environment, and when $x \notin dom(\Gamma)$, we write $\Gamma, x : \sigma$ for the type environment Δ such that $dom(\Delta) = dom(\Gamma) \cup \{x\}$, $\Delta(x) = \sigma$ and $\Delta(y) = \Gamma(y)$ for $y \in dom(\Gamma)$.

The type judgment relation will be defined by using typing rules. We first give a few auxiliary definitions used in the typing rules.

DEFINITION 3.10. *Let $\mathcal{C}$ be a usage context. Suppose the set of free usage variables in σ or Γ is disjoint from the set of bound usage variables in $\mathcal{C}$. We define $\mathcal{C}[\sigma]$ and $\mathcal{C}[\Gamma]$ by:*

$$\mathcal{C}[\mathbf{bool}] = \mathbf{bool} \quad \mathcal{C}[(\mathbf{R}, U)] = (\mathbf{R}, \mathcal{C}[U])$$
$$\mathcal{C}[(\sigma_1 \xrightarrow{\varphi} \sigma_2, U)] = (\sigma_1 \xrightarrow{\varphi} \sigma_2, \mathcal{C}[U])$$

$$dom(\mathcal{C}[\Gamma]) = dom(\Gamma) \quad \mathcal{C}[\Gamma](x) = \mathcal{C}[\Gamma(x)]$$

DEFINITION 3.11. *Let $\mathbf{op}$ be a binary usage constructor ';' , '&' or ';$_E$'. $\sigma_1 \mathbf{op} \sigma_2$ is defined as follows:*

$\mathbf{bool}$	$\mathbf{op}$	$\mathbf{bool}$	$=$	$\mathbf{bool}$
$(\sigma_1 \xrightarrow{\varphi} \sigma_2, U_1)$	$\mathbf{op}$	$(\sigma_1 \xrightarrow{\varphi} \sigma_2, U_2)$	$=$	$(\sigma_1 \xrightarrow{\varphi} \sigma_2, U_1 \mathbf{op} U_2)$
$(\mathbf{R}, U_1)$	$\mathbf{op}$	$(\mathbf{R}, U_2)$	$=$	$(\mathbf{R}, U_1 \mathbf{op} U_2)$

Let Γ_1 and Γ_2 be type environments with the same domain ($dom(\Gamma_1) = dom(\Gamma_2)$). Then, $\Gamma_1 \mathbf{op} \Gamma_2$ is defined as follows:

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

For example, the type environment $\Gamma_1; \Gamma_2$ states that the value stored in each variable $x \in dom(\Gamma_1) (= dom(\Gamma_2))$ should be first used according to $\Gamma_1(x)$ and then should be used according to $\Gamma_2(x)$.

We also define the type environment $\blacklozenge_x \Gamma$ as follows:

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

Note that, if $\Gamma(x) = \mathbf{bool}$ or $\Gamma(x) = (\sigma_1 \xrightarrow{\varphi} \sigma_2, U)$, then $\blacklozenge_x \Gamma$ is not defined.

DEFINITION 3.12 (Type Judgment). *The type judgment relation $\Gamma \parallel \varphi \vdash M : \delta$ is the least relation closed under the rules in Figure 4.*

Note that when $\Gamma_1; \Gamma_2$ or $\Gamma_1;_E \Gamma_2$ appears in the conclusion of a rule, the rule can be applied only when the operation is well-defined; In particular, it must be the case that $dom(\Gamma_1) = dom(\Gamma_2)$.

Now we explain the key typing rules of T-RAISE, T-TRY, T-FUN, T-APP, and T-WEAK below. The others are essentially the same as those in the original type system [10] (except for the effect part in type judgments).

Rule T-RAISE is the easiest: Since the term $\mathbf{raise}$ immediately raises an exception without accessing any resources, it is typed under the empty type environment with effect E.

Rule T-TRY is explained as follows. Usages in Γ_1 and Γ_2 record how each resource is accessed before and after, respectively, an exception is raised. So, the total usage for $\mathbf{try}\ M_1\ \mathbf{with}\ M_2$ is expressed by $\Gamma_1;_E \Gamma_2$, obtained by applying $;_E$ to usages in those type environments.

Rule T-FUN is defined according to the following intuitions. First, the premise says: *Each time* the function $\mathbf{fun}(f, x, M)$ is called, its body M causes effect φ, accessing the function's free variables according to Γ. In addition, M recursively calls f according to usage U_1. M also uses the argument x according to type σ_1 and yields a value of type δ_2. Therefore, the function is given a type $(\delta_1 \xrightarrow{\varphi} \delta_2, U)$, where $\delta_1 \leq \sigma_1 \backslash E$. Here, E is removed by $\backslash E$ from σ_1 since any possible exception that may be raised in M is already considered in the latent effect φ. The type environment for the function is obtained by *multiplying* $\Diamond(\Gamma \backslash E)$ (which expresses how the function's free variables are accessed *each time* the function is called) according to U (which expresses how often the function is called from the outside) and $U_1 \backslash E$ (which expresses how often the function is called recursively). We safely approximate this multiplication by considering only the following three simple cases: the function is never called, it is called exactly once, or it is called an arbitrary number of times. In the first case, the free variables are never accessed. In the second case, the free variables are accessed exactly according to $\Diamond(\Gamma \backslash E)$. In the last case, $\Diamond(\Gamma \backslash E)$ is arbitrarily replicated by ! where !U is defined by $\mu\alpha.\mathbf{0} \& (U \otimes \alpha)$ and $!\Gamma$ is its pointwise extension. Thus, the approximated multiplication $\Delta^{\mathbf{fun}}_{(U, U_1, \Gamma)}$ is defined as follows:

$$\Delta^{\mathbf{fun}}_{(U, U_1, \Gamma)} = \begin{cases} \emptyset & \text{if } 1 \notin [\![U]\!] \\ \Gamma & \text{if } (1 \in [\![U]\!] \subseteq \{\epsilon, 1, 1\downarrow\}) \wedge (1 \notin [\![U_1]\!]) \\ !\Gamma & \text{otherwise} \end{cases}$$

Rule T-APP is explained as follows. When the term $M_1 M_2$ is evaluated, the term M_1 is first evaluated to a function and M_2 is then evaluated and finally the function is called. The type environment $\Gamma_1; \Gamma_2; (\varphi_3)^{use}$ reflects this order, where φ_3 comes from the latent effect of the type of the function.

Rule T-WEAK deals with weakening and subsumption (on types and effects). Here, $\Gamma \leq_\varphi \Gamma'$ is defined by

$$\Gamma \leq_\varphi \Gamma' \Leftrightarrow \begin{cases} dom(\Gamma) \supseteq dom(\Gamma') & \\ \Gamma(x) \leq \Gamma'(x) & \text{for each } x \in dom(\Gamma') \\ Use_\varphi(\Gamma'(x)) \leq (\varphi)^{use} & \text{for each } x \in dom(\Gamma) \setminus dom(\Gamma') \end{cases}$$

It means that if we add $x : \sigma$ to the domain of Γ', σ must respect the effect φ of the term. For example, from $\Gamma' \parallel E \vdash M : \delta$, we can derive $\Gamma', x : (\mathbf{R}, E) \parallel E \vdash M : \delta$ (where $x \notin dom(\Gamma')$) but not $\Gamma', x : (\mathbf{R}, \mathbf{0}) \parallel E \vdash M : \delta$. In the latter, the usage of x contradicts with the effect E.

Example 3.6. The following type judgments can be derived for the terms M_1, M_2 and M of Example 2.2:

$$x : (\mathbf{R}, I; ((W; C) \& E)) \parallel E^? \vdash M_1 : \mathbf{bool}$$
$$x : (\mathbf{R}, I; ((W; C) \& E));_E C \parallel \mathbf{0} \vdash M_2 : \mathbf{bool}$$
$$\emptyset \parallel \mathbf{0} \vdash M : \mathbf{bool}\ .$$

□

4. Type Soundness

Our type system is sound in the sense that if a closed well-typed term of type τ where $Use(\tau) = \mathbf{0}$ is evaluated, any resource is accessed according to the specification (declared by the resource creation primitive $\mathbf{new}^\Phi()$).

We say that M is *well-annotated* if all the annotations on escape information $\cdot^{\{x\}}$ are sound, i.e., if $(\{\}, M)$ is never reduced to a

$$\frac{c = \mathbf{true} \text{ or } \mathbf{false}}{\emptyset \parallel \mathbf{0} \vdash c : \mathbf{bool}} \quad \text{(T-CONST)}$$

$$x : \Diamond\delta \parallel \mathbf{0} \vdash x : \delta \quad \text{(T-VAR)}$$

$$\frac{[\![U]\!] \subseteq \Phi}{\emptyset \parallel \mathbf{0} \vdash \mathbf{new}^{\Phi}() : (\mathbf{R}, U)} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \parallel \varphi \vdash M : (\mathbf{R}, a)}{\Gamma \parallel \varphi \vdash \mathbf{acc}^{a}(M) : \mathbf{bool}} \quad \text{(T-ACC)}$$

$$\frac{\Gamma_1 \parallel \varphi_1 \vdash M_1 : \mathbf{bool} \quad \Gamma_2 \parallel \varphi_2 \vdash M_2 : \delta \quad \Gamma_2 \parallel \varphi_2 \vdash M_3 : \delta}{\Gamma_1;\Gamma_2 \parallel \varphi_1;\varphi_2 \vdash \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : \delta} \quad \text{(T-IF)}$$

$$\frac{\Gamma_1 \parallel \varphi_1 \vdash M_1 : \sigma_1 \setminus E \quad \Gamma_2, x : \sigma_1 \parallel \varphi_2 \vdash M_2 : \delta_2}{\Gamma_1;\Gamma_2 \parallel \varphi_1;\varphi_2 \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \delta_2} \quad \text{(T-LET)}$$

$$\frac{\Gamma_1 \parallel \varphi_1 \vdash M_1 : (\delta_1 \xrightarrow{\varphi_3} \delta_2, 1) \quad \Gamma_2 \parallel \varphi_2 \vdash M_2 : \delta_1}{\Gamma_1;\Gamma_2;(\varphi_3)^{use} \parallel \varphi_1;\varphi_2;\varphi_3 \vdash M_1 M_2 : \delta_2} \quad \text{(T-APP)}$$

$$\frac{\Gamma, f : (\delta_1 \xrightarrow{\varphi} \delta_2, U_1), x : \sigma_1 \parallel \varphi \vdash M : \delta_2 \quad \delta_1 \leq \sigma_1 \setminus E}{\Delta^{\mathbf{fun}}_{(U, U_1 \setminus E, \Diamond(\Gamma \setminus E))} \parallel \mathbf{0} \vdash \mathbf{fun}(f, x, M) : (\delta_1 \xrightarrow{\varphi} \delta_2, U)} \quad \text{(T-FUN)}$$

$$\frac{\Gamma \parallel \varphi \vdash M : \delta}{\blacklozenge_x \Gamma \parallel \varphi \vdash M^{\{x\}} : \delta} \quad \text{(T-NOW)}$$

$$\emptyset \parallel E \vdash \mathbf{raise} : \delta \quad \text{(T-RAISE)}$$

$$\frac{\Gamma_1 \parallel \varphi_1 \vdash M_1 : \delta \quad \Gamma_2 \parallel \varphi_2 \vdash M_2 : \delta}{\Gamma_1;_E \Gamma_2 \parallel \varphi_1;_E \varphi_2 \vdash \mathbf{try}\ M_1\ \mathbf{with}\ M_2 : \delta} \quad \text{(T-TRY)}$$

$$\frac{\varphi \sqsubseteq \varphi' \quad \Gamma \leq_{\varphi'} \Gamma' \quad \Gamma' \parallel \varphi' \vdash M : \delta' \quad \delta' \leq \delta}{\Gamma \parallel \varphi \vdash M : \delta} \quad \text{(T-WEAK)}$$

Figure 4. Typing Rules

configuration $(H, \mathcal{E}[v^{\{x\}}])$ such that $x \in \mathbf{FV}(v)$. The soundness of our type system is stated formally as follows:

THEOREM 4.1 (Type Soundness). *Suppose M is well-annotated. If $\emptyset \parallel \varphi \vdash M : \delta$ and $Use_{\mathbf{0}}(\delta) \leq \mathbf{0}$, then all the following properties hold:*

(1) $(\{\}, M) \not\rightsquigarrow^{*}$ **Error**.
(2) If $(\{\}, M) \rightsquigarrow^{*} (H, M') \not\rightsquigarrow$, *then* $\forall x \in dom(H). \downarrow \in H(x)$.

The condition $Use_{\mathbf{0}}(\tau) \leq \mathbf{0}$ states that even if the term M is evaluated to a resource, the resource may not be accessed after the evaluation. Property (1) means that M never performs an illegal resource access. Property (2) means that all the resources are used up when the evaluation terminates (normally or abruptly). Note that property (1) holds even if $Use_{\mathbf{0}}(\delta) \not\leq \mathbf{0}$.

We give an outline of the proof of Theorem 4.1 below. The full proof is available in the full version [11].

We first define a type judgment relation $\varphi \vdash (H, M) : \delta$, which means that the state (H, M) is well-typed under the effect φ.

DEFINITION 4.1.

$$\frac{\begin{array}{c} x_1 : (\mathbf{R}, U_1), \ldots, x_n : (\mathbf{R}, U_n) \parallel \varphi \vdash M : \delta \\ dom(H) = \{x_1, \ldots, x_n\} \\ [\![U_1]\!] \subseteq H(x_1), \ldots, [\![U_n]\!] \subseteq H(x_n) \end{array}}{\varphi \vdash (H, M) : \delta}$$

The first premise means that M uses the resources $x_1, \ldots, x_n$ according to $U_1, \ldots, U_n$. The other premises mean that the current heap indeed allows such resource usage.

We list main lemmas below. Lemma 4.1 states that typing is preserved by reductions. Lemma 4.2 states that an invalid resource access cannot happen immediately in a well-typed state. Lemma 4.3 states that evaluation may terminate only when the expression becomes a value or raises an uncaught exception. Lemma 4.4 states that every heap element contains $\downarrow$ in a well-typed, final state. (See [11] for proofs.)

LEMMA 4.1 (Type Preservation). *If $\varphi \vdash (H, M) : \sigma$ and $(H, M) \rightsquigarrow (H', M')$, then $\varphi \vdash (H', M') : \sigma$.*

LEMMA 4.2 (Safety I). *If $\varphi \vdash (H, M) : \delta$, then $(H, M) \not\rightsquigarrow$* **Error**.

LEMMA 4.3 (Progress). *Suppose M is well-annotated. If $\varphi \vdash (H, M) : \delta$, then either $(H, M) \rightsquigarrow (H', M')$ for some H' and M' or M is either a value v, or of the form $\mathcal{E}^{\overline{try}}[\mathbf{raise}]$.*

LEMMA 4.4 (Safety II). *(1) If $\varphi \vdash (H, v) : \delta$ and $Use_{\mathbf{0}}(\delta) \leq \mathbf{0}$, then $\forall x \in dom(H). \downarrow \in H(x)$.*
(2) If $\varphi \vdash (H, \mathcal{E}^{\overline{try}}[\mathbf{raise}]) : \delta$, then $\forall x \in dom(H). \downarrow \in H(x)$.

Theorem 4.1 is an immediate corollary of the above lemmas: Property (1) follows from Lemmas 4.1 and 4.2; Property (2) follows from Lemmas 4.3 and 4.4.

5. Type Inference

By the soundness of the type system, a sufficient condition for a closed term M to access resources in a valid manner is that there exists an effect φ and δ such that $\emptyset \parallel \varphi \vdash M : \delta$ and $Use_{\mathbf{0}}(\delta) \leq \mathbf{0}$. (Actually, it is sufficient to give an algorithm to check whether $\emptyset \parallel \varphi \vdash M : \mathbf{bool}$, since if M does not have type $\mathbf{bool}$, we can check the term $(\lambda x.\mathbf{true})M$ instead.) We sketch an algorithm for checking the sufficient condition in this section.

The overall structure of the algorithm is the same as the constraint-based type inference algorithm for Igarashi and Kobayashi's type system [10]. Based on the typing rules, we can construct an algorithm which, given a closed term M, generates constraints on variables expressing unknown usages, effects, and types as a sufficient and necessary condition for $\emptyset \parallel \varphi \vdash M : \delta$. By reducing the constraints on type variables (using the standard unification algorithm), we can obtain constraints of the following form:

$$\{ \quad \begin{array}{l} \xi_1 \leq \varphi_1, \ldots, \xi_m \leq \varphi_m, \\ \alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n, \ [\![U'_1]\!] \subseteq \Phi_1, \ldots, [\![U'_k]\!] \subseteq \Phi_k \end{array} \quad \}$$

At this point, $U_1, \ldots, U_n$ may contain effect variables (in the form of $(\xi)^{use}$) and expressions of the form $\Delta^{\mathbf{fun}}_{(U_1, U_2, U_3)}$ (which is defined in the same way as $\Delta^{\mathbf{fun}}_{(U_1, U_2, \Gamma)}$), where U_1 and U_2 are the usages of functions. To remove them, we first solve constraints on effects and function usages by using a standard method for solving constraints over a finite lattice [18].

Then, the greatest solution for a subusage constraint of the form $\alpha \leq U$ (where U no longer contains an effect or $\Delta^{\mathbf{fun}}_{(U_1,U_2,U_3)}$) can be represented by $\mu\alpha.U$. Thus, the above constraints can be further reduced to constraints of the form: $\{[\![U_1'']\!] \subseteq \Phi_1, \ldots, [\![U_k'']\!] \subseteq \Phi_k\}$.

Like in our previous type system [10], the relation $[\![U]\!] \subseteq \Phi$ is generally undecidable (think of the case where Φ is a context-free language). We, however, think that for a certain class of languages for describing Φ, we can develop an algorithm (which may be incomplete but sound at least) to verify the condition $[\![U]\!] \subseteq \Phi$.

In fact, we have already implemented such an algorithm for the case where $\Phi = (R^* C \downarrow)^{\#}$: see Section 6.

Example 5.1. Consider the term

$$\mathbf{let}\ x = \mathbf{new}^{(R^* C \downarrow)^{\#}}() \ \mathbf{in}\ \mathbf{let}\ y = \mathbf{new}^{(W^* C \downarrow)^{\#}}() \ \mathbf{in}\ M^a$$

Here, M^a is a term obtained by annotating every access to a resource x by $(\cdot)^{\{x\}}$ —for example, $\mathbf{close}(x)$ becomes $\mathbf{close}(x)^{\{x\}}$— in the term M of Example 2.3. Then, we finally gain the following constraints after extracting and solving subusage and subeffect constrains:

$$[\![(!(\diamond((R; (\mathbf{0}\&E); E)\setminus E)); E);_E C]\!] \subseteq (R^* C \downarrow)^{\#}$$
$$[\![(!(\diamond((W; (\mathbf{0}\&E); E)\setminus E)); E);_E C]\!] \subseteq (W^* C \downarrow)^{\#}$$

Since these constraints are satisfied, we can conclude that the above term is well-typed. □

Example 5.2. Consider the term

$$M \triangleq \mathbf{let}\ x = \mathbf{new}^{(R^* C \downarrow)^{\#}}() \ \mathbf{in}\ M_2,$$

where $M_2 \triangleq \mathbf{try}\ (\mathbf{read}(x)^{\{x\}}; \mathbf{raise})\ \mathbf{with}\ \mathbf{close}(x)$. The following constraints are extracted.

$$\Gamma_{\mathbf{read}(x)} = x : (\mathbf{R}, \alpha_1) \qquad \Gamma_{\mathbf{read}(x)^{\{x\}};\mathbf{raise}} = x : (\mathbf{R}, \alpha_5)$$
$$\Gamma_{\mathbf{raise}} = x : (\mathbf{R}, \alpha_2) \qquad \Gamma_{M_2} = x : (\mathbf{R}, \alpha_6)$$
$$\Gamma_{\mathbf{close}(x)} = x : (\mathbf{R}, \alpha_3) \qquad \Gamma_{\mathbf{new}^{(R^* C \downarrow)^{\#}}()} = \emptyset$$
$$\Gamma_{\mathbf{read}(x)^{\{x\}}} = x : (\mathbf{R}, \alpha_4) \qquad \Gamma_M = \emptyset$$

$$\varphi_{\mathbf{read}(x)} = \xi_1 \quad \varphi_{\mathbf{raise}} = \xi_2 \quad \varphi_{\mathbf{close}(x)} = \xi_3$$
$$\varphi_{\mathbf{read}(x)^{\{x\}}} = \xi_4 \quad \varphi_{\mathbf{read}(x)^{\{x\}};\mathbf{raise}} = \xi_5$$
$$\varphi_{M_2} = \xi_6 \quad \varphi_{\mathbf{new}^{(R^* C \downarrow)^{\#}}()} = \xi_7 \quad \varphi_M = \xi_8$$

$$\delta_{\mathbf{read}(x)} = \mathbf{bool} \quad \delta_{\mathbf{raise}} = \mathbf{bool} \quad \delta_{\mathbf{close}(x)} = \mathbf{bool}$$
$$\delta_{\mathbf{read}(x)^{\{x\}}} = \mathbf{bool} \quad \delta_{\mathbf{read}(x)^{\{x\}};\mathbf{raise}} = \mathbf{bool}$$
$$\delta_{M_2} = \mathbf{bool} \quad \delta_{\mathbf{new}^{(R^* C \downarrow)^{\#}}()} = (\mathbf{R}, \alpha_7) \quad \delta_M = \mathbf{bool}$$

$$[\![\alpha_7]\!] \subseteq (R^* C \downarrow)^{\#} \qquad \alpha_4 \leq \blacklozenge\alpha_1 \qquad \xi_1 \sqsubseteq \mathbf{0} \qquad \xi_5 \sqsubseteq \xi_4; \xi_2$$
$$\alpha_1 \leq \diamond R \qquad \alpha_5 \leq \alpha_4; \alpha_2 \qquad \xi_2 \sqsubseteq E \qquad \xi_6 \sqsubseteq \xi_5;_E \xi_3$$
$$\alpha_2 \leq E \qquad \alpha_6 \leq \alpha_5;_E \alpha_3 \qquad \xi_3 \sqsubseteq \mathbf{0} \qquad \xi_7 \sqsubseteq \mathbf{0}$$
$$\alpha_3 \leq \diamond C \qquad \alpha_7 \leq \alpha_6 \qquad \xi_4 \sqsubseteq \xi_3 \qquad \xi_8 \sqsubseteq \xi_7; \xi_6$$

Here, Γ_N, ξ_N, and δ_N are respectively the type environment, effect, and type of a subterm N. By solving the constraints on effects and usages, we obtain $\alpha_7 = (\blacklozenge\diamond R; E);_E \diamond C$ and $\varphi_8 = \mathbf{0}$. Since $[\![\alpha_7]\!] \subseteq (R^* C \downarrow)^{\#}$ holds, we can conclude that M is well-typed. □

6. Experiments

Based on our type system, we have implemented a prototype resource usage analyzer. The implementation is made available at `http://www.kb.ecei.tohoku.ac.jp/~iwama/rue/`. The analyzer inputs a program written in $\lambda^{\mathcal{R}}_E$, without annotations $(\cdot^{\{x\}})$ on escape information. The analyzer first performs the standard type inference and annotate terms of non-function types with escape information. It then performs the usage analysis as described in the previous section. In the final phase, constraints of the form $[\![U]\!] \subseteq \Phi$ are checked. Currently, the analyzer accepts only the specification $\Phi = (R^* C \downarrow)^{\#}$, and uses a sound but incomplete algorithm for checking $[\![U]\!] \subseteq \Phi$. The algorithm works as sketched in Section 6.6 of our previous paper [10]. The basic observation behind the algorithm is as follows. Although the language of usage expressions is very expressive (for example, it can express any context-free languages as well as some context-sensitive languages), we can approximate usages by using a finite set of *abstract usages* as long as the specification Φ is regular; For example, we need not distinguish between usages $R; C$ and $R; R; C$ when the specification is $(R^* C \downarrow)^{\#}$. We have designed an abstract usage domain that is sufficient for checking the inclusion with respect to the specification $\Phi = (R^* C \downarrow)^{\#}$, so that the constraint $[\![U]\!] \subseteq \Phi$ can be replaced by a decidable, sufficient condition $[\![\alpha(U)]\!] \subseteq \Phi$ (where α is the abstraction function). The formalization of an algorithm that can deal with arbitrary regular languages Φ is left for future work.

Experiments We have tested several programs including the examples given in this paper (where $\mathbf{init}(x)$ is replaced by $\mathbf{read}(x)$ since the current system can handle only the specification $(R^* C \downarrow)^{\#}$). We confirmed that the analyzer gives correct answers. The tested programs include the following tricky one.

```
let create =
  fun(f,x,let y=new[read*;close]() in y) in
let repeat =
  fun(g,x, let z = create x in
    try
      if acc[read](z) then raise
      else (g x; acc[close](z))
    with acc[close](z)) in
repeat true;;
```

The above program repeatedly creates a new resource and closes it. Note that arbitrarily many resources may be created, and also that arbitrarily many exception handlers can be nested.

We have also inspected source programs of O'Caml compiler (3.08.4), manually translated some fragments of the programs accessing input files, and run our analyzer. Of 46 fragments of the code we have inspected, 40 of them can be categorized into the access patterns (expressed in our target language) summarized in Figure 5. We have confirmed that all of the four patterns can be analyzed by our prototype system. For example, the following is an example of the 4th pattern:

```
let exclude filename =
let ic = open_in filename in
try
  while true do
    let s = input_line ic in
    primitives := StringSet.remove s !primitives
  done
with End_of_file -> close_in ic
   | x -> close_in ic; raise x
```

The body of the above function is expressed in our language:

```
let input_line = lambda x.
      if acc[read](x) then true else raise in
let ic = new[read*;close]()
in  try  fun(g,x, input_line ic;g x) true
    with acc[close](ic);;
```

Our prototype analyzer accepts the above program, while it rejects the slightly modified program obtained by replacing `acc[close](ic)` with `false`.

Normal pattern: 16 places ... $(\mathbf{let}\ z = \mathbf{new}^{(R^*C)^\#}()\ \mathbf{in}\ (\mathbf{read}(z); ..; \mathbf{close}(z)))$ ...
TryWith pattern: 18 places $\mathbf{let}\ z = \mathbf{new}^{(R^*C)^\#}()$ $\mathbf{in}\ \mathbf{try}\ \mathbf{read}(z);\ ..\ ; \mathbf{close}(z)\ \mathbf{with}\ \mathbf{close}(z)$
TryClose pattern: 3 places $\mathbf{let}\ z = \mathbf{new}^{(R^*C)^\#}()$ $\mathbf{in}\ (\mathbf{try}\ \mathbf{read}(z);\ ..\ \mathbf{with}\ ..); \mathbf{close}(z)$
WhileTrue pattern: 5 places $\mathbf{let}\ f = \lambda x.(..;\ \mathbf{read}(x); ..)\ \mathbf{in}$ $\mathbf{let}\ z = \mathbf{new}^{(R^*C)^\#}()$ $\mathbf{in}\ \mathbf{try}\ while(true)\{..;\ (f\ z);\ ..\}\ \mathbf{with}\ \mathbf{close}(z)$

Figure 5. Typical file access patterns in O'Caml program

Of the remaining 6 fragments (that do not fit any of the four patterns), two of them seem to forget to close a file (in `asmcomp/asmlink.ml` and `debugger/source.ml`). Our analyzer rejects them as ill-typed.

The other 4 fragments seem to use files correctly, but they are rejected or cannot be handled by our analyzer. They use pointers (reference cells) or records to store file pointers. The following is the most interesting one:

```
let ic = open_in_bin Sys.executable_name in
Bytesections.read_toc ic;
{ read_string = Bytesections.read_section_string ic;
  read_struct = Bytesections.read_section_struct ic;
  close_reader = fun () -> close_in ic }
```

It opens a file, and then creates a record consisting of closures for reading and closing files. To properly handle this, we need to refine the type system to control the order between the uses of record elements.

7. Discussion

Alternative approach to dealing with exceptions An alternative, more straightforward approach to dealing with exceptions would be to encode exception primitives into $\lambda^{\mathcal{R}}$ (e.g., by using the continuation-passing style) [10] or the extension of π-calculus with resources [15], and then apply previous type systems [10, 15]. The resulting analysis is not, however, accurate enough to deal with the examples given in this paper. For example, let us consider the following program:

$$M \triangleq \mathbf{let}\ f = \lambda y.\mathbf{raise}\ \mathbf{in}\ (\mathbf{try}\ f()\ \mathbf{with}\ \mathbf{write}(x)); \mathbf{close}(x)$$

It can be encoded into following $\lambda^{\mathcal{R}}$ term:

$$\begin{array}{l}\mathbf{let}\ f = \lambda y.\lambda h.\lambda c.h()\ \mathbf{in}\\ \mathbf{let}\ c = \lambda z.\mathbf{close}(x)\ \mathbf{in}\\ \mathbf{let}\ h = \lambda z.(\mathbf{write}(x); c())\ \mathbf{in}\ f()(h)(c)\end{array}$$

Here the function f takes two continuations h and c as extra arguments: h is to be called when an exception is raised, while c is to be called when the evaluation terminates normally. Igarashi and Kobayashi's type system would infer that the usage of x is $\diamond C; \diamond W$, which does not tell us which of W or C is performed first.

Another approach would be to put information about both resource usage and exceptions into effects to make the type system simpler. For example, a function

$$\lambda x.(\mathbf{write}(x); \mathbf{close}(x); \mathbf{raise})$$

can be given a type $(\mathbf{R}, \rho) \xrightarrow{\rho^W;\rho^C;E} \mathbf{bool}$, where ρ is an abstract resource (called a region). The effect $\rho^W; \rho^C; E$ means that a resource belonging to the region ρ is written and closed, and then an exception is raised. As discussed elsewhere [10, 14], however, this approach does not work well when different resources are aliased to the same region.

Extensions for multiple exceptions and exception arguments Unlike the simple language studied in this paper, real languages like ML allow multiple exceptions and exception arguments. We can extend our type system to deal with multiple exceptions, by introducing distinct usage constructors E_i and $;_{E_i}$ for each kind of exception. As for exception arguments, there are two main issues: (1) how to deal with an exception having a resource as an argument (for example, consider the case where an exception carries a file that must be closed), and (2) how to deal with pattern matching on arguments, like "`try ... with E 1 -> ...`". We can deal with both issues by combining our type system with analyses of uncaught exceptions [17, 27]. For the first issue, we can impose a restriction that the usage U of a resource passed as an argument of an uncaught exception must be a subusage of $\mathbf{0}$. For the second issue, we can extend usage constructors E_i and $;_{E_i}$ by annotating them with information (like "rows" [17]) about exception arguments.

8. Related Work

A number of type systems have been proposed for statically checking whether a certain kind of resource is accessed in a valid manner [1, 3, 8, 10, 22, 24]. Only a few of them, however, deal with exception primitives. Type systems for JVM lock primitives [3, 12] support exceptions. In those type systems, the handler for each exception is statically known, so that exceptions can be treated in the same manner as *if*-statements.

It seems easier to extend effect-based type systems [4, 7, 19] for dealing with exceptions than to extend Igarashi and Kobayashi's type system. The effect-based approach, however, suffers from the problem mentioned in Section 7.

Another approach to the analysis of resource usage in the presence of exceptions would be to extend work on typestates [20, 25, 26, 6, 5]. In this approach, each type have several states (typestates) and a typestate of each resource may be changed by resource accesses or procedure calls. Each access to a resource is permitted only if the resource is in a valid typestate, so, inferring the typestate of each resource at any program points, we can verify valid resource usage.

Indeed, the original work on typestates [20] does deal with exceptions. However, unlike ours, their method requires; (1) explicit annotations on procedures, where with a specification: *pre-typestates*, *post-typestates*, *exception-typestates* meaning that typestates of argument values before resource-access/procedure-call, after the resource-access/procedure-call has been performed normally and abruptly; (2) cannot deal with aliasing, that makes their verification be unsound; (3) cannot deal with higher-order functions.

Indeed, the original work on typestates [20] does deal with exceptions. However, unlike ours, their method requires explicit annotations on procedures, and cannot deal with aliasing and higher-order functions. The succeeding work on typestates [25, 26, 6, 5] extends the original work to lift the restriction on aliasing and deal with data structures (or objects) and pointers (although exceptions are no longer explicitly discussed). Those methods seem to either require complex annotations for higher-order functions or suffer from the same problem as the effect-based approach discussed in Section 7.

Kobayashi [14] has proposed another combination of linear types and effect systems. His type system is, however, so complicated that no reasonable type inference algorithm has been developed.

In parallel to the present work, we have recently studied type-based resource usage analysis for concurrent programs [15]. It would be interesting future work to integrate the type system in this paper with that type system.

Model checking technologies [2, 9] have recently been applied to verification of temporal properties of programs. Advantages of our type-based approach are that our analysis is modular, and that our analysis can deal with programs creating infinitely many resources (recall the tricky example shown in Section 6).

9. Conclusion

We have extended Igarashi and Kobayashi's type-based resource usage analysis to deal with exceptions, proved the soundness of the extended analysis, and implemented a prototype analyzer.

Future work includes an extension of the type system to deal with a larger set of language constructs (e.g., multiple exceptions, pointers, concurrency primitives) and development of an algorithm for checking $[\![U]\!] \subseteq \Phi$ for a certain class of languages Φ.

Acknowledgments

We would like to thank Eijiro Sumii, Manuel Fähndrich and anonymous referees for useful comments.

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] 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.

[5] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[6] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 439–462. Springer, 2003.

[7] 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.

[8] 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.

[9] 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.

[10] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, Mar. 2005.

[11] F. Iwama, A. Igarashi, and N. Kobayashi. Resource usage analysis for a functional language with exceptions, 2005. Full version. Available from `http://www.kb.ecei.tohoku.ac.jp/~iwama/rue/res-use-exce-full.pdf`.

[12] F. Iwama and N. Kobayashi. A new type system for JVM lock primitives. In *Proceedings of ASIA-PEPM'02*, pages 156–168. ACM Press, 2002.

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

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

[15] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for π-calculus. In *Proceedings of VMCAI'06*, 2006.

[16] I. Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.

[17] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 276–290, 1999.

[18] J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.

[19] C. Skalka and S. Smith. History effects and verification. In *Proceedings of APLAS 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 107–128. Springer-Verlag, 2004.

[20] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE, Transactions on Software Engineering*, 12(1):157–171, Jan. 1986.

[21] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 162–173, 1992.

[22] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

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

[24] 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.

[25] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 70–82, New York, NY, USA, 2000. ACM Press.

[26] Z. Xu, T. W. Reps, and B. P. Miller. Typestate checking of machine code. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 335–351, London, UK, 2001. Springer-Verlag.

[27] K. Yi. Compile-time detection of uncaught exceptions in Standard ML programs. In *Proceedings of SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 238–254, 1994.