Enforcing Resource Usage Protocols via Scoped Methods*

Gang Tan Xinming Ou David Walker
Princeton University
{gtan, xou, dpw}@cs.princeton.edu

Abstract

Traditional modularity mechanisms such as Java's classes and packages or ML's structures and functors restrict the set of functions that may be applied to an object, but are unable to restrict the timing of these function applications effectively. We propose a new language construct, the *scoped method*, which allows the implementer of a class to specify a temporal *resource usage protocol*. This protocol limits the sequence of methods that may be applied to an object. For example, a protocol for file access might specify that the file must be opened, read or written some number of times, and then closed. We present a type-based analysis to enforce the protocol and prove its correctness.

1 Introduction

One of the fundamental ideas of modern programming languages is to provide support for data abstraction. In object-oriented programming languages such as Java and C++, classes and packages provide protection mechanisms that allow programmers to hide data representations, and in functional languages like ML, opaque signatures serve a similar purpose. By hiding data representations, implementers can establish and maintain many of the invariants necessary for a correct implementation of the module. In particular, it is possible to provide guarantees that data with an abstract type has just the right structure to satisfy the requirements of a given function or method declared in the same interface. However, while conventional abstraction mechanisms provide excellent support for controlling data structure, they offer little to help implementors control the order or timing of methods or function calls.

A computational *resource* is an object that must be used according to some well-defined *protocol*. In other words, correct usage of the object demands a certain pattern of method calls be applied to the object. For example, a file needs to be opened before being read. A memory cell cannot be accessed after deallocation. A socket needs to be bound to a local port before it can receive data and should be closed when it is no longer needed. The correctness of software depends on faithfully following these protocols, but traditional modularity mechanisms are unable to enforce them.

The main contribution of this paper is to propose a new abstraction mechanism, called a *scoped method*, for encapsulating resource usage protocols. We also design a type system to check whether a program obeys such protocols and we prove that any well-typed program will not violate the protocol.

1.1 Scoped Methods

The role of a scoped method is to encapsulate a resource usage protocol. Definition and invocation of scoped methods is best explained by an example. Figure 1 defines a class for manipulating files in a Java-

*This paper is a revised version of Gang Tan and Xinming Ou's final project in David Walker's course on Foundations of Language-based Security at Princeton University, Spring 2002. The opinions and conclusions contained in this paper are the authors' alone. We would like to thank Princeton University for helping support Gang and Xinming's travel to FOOL 10. We would also like to thank Microsoft research for a generous gift, DARPA for award F30602-99-1-0519 and NSF Trusted Computing Program for grant CCR-0208601, all of which helped support this research.

```
class ROFile {
  private void open(String path);
  private void close();
  Byte read();
  void readOnly(String path){
    enter {this.open(path);}
    finally {this.close();}
  }
}
```

Figure 1: Read Only Files

```
void readTenBytes(fn) {
  File f = new File ();
  scope f.readOnly(fn) {
    Byte b;
    int i = 0;
    while (i<10) {
      i++;
      b = f.read();
      ...
      // do something with the input
    }
  }
}
```

Figure 2: A client of `ROFile`

like syntax. It includes methods for opening, closing and reading files. We omit the implementation details.

The `readOnly` method is one of our special scoped methods. When `f` is a file object, the `readOnly` method may be invoked as follows.

```
scope f.readOnly (path) {
  ...
}
```

Operationally, the code in the enter clause of the scoped method is executed as the new scope is entered (before the ellipsis in the example above) and the code in the finally clause of scoped method is executed as the new scope is exited. Figure 2 presents a more elaborate use of a scoped method.

The `readOnly` method in Figure 1 encapsulates a portion of the protocol that a client must follow in order to use read-only files correctly: every time a read-only file is opened, it must be paired with a corresponding close. The client need not remember to match up open and close method calls when using the `readOnly` method, the implementer has done it for them. Moreover, the client is unable to call open and close methods at inappropriate points since they are declared private. Still, there is another element to correct usage of read-only files. The read method should only be called within the scope of the corresponding `readOnly` method.

1.2 Effects

Figure 3 presents a revised version of the read-only file class that enforces the constraint that the `read` method is only called within the scope of the corresponding `readOnly` method. The main additions are the effect declaration and the annotations on both the `read` method and the scoped `readOnly` method.

First, the declaration `effect` r`;` declares a new sort of effect under the control of the implementer of the current class. Next, the implementer may assert that this effect occurs when methods in the current class are invoked. For example, in this case, we decide that the `read` method incurs the effect r. Finally, the class implementer may decide to permit certain effects within the scope of a scoped method. Here, we decide that within the scope of the `readOnly` method, we will allow the r effect.

The annotation on the `readOnly` method is actually a regular expression that can describe a sequence of effects that are allowed to occur. Here, we use the annotation `permit` r^* to indicate that any sequence of zero or more read effects are allowed. If we wanted to ensure that exactly one read effect occurs within the scope of a `readOnly` method, we would have rewritten this declaration as `permit` r. Given the annotations in Figure 3, our type checker prevents reads from occurring outside the scope of a `readOnly` method.

The standard protocol for using sockets is a little more involved than our file reader. The code appears in figure 4. This example uses two effects, ac to mark the occurrence of an accept action and rv to mark the occurrence of a receive. There are two scoped

```
class ROFile {
  effect r;
  private void open(String path);
  private void close();
  Byte read() assert r;
  void readOnly(String path) permit r* {
    enter {this.open(path);}
    finally {this.close();}
  }
}
```

Figure 3: Read Only Files, Version 2

methods, one for the server and one for the receiver.

1.3 Singleton Types

Resource usage protocols often need to be enforced on every instantiation of the resource. For example, the type system should not allow clients to open a `readOnly` scope on `f1` but call the `read` method of `f2` instead, as the following code shows.

```
scope f1.readOnly(fn) {
  f2.read();
}
```

In order to prevent these sorts of errors, the type system must be able to tell different objects apart and trace them individually. Following the approach used by other groups [22, 5, 4], we use *singleton types* to differentiate between different objects with the same static class. A singleton type has the form C^l, where C is the class name of the object and l is the *static key* that is used to track object identity.

1.4 Synthesized Effects

In our file reader example, the implementer of the `ROFile` class annotated the read method with the effect r. We call such an annotation an *asserted effect.* Asserted effects may only appear on methods in the class where such effects are declared. Methods can also be annotated with *synthesized effects*, reflecting the fact that such methods emit effects by calling other effect-producing methods. For example, consider the `readLine` method defined below.

Socket implementation:

```
class Socket {
  effects  ac, rv;
  Socket ();
  private void bind (int localport);
  private void listen (int backlog);
  private void close();
  private void
    connect (InetAddress address, int port);
  void accept assert ac ();
  Bool receive assert rv (Buffer bf);
  void server (int localport, int backlog)
  permit ac·(rv)* {
    enter {bind(localport); listen(backlog);}
    finally {close();}
  }
  void client (InetAddress address, int port)
  permit (rv)* {
    enter {connect(address, port);}
    finally {close();}
  }
}
```

Client code:

```
let s=new Socket() in
  scope s.server(0xCAFE, 5) {
    s.accept();
    Bool flag = True;
    while (flag) {
      Buffer bf(1024);
      flag = s.receive(bf);
      ...
    }
}
```

Figure 4: UNIX Sockets

```
⟨l⟩ String readLine(ROFile^l f) ⇒{l : r·r*} {
  String result = "";
  Byte b = f.read();
  while (b != '\n'){
    result += b;
    b = f.read();
  }
  return result;
}
```

This method must be annotated with the effect $\{l : r{\cdot}r^*\}$, because it calls the `read` method of object l several times: once at the beginning of the method and zero or more times in the **while** loop.

The class implementer must annotate all methods with their proper synthesized effects. These annotations are checked for correctness by the type checker. A mechanism for type inference might relieve much of the annotation burden from programmers, but we leave an investigation of more sophisticated type inference techniques to future work.

The `readLine` example also demonstrates the use of key polymorphism. The notation $\langle l \rangle$ prior to method declaration introduces a polymorphic parameter that may appear in argument or result types. This *key polymorphism* enables the method to accept argument that corresponds to the `ROFile` class.

In the rest of the paper, we provide a more formal explanation of the concepts we have introduced above. Keep in mind that it is not our goal to produce the most expressive possible resource usage analysis. Rather, we seek to explain our key new ideas in a simple context and focus on the interaction between scoped methods and regular expression effects. Hence, when given a choice between an extra ounce of expressiveness and a spoonful of simplicity, we will normally choose simplicity.

Section 2 introduces the syntax of our language, which is a variant of Classic Java [6]. Section 3 gives a type system for statically checking correct usage of resources. Section 4 gives the operational semantics of the language by describing a virtual machine that monitors resource usage. Section 5 formulates type soundness theorems, which mean any well-typed program will not violate resource usage protocol. In section 6, we suggest a number of possible extensions to our work that will make it more practical. Finally, we discuss related work in section 7 and conclude in section 8.

$$
\begin{array}{rcl}
\tau & ::= & \mathbf{bool} \mid C \mid C^l \\
P & ::= & \overline{\mathit{defn}}\ e \\
\mathit{defn} & ::= & \mathbf{class}\ C\ \mathbf{extends}\ C \\
 & & \{\overline{\tau}\ \overline{f};\ \mathit{effs};\ \mathit{cons};\ \overline{\mathit{meth}};\ \overline{\mathit{smeth}}\} \\
\mathit{effs} & ::= & \mathbf{effects}\ \overline{p} \\
\mathit{cons} & ::= & C\ (\overline{\tau}\ \overline{x})\ \{e\} \\
\mathit{meth} & ::= & \tau\ m\ (\overline{\tau}\ \overline{x})\ \mathbf{assert}\ p\ \{e\} \\
 & \mid & \langle \overline{l} \rangle\ \tau\ m\ (\tau_{this}\ \mathbf{this}, \overline{\tau}\ \overline{x})\ \Rightarrow \Pi\ \{e\} \\
\mathit{smeth} & ::= & \tau\ m\ (\overline{\tau}\ \overline{x})\ \mathbf{permit}\ r \\
 & & \{\mathbf{enter}\ \{e\}\ \ \mathbf{finally}\ \{e\}\} \\
e & ::= & x \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid e.f \mid e.f = e \\
 & \mid & \mathbf{let}\ x = e\ \mathbf{in}\ e \\
 & \mid & \mathbf{let}\ x = \mathbf{new}\ C(\overline{e})\ \mathbf{in}\ e \\
 & \mid & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{while}\ (e)\ \{e\} \\
 & \mid & e.m(\overline{e}) \mid \mathbf{scope}\ e.m(\overline{e})\ \{e\} \\
r & ::= & \epsilon \mid p \mid r \cdot r \mid r \cup r \mid r^*
\end{array}
$$

Figure 5: Syntax

2 Syntax

The syntax of our language is presented in Figure 5. We use $C, D, \ldots$ as class names. We use $\overline{x}$ as a shorthand for $x_1, x_2, \cdots, x_n$ and $\overline{x}\ \overline{y}$ for $x_1\ y_1,\ x_2\ y_2,\ \cdots, x_n\ y_n$.

A type τ is either **bool**, a class name C, or a singleton type C^l. A program P is a sequence of class definitions followed by an expression. A class definition *defn* consists of five parts: a series of field definitions, a set of asserted effects managed by the class, a constructor definition, and finally a set of ordinary method and scoped method definitions.

A method m with asserted effect p has the form

$$\tau\ m\ (\overline{\tau}\ \overline{x})\ \mathbf{assert}\ p\ \{e\}$$

and its type is written as $\overline{\tau} \xrightarrow{p} \tau$.

A method m with synthesized effect is of the form

$$\langle \overline{l} \rangle\ \tau\ m\ (\tau_{this}\ \mathbf{this}, \overline{\tau}\ \overline{x})\ \Rightarrow \Pi\ \{e\}$$

The type of the method is written as $\forall \overline{l}.\tau_{this}, \overline{\tau} \xrightarrow{\Pi} \tau$. The universally quantified $\overline{l}$ is a set of keys that can be used in the argument and return types, which are of the form C or C^l, where l is one of the $\overline{l}$. Since **this** is an implicit argument in Java, it may also be associated with a key. The syntax shows its type explicitly as τ_{this}. The latent effect of the method is given by Π, which is a mapping from keys to a regular expressions.

A scoped method m is defined as

$$\tau\ m\ (\overline{\tau}\ \overline{x})\ \textbf{permit}\ r\ \{\textbf{enter}\ \{e_1\}\ \ \textbf{finally}\ \{e_2\}\}$$

where r is a regular expression describing the allowed resource usage protocol. The type of the scoped method is $\overline{\tau} \to \tau$.

Most aspects of the expression syntax are standard. We have **let** expression for introducing a local variable. A newly created object must be bound to a local variable to be used. We use $e_1; e_2$ as an abbreviation for **let** $x = e_1$ **in** e_2 if x is not free in e_2. We also have **if** and **while** expressions.

A **scope** expression is used to call a scoped method. It has the form **scope** $e_0.m(\overline{e})\ \{e_1\}$, where e_1 is the scope body.

$$\begin{array}{lcl} \Pi_1 \cdot \Pi_2 & = & \lambda l.\ (\Pi_1(l) \cdot \Pi_2(l)) \\ \Pi_1 \cup \Pi_2 & = & \lambda l.\ (\Pi_1(l) \cup \Pi_2(l)) \\ \Pi^* & = & \lambda l.\ (\Pi(l)^*) \\ \cdot\overline{\Pi} & = & \Pi_1 \cdot \Pi_2 \cdot \ldots \cdot \Pi_n \end{array}$$

$$\frac{\forall l.\ \Pi_1(l) \subseteq \Pi_2(l)}{\Pi_1 \subseteq \Pi_2}$$

Figure 6: Operations on Π

$$\frac{\textbf{class}\ D\ \textbf{extends}\ C}{D \le C} \qquad \frac{}{C^l \le C} \qquad \frac{D \le C}{D^l \le C^l}$$

$$\frac{}{\tau \le \tau} \qquad \frac{\tau_1 \le \tau_2 \qquad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}$$

Figure 7: subtype relation

3 Type System

3.1 Notation

Before continuing with a description of our type system, we must introduce a number of notational conventions. If X is a finite partial map, we write $\mathrm{dom}(X)$ for the domain of X. We write $X, l : v$ for the extension of X that maps l to v. If l already appears in X then $X, l : v$ is undefined. We use the notation $X[l \mapsto v]$ to update the map X (and allow X to be defined on l).

We use the metavariable Π to range over total mappings from keys to regular expressions. We lift operations on regular expressions (including concatenation, union and Kleene closure) to mappings as defined in Figure 6. We write $\{l \mapsto r\}$ as the map that associates l with r and associates all other keys with ϵ.

3.2 Subtype Relation

Our type system makes use of the subtyping relation defined in Figure 7. The only interesting point different from the ordinary Java subtyping relation is that the singleton type C^l is a subtype of C.

3.3 Expression Typing

The typing judgment for expressions has the form

$$\Lambda;\ \Gamma \vdash_{\mathsf{e}} e : \tau \Rightarrow \Pi$$

Λ is an environment that maps a key l to a pair (C, o), where C is the class of the object and o is the *in-scope bit*. When o is $\top$, the expression being checked is inside a scoped method for object l, otherwise o is $\bot$. Γ is an environment that maps a variable to its type. The mapping Π is the synthesized effect of the expression e.

To enforce correct resource usage, the typing rules ensure two invariants:

1. An effectful method can only be called inside the appropriate scope;
2. The effect a scope body emits is consistent with the effect permitted by the scoped method body.

Some auxiliary definitions are shown in Table 1. The complete typing rules for expressions are in Figure 8. We highlight some important ones below.

Key Creation In the *new* rule, an object of class C is created with a fresh key. The initial in-scope bit is set to $\bot$. The rule requires $\Lambda \vdash_{\mathtt{t}} \tau$, which specifies that τ is well-formed in the environment Λ:

$$\frac{\forall C, l.\ \tau = C^l \ \text{ implies } \ l \in \mathrm{dom}(\Lambda)}{\Lambda \vdash_{\mathtt{t}} \tau}$$

This condition ensures that the key of the newly created object does not escape the **let** expression.

If Expression The *if* rule shows how the effect of an expression is synthesized. Since expressions e_0, e_1 and e_2 produce effects Π_0, Π_1 and Π_2 respectively, the synthesized effect of expression **if** e_0 **then** e_1 **else** e_2 is $\Pi_0 \cdot (\Pi_1 \cup \Pi_2)$.

Scoped Method Call In the *scope* rule, a scoped method is called on object l_0. The synthesized effect of the scope body is checked against the specification ($\Pi_1(l_0) \subseteq r$), making sure the resource represented by l_0 is used correctly. The effect on l_0 that the **scope** expression exhibits to the outside world is empty because all the effects inside the scope are already checked and shouldn't be exposed.

The *scope* rule also checks that the original in-scope bit of l_0 is $\bot$. This is to rule out nested scopes on the same object, which will cause problems. Since the inner scope exhibits zero effect on the object, the outer-scope cannot capture the effect behaviour of the inner scope, violating the principle that all the effects in a scope body should be checked against the specification. On the other hand, nested scopes on different objects are OK. For example we can have a write-only scope for File 2 nested in a read-only scope for File 1 and copy File 1 to File 2.

The *grant* rule type checks the **grant** expression, which is not part of the static programming language, but instead is generated during evaluation of a **scope** expression.

Asserted-effect Method Call In this rule, the asserted effect $\{l \mapsto p\}$ is recorded in the synthesized effect. The rule requires that l's in-scope bit is $\top$, which means the method call must happen inside a scope body.

Synthesized-effect Method Call This rule makes use of the *compatibility relation* between Λ and Π. It is defined with respect to a set of keys $\bar{l}$:

$$\frac{\begin{array}{c}\forall l \in \bar{l}.\ \Lambda(l) = (\_\,, \bot) \ \ \mathit{iff}\ \ \Pi(l) = \epsilon. \\ \forall l \notin \bar{l}.\ \Pi(l) = \epsilon.\end{array}}{\Lambda \overset{\bar{l}}{\sim} \Pi}$$

In the *synthesized-effect method call* rule, we first find a set of *distinct* keys $\bar{l}'$ to instantiate the universally bound keys $\bar{l}$ in the method definition. $\bar{l}'$ is substituted into the types and effects(Π) of the method.

The requirement $\Lambda \overset{\bar{l}'}{\sim} \Pi[\bar{l}'/\bar{l}]$ makes sure that whenever a key's synthesized effect is non-empty, the in-scope bit of the key is $\top$, and whenever a key's synthesized effect is empty, the in-scope bit of the key is $\bot$. The rationale for this is:

1. if the object produces nonzero effect in the method, it must be called inside a scope.
2. if it produces zero effect, the only reason the method needs the key is to call a scoped method. Since nested scope is not allowed, the in-scope bit before entering the method must be $\bot$.

Location Typing The *loc* rule is to type check a memory location l, which is generated during evaluation when a new object is allocated and constructed. This location serves as the key for the object.

Polymorphic Method Typing The typing rules for synthesized-effect method definition is shown in Figure 9. The method body e is checked under Λ, which ranges over the universally quantified keys $\bar{l}$. Λ is compatible with Π and all the types should be well-formed under Λ. Since τ_{this} is the implicit **this** argument, the rule also requires that it is either C or C^l. The synthesized effect Π' of e is checked against the annotation Π.

Regular Expression Matching Our typing rules need an algorithm to decide whether a synthesized effect matches a specification. Since effects are made

$$\frac{}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} x : \Gamma(x) \Rightarrow \{\}}\ \textit{var} \qquad \frac{\Lambda(l) = (C,\ \_\ )}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} l : C^{l} \Rightarrow \{\}}\ \textit{loc} \qquad \frac{}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{null} : C \Rightarrow \{\}}\ \textit{null}$$

$$\frac{}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{true} : \mathbf{bool} \Rightarrow \{\}}\ \textit{true} \qquad \frac{}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{false} : \mathbf{bool} \Rightarrow \{\}}\ \textit{false}$$

$$\frac{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e : \tau \Rightarrow \Pi \qquad \textit{fieldty}(\tau, f) = \tau_f}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e.f : \tau_f \Rightarrow \Pi}\ \textit{get} \qquad \frac{\begin{array}{c}\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0 : \tau_0 \Rightarrow \Pi_0 \qquad \textit{fieldty}(\tau_0, f) = \tau_f \\ \Lambda;\ \Gamma \vdash_{\mathsf{e}} e_1 : \tau_1 \Rightarrow \Pi_1 \qquad \tau_1 \leq \tau_f\end{array}}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0.f = e_1 : \tau_1 \Rightarrow \Pi_0 \cdot \Pi_1}\ \textit{set}$$

$$\frac{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0 : \tau_0 \Rightarrow \Pi_0 \qquad \Lambda;\ \Gamma, x : \tau_0 \vdash_{\mathsf{e}} e_1 : \tau_1 \Rightarrow \Pi_1}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 : \tau_1 \Rightarrow \Pi_0 \cdot \Pi_1}\ \textit{let}$$

$$\frac{\begin{array}{c}\textit{ctype}(C) = \overline{\tau} \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} \overline{e} : \overline{\tau}' \Rightarrow \overline{\Pi} \qquad \overline{\tau}' \leq \overline{\tau} \\ \Lambda, l : (C, \bot);\ \Gamma, x : C^{l} \vdash_{\mathsf{e}} e : \tau' \Rightarrow \Pi \qquad \tau' \leq \tau \qquad \Lambda \vdash_{\mathtt{t}} \tau\end{array}}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{let}\ x = \mathbf{new}\ C(\overline{e})\ \mathbf{in}\ e : \tau \Rightarrow (\cdot\overline{\Pi}) \cdot \Pi}\ \textit{new}$$

$$\frac{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0 : \mathbf{bool} \Rightarrow \Pi_0 \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} e_1 : \tau_1 \Rightarrow \Pi_1 \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} e_2 : \tau_2 \Rightarrow \Pi_2 \qquad \tau_1 \leq \tau \qquad \tau_2 \leq \tau}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau \Rightarrow \Pi_0 \cdot (\Pi_1 \cup \Pi_2)}\ \textit{if}$$

$$\frac{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0 : \mathbf{bool} \Rightarrow \Pi_0 \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} e_1 : \tau \Rightarrow \Pi_1}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{while}\ (e_0)\ \{e_1\} : \textit{void} \Rightarrow \Pi_0 \cdot (\Pi_1 \cdot \Pi_0)^{*}}\ \textit{while}$$

$$\frac{\begin{array}{c}\Lambda;\ \Gamma \vdash_{\mathsf{e}} e : C^{l} \Rightarrow \Pi \qquad \Lambda(l) = (C, \top) \\ \textit{mtype}(C, m) = \overline{\tau} \xrightarrow{p} \tau \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} \overline{e} : \overline{\tau}' \Rightarrow \overline{\Pi} \qquad \overline{\tau}' \leq \overline{\tau}\end{array}}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e.m(\overline{e}) : \tau \Rightarrow \Pi \cdot (\cdot\overline{\Pi}) \cdot \{l \mapsto p\}}\ \textit{asserted-effect method call}$$

$$\frac{\begin{array}{c}\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0 : \tau_0 \Rightarrow \Pi_0 \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} \overline{e} : \overline{\tau}' \Rightarrow \overline{\Pi} \qquad \textit{mtype}(\tau_0, m) = \forall \overline{l}.\tau_{this}, \overline{\tau} \xrightarrow{\Pi} \tau \\ \exists \text{distinct } \overline{l}' \in \mathrm{dom}(\Lambda).\ \tau_0, \overline{\tau}' \leq (\tau_{this}, \overline{\tau})[\overline{l}'/\overline{l}] \text{ and } \Lambda \overset{\overline{l}'}{\sim} \Pi[\overline{l}'/\overline{l}]\end{array}}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0.m(\overline{e}) : \tau[\overline{l}'/\overline{l}] \Rightarrow \Pi_0 \cdot (\cdot\overline{\Pi}) \cdot (\Pi[\overline{l}'/\overline{l}])}\ \begin{array}{l}\textit{synthesized-effect} \\ \textit{method call}\end{array}$$

$$\frac{\begin{array}{c}\Lambda;\ \Gamma \vdash_{\mathsf{e}} e_0 : {C_0}^{l_0} \Rightarrow \Pi_0 \qquad \Lambda(l_0) = (C_0, \bot) \\ \textit{stype}(C_0, m) = \overline{\tau} \rightarrow \tau \qquad \Lambda;\ \Gamma \vdash_{\mathsf{e}} \overline{e} : \overline{\tau}' \Rightarrow \overline{\Pi} \qquad \overline{\tau}' \leq \overline{\tau} \\ \Lambda[l_0 \mapsto (C_0, \top)];\ \Gamma \vdash_{\mathsf{e}} e_1 : \tau_1 \Rightarrow \Pi_1 \qquad \textit{permit}(C_0, m) = r \qquad \Pi_1(l_0) \subseteq r \qquad \Pi_2 = \Pi_1[l_0 \mapsto \epsilon]\end{array}}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{scope}\ e_0.m(\overline{e})\ \{e_1\} : \tau \Rightarrow \Pi_0 \cdot (\cdot\overline{\Pi}) \cdot \Pi_2}\ \textit{scope}$$

$$\frac{\Lambda(l) = (C, \bot) \qquad \Lambda[l \mapsto (C, \top)];\ \Gamma \vdash_{\mathsf{e}} e : \tau \Rightarrow \Pi \qquad \Pi(l) \subseteq r}{\Lambda;\ \Gamma \vdash_{\mathsf{e}} \mathbf{grant}\ l.r\ \mathbf{in}\ e : \tau \Rightarrow \Pi[l \mapsto \epsilon]}\ \textit{grant}$$

Figure 8: Expression Typing

fieldty(τ, f), $\tau = C^l$ or C	The type of field f of class C
effects (τ)	All the effects of class C as defined in the **effects** declaration
ctype(τ)	Type of constructor of class C
cbody(τ)	Body of constructor of class C
mtype(τ, m)	Type of normal method m in class C
mbody(τ, m)	Body of normal method m in class C
emit(τ, m)	Effect that method m of class C emits as defined in the **emit** annotation ("none" if no annotation).
stype(τ, m)	Type of scope method m in class C
sbody(τ, m)	Body of scope method m in class C
permit(τ, m)	Allowable effect in the body of scope method m of class C as defined in the **permit** annotation.

Table 1: Auxiliary Definitions

$$\frac{\vdash_{\mathtt{c}} \overline{\mathit{defn}} \qquad \{\};\ \{\} \vdash_{\mathtt{e}} e : \tau \Rightarrow \{\}}{\vdash_{\mathtt{p}} \overline{\mathit{defn}}\ e : \tau}\ \textit{Program typing}$$

$$\frac{\{\} \vdash_{\mathtt{t}} \overline{\tau} \qquad \vdash_{\mathtt{ct}} \mathit{cons} \qquad \vdash^{C}_{\mathtt{m}} \overline{\mathit{meth}} \qquad \vdash^{C}_{\mathtt{s}} \overline{\mathit{smeth}}}{\vdash_{\mathtt{c}} \textbf{class}\ C\ \textbf{extends}\ D\ \{\overline{\tau}\ \overline{f};\ \mathit{effs};\ \mathit{cons}\ \overline{\mathit{meth}}\ \overline{\mathit{smeth}}\}}\ \textit{Class typing}$$

$$\frac{\{\};\ \overline{x} : \overline{\tau}, \textbf{this} : C \vdash_{\mathtt{e}} e : \tau \Rightarrow \{\}}{\vdash_{\mathtt{ct}} C\ (\overline{\tau}\ \overline{x})\ \{e\}}\ \textit{Constructor typing}$$

$$\frac{\begin{array}{c}\{\};\ \overline{x} : \overline{\tau}, \textbf{this} : C \vdash_{\mathtt{e}} e : \tau' \Rightarrow \{\} \qquad \tau' \leq \tau \\ \{\} \vdash_{\mathtt{t}} \tau,\ \overline{\tau} \qquad p \in \mathit{effects}(C)\end{array}}{\vdash^{C}_{\mathtt{m}} \tau\ m\ (\overline{\tau}\ \overline{x})\ \textbf{assert}\ p\ \{e\}}\ \textit{Asserted-effect method typing}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\Lambda) = \overline{l} \qquad \Lambda \overset{\overline{l}}{\sim} \Pi \qquad \Lambda \vdash_{\mathtt{t}} \tau,\ \tau_{this},\ \overline{\tau} \qquad \tau_{this} = C \text{ or } C^l \\ \Lambda;\ \textbf{this} : \tau_{this}, \overline{x} : \overline{\tau} \vdash_{\mathtt{e}} e : \tau' \Rightarrow \Pi' \qquad \tau' \leq \tau \qquad \Pi' \subseteq \Pi\end{array}}{\vdash^{C}_{\mathtt{m}} \langle \overline{l} \rangle\ \tau\ m\ (\tau_{this}\ \textbf{this},\ \overline{\tau}\ \overline{x})\ \Rightarrow \Pi\ \{e\}}\ \textit{Synthesized-effect method typing}$$

$$\frac{\begin{array}{c}\{\};\ \overline{x} : \overline{\tau}, \textbf{this} : C \vdash_{\mathtt{e}} e_1 : \tau_1 \Rightarrow \{\} \\ \{\};\ \overline{x} : \overline{\tau}, \textbf{this} : C \vdash_{\mathtt{e}} e_2 : \tau' \Rightarrow \{\} \qquad \tau' \leq \tau\end{array}}{\vdash^{C}_{\mathtt{s}} \tau\ m\ (\overline{\tau}\ \overline{x})\ \textbf{permit}\ r\ \{\textbf{enter}\ \{e_1\}\ \ \textbf{finally}\ \{e_2\}\}}\ \textit{Scoped-method typing}$$

Figure 9: Typing Rules (except for expressions).

$$
\begin{array}{rcl}
\textit{state} & = & (X, M, S, e) \\
X & = & \{l \ \mapsto\ r\} \\
M & = & \{l \ \mapsto\ (C, F, o)\} \quad o = \bot \ \textit{or} \ \top \\
S & = & \bullet \mid E \triangleright S \mid \textbf{check}\ l.r\ \textbf{at}\ S \\
e & = & \ldots \mid l \mid \textbf{grant}\ l.r\ \textbf{in}\ e \\
E & = & \Box.f \mid \Box.f = e \mid v.f = \Box \\
 & \mid & \textbf{let}\ x = \textbf{new}\ C\ (\overline{v}\ \Box\ \overline{e}\ )\ \textbf{in}\ e \\
 & \mid & \textbf{let}\ x = \Box\ \textbf{in}\ e \\
 & \mid & \textbf{if}\ \Box\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & \mid & \Box.m(\overline{e}) \mid v.m(\overline{v}\ \Box\ \overline{e}) \\
 & \mid & \textbf{scope}\ \Box.m(\overline{e})\ \{e\} \\
 & \mid & \textbf{scope}\ v.m(\overline{v}\ \Box\ \overline{e})\ \{e\} \\
v & = & l \mid \textbf{true} \mid \textbf{false} \mid \textbf{null}
\end{array}
$$

Figure 10: Machine state

up of regular expressions the problem reduces to deciding whether one regular expression is a subset of another, which is P-SPACE complete in the worst case [9]. There exist many implementations of regular expression operations [1].

Our type system is capable of checking that, for each object in the program, any effect that a scope body emits on the object satisfies the annotation of the scoped method, and all effectful methods are called within a scope body. So if a program type checks, it must use the resource according to the prescribed protocols. This is formalized as type safety theorems in section 5. Before discussing type safety, we first describe the virtual machine that serves as our evaluation model for the language.

4 Virtual Machine

Machine State A machine state is a quadruple (X, M, S, e), as shown in Figure 10. e is the current expression to be evaluated; S is the continuation stack; M is the memory containing all the created objects and X is the *effect tracer*.

The expression syntax is extended with a memory location l and a **grant** expression. We use memory locations as keys of objects. The expression **grant** $l.r$ **in** e means "allow effect r on object l in evaluating expression e". This expression does not appear in the surface syntax and is only generated when evaluating a **scope** expression.

The continuation stack is either an empty stack, a stack with E as the top frame, or a *check stack* in the form of **check** $l.r$ **at** S. The check stack serves as a mark where the effect on object l should be checked against specification r.

The memory maps a location l to a triple (C, F, o). C is the class of the object stored at this location; F contains all the fields of the object and o is the in-scope bit.

The effect tracer X is a mapping from an object l to a string r, which records the effects that have already happened on object l.

Dynamic Semantics We write $e[v/x]$ as the substitution of v for x in e. The complete evaluation rules are listed in Figure 11.

In the *new* rule, an object is created, a new memory location is allocated and all its fields are initialized to null. The initial in-scope bit is $\bot$.

A method call with asserted effect can only proceed if the current evaluation is in a scope. This is why the in-scope bit is checked in the *Asserted-effect method call* rule. The effect p is appended to the effect tracer X.

The evaluation rule for the synthesized-effect method call is a simple substitution. If the method has nonzero effect, its body will eventually evaluate to a point where some asserted-effect method is called and then the necessary check can be performed. Also note that the polymorphism of a method does not introduce any run-time behaviour. It is only used in type checking the program.

The evaluation of a **scope** expression yields a sequence of three expressions: an **enter** expression, a **grant** expression wrapping the scope body, and a **finally** expression.

The *grant* rule requires that the in-scope bit is $\bot$. This condition detects and rejects nested scopes on the same object. Then the object's in-scope bit is changed to $\top$ and a "check" mark is made on the continuation stack. The purpose of this mark is that when the scope body finishes its evaluation and the

[1] One can be found at http://www.brics.dk/automaton/

$(X, M, S, E[e]) \hookrightarrow (X, M, E \triangleright S,\ e)$ [*push*]

$(X, M, E \triangleright S, v) \hookrightarrow (X, M, S, E[v])$ [*pop*]

$(X, M, S,\ l.f) \hookrightarrow (X, M, S,\ v)$ [*get*]
`where` $M(l) = (\ \_\ , F,\ \_\ ),\ \ F(f) = v,$

$(X, M, S,\ l.f = v) \hookrightarrow (X, M[l \mapsto (C, F[f \mapsto v], o)], S, v)$ [*set*]
`where` $M(l) = (C, F, o)$

$(X, M, S,\ \textbf{let}\ x = \textbf{new}\ C(\overline{v})\ \textbf{in}\ e) \hookrightarrow (X, M', S, e_0[l/\textbf{this}, \overline{v}/\overline{x}];\ e[l/x])$ [*new*]
`where` $\mathit{cbody}(C) = C\ (\overline{\tau}\ \overline{x})\ \{e_0\},$
$M' = M,\ l : (C, \{\overline{f} \mapsto \textbf{null}\}, \bot)$

$(X, M, S,\ \textbf{if true then}\ e_1\ \textbf{else}\ e_2) \hookrightarrow (X, M, S, e_1)$ [*if_ true*]

$(X, M, S,\ \textbf{if false then}\ e_1\ \textbf{else}\ e_2) \hookrightarrow (X, M, S, e_2)$ [*if_ false*]

$(X, M, S,\ \textbf{while}\ e\ \{e_1\}) \hookrightarrow (X, M, S, e')$ [*while*]
`where` $e' =\ \textbf{if}\ e\ \textbf{then}\ \{e_1;\ \textbf{while}\ e\ \{e_1\}\}\ \textbf{else null}$

$(X, M, S, \textbf{let}\ x = v\ \textbf{in}\ e) \hookrightarrow (X, M, S, e[v/x])$ [*let*]

$(X, M, S, l.m(\overline{v})) \hookrightarrow (X', M, S, e[l/\textbf{this},\ \overline{v}/\overline{x}])$ [*asserted-effect method call*]
`where` $M(l) = (C,\ \_\ , \top),\quad \mathit{mbody}(C, m) = \tau\ m\ (\overline{\tau}\ \overline{x})\ \textbf{assert}\ p\ \{e\},$
$X(l) = r,\quad X'(l) = X[l \mapsto r \cdot p]$

$(X, M, S,\ l.m(\overline{v})) \hookrightarrow (X, M, S,\ e[l/\textbf{this},\ \overline{v}/\overline{x}])$ [*synthesized-effect method call*]
`where` $M(l) = (C,\ \_\ ,\ \_\ ),$
$\mathit{mbody}(C, m) = \langle \overline{l} \rangle\ \tau\ m\ (\tau_{this}\ \textbf{this},\ \overline{\tau}\ \overline{x}) \Rightarrow \Pi\ \{e\}$

$(X, M, S, \textbf{scope}\ l.m(\overline{v})\ \{e\}) \hookrightarrow (X, M, S, e')$ [*scope*]
`where` $M(l) = (C,\ \_\ ,\ \_\ ),$
$\mathit{sbody}(C, m) = \tau\ m\ (\overline{\tau}\ \overline{x})\ \textbf{permit}\ r\ \{\textbf{enter}\ \{e_0\}\ \ \textbf{finally}\ \{e_1\}\},$
$e' = e_0[l/\textbf{this},\ \overline{v}/\overline{x}];\ \textbf{grant}\ l.r\ \textbf{in}\ e;\ e_1[l/\textbf{this},\ \overline{v}/\overline{x}]$

$(X, M, S,\ \textbf{grant}\ l.r\ \textbf{in}\ e) \hookrightarrow (X, M',\ \textbf{check}\ l.r\ \textbf{at}\ S,\ e)$ [*grant*]
`where` $M(l) = (C, F, \bot),\quad M' = M[l \mapsto (C, F, \top)]$

$(X, M,\ \textbf{check}\ l.r\ \textbf{at}\ S,\ v) \hookrightarrow (X', M', S, v)$ [*check*]
`where` $X(l) = r',\qquad r' \subseteq r,\qquad X' = X[l \mapsto \epsilon],$
$M(l) = (C, F, \top),\qquad M' = M[l \mapsto (C, F, \bot)]$

Figure 11: Dynamic Semantics

stack returns to the mark, the effect that has happened on object l as recorded in X can be checked against specification r, which is shown in the *check* rule. The *check* rule also changes the object's in-scope bit back to $\perp$ because the control is about to leave the scope. The $X(l)$ is reset to empty, reflecting the fact that a scope expression shows zero effect to the outside world.

So far we have described a simple virtual machine that monitors effects and enforces that resources are used according to predefined protocols. The purpose of our type system is to *statically* enforce these protocols. In other words, a well-typed program should never get stuck dynamically when it runs on the virtual machine. In the following section we formalize this as type safety theorems.

5 Type Safety

Well-formed Machine States To formalize the well-formedness of a machine state we first define the well-formedness of memory, effect-tracer and continuation stack, as shown in Figure 12.

A memory M is well-formed with respect to a key environment Λ if each object's class and in-scope bit stored in M matches those in Λ, and all the fields are of appropriate types.

An effect-tracer X is well-formed with respect to Λ if whenever an object's in-scope bit is $\perp$, the outstanding effect of the object recorded in X should be empty.

The well-formedness judgment for continuation stacks has the following form.

$$\Lambda;\ \Pi\ \vdash\ S : \tau_1 \to \tau_2$$

Π is a mapping from keys l to regular expressions r, which describe the effect that has accumulated on object l before control jumps to S. For the stack (**check** $l.r$ **at** S) to be well-formed, the effect that has accumulated on l must satisfy the specification r.

A machine state (X, M, S, e) is well-formed if there exists Λ such that both X and M are well-formed under Λ, the expression e type checks and produces effect Π, and the stack S is well-formed with respect to effect $X \cdot \Pi$: when control jumps to S, e will have already been fully evaluated, so its effect should be appended to X to check stack S.

Final States In order to express our type safety result, we need to define the sensible *final states* of our virtual machine. For any X, M, S, v, m, or f, the following states are final.

1. $(X, M, \bullet, v)$
2. $(X, M, S,\ \textbf{null}.f)$, $(X, M, S,\ \textbf{null}.f = v)$
3. $(X, M, S,\ \textbf{null}.m(\overline{v}))$

Preservation and Progress The safety of our virtual machine is expressed through the following standard Preservation and Progress theorems. The proofs are straightforward due to careful organization of the operational semantics of our virtual machine.

Theorem 1 (Preservation) *If* $\vdash (X, M, S, e)\ \texttt{wf}$ *and* $(X, M, S, e) \hookrightarrow (X', M', S', e')$, *then* $\vdash (X', M', S', e')\ \texttt{wf}$

Theorem 2 (Progress) *If* $\vdash (X, M, S, e)\ \texttt{wf}$ *then either* (X, M, S, e) *is a final state or there exists* (X', M', S', e') *such that* $(X, M, S, e) \hookrightarrow (X', M', S', e')$.

6 Extensions

In this report, we have laid out the basic structure and soundness of scoped methods. However, integration of this programming construct into a realistic language will require a number of extensions to the basic theory. We discuss a few of them below.

Exceptions Exceptions are a heavily-used feature of object-oriented languages such as Java, and they cause problems for almost any static analysis. When an exception is raised in the middle of a scoped method, we would like to ensure that the appropriate protocol is completed. We plan to explore extensions to the **finally** clause in our scope methods, and more generally, the catch and finally clauses of Java, that

$$\frac{\mathrm{dom}(\Lambda) = \mathrm{dom}(M) \qquad \forall l.M(l) = (C, F, o) \text{ implies } \Lambda(l) = (C, o) \text{ and } \quad \forall f \in \mathrm{dom}(F).\ (\Lambda;\ \{\} \vdash_{\mathtt{e}} F(f) : \tau' \Rightarrow \{\} \text{ and } \mathit{fieldty}(C, f) = \tau) \text{ implies } \tau' \leq \tau}{\vdash M : \Lambda}$$

$$\frac{\forall l.\Lambda(l) = (\_\,, \bot) \text{ implies } X(l) = \epsilon}{\vdash X : \Lambda}$$

$$\frac{}{\Lambda;\ \{\} \vdash \bullet : \tau \to \tau}$$

$$\frac{\Lambda;\ x : \tau_1 \vdash_{\mathtt{e}} E[x] : \tau_2 \Rightarrow \Pi_2 \qquad \Lambda;\ \Pi \cdot \Pi_2 \vdash S : \tau_2 \to \tau_3}{\Lambda;\ \Pi \vdash E \triangleright S : \tau_1 \to \tau_3}$$

$$\frac{\Pi(l) \subseteq r \qquad \Lambda(l) = (C, \top) \qquad \Lambda[l \mapsto (C, \bot)];\ \Pi[l \mapsto \epsilon] \vdash S : \tau_1 \to \tau_2}{\Lambda;\ \Pi \vdash \textbf{check } l.r \textbf{ at } S : \tau_1 \to \tau_2}$$

$$\frac{\vdash X : \Lambda \qquad \vdash M : \Lambda \qquad \Lambda;\ \{\} \vdash_{\mathtt{e}} e : \tau_1 \Rightarrow \Pi \qquad \Lambda;\ X \cdot \Pi \vdash S : \tau_1 \to \tau_2}{\vdash (X, M, S, e)\ \mathtt{wf}}$$

Figure 12: Well-formed machine states

allow us to specify program behavior in the case that exceptions cause protocols to be aborted early.

Concurrency Concurrency primitives are another common object-oriented programming feature that cause difficulties for static analysis. We must prevent concurrent threads from interfering with each other's protocols. It seems likely that we will need to use dynamic mechanisms to prevent such interference.

Extended anti-aliasing constraints We use a simple form of syntactic control of interference [15, 16] to rule out aliasing of keys. Such a simplistic scheme is unlikely to work effectively in practice, but we believe we will be able to use previous research to lift these restrictions.

For example, we could easily extend our anti-aliasing constraints by moving to a more flexible linear or quasi-linear type system [20, 13, 17, 23, 4]. We chose to use the simplest anti-aliasing scheme possible since the aliasing problem is essentially orthogonal to the main concerns of the paper.

Existential types over keys There are two limitations of our system that can be solved if we had existential types over keys: keys cannot escape the **let** expression, which further prohibits its escape over function boundaries; a class field cannot track the identity, key, of its value. However, in order to add a general form of existential types, we would also need to add linear types to control the copying of existentials. See Walker and Morrisett [23] and Deline and Fahndrich [4] for examples of how these two features interact.

Extended specification languages There are plenty of protocols that cannot be expressed accurately using our regular expression effects. For example, regular expressions cannot specify that resources be used in a stack-like fashion. Our effects are also not powerful enough to express binary, or more generally, n-ary relations between resources. However, more expressive grammars also make it more difficult to test inclusion of the languages that they generate.

7 Related work

This research continues the study of type and effect systems which started in the mid-'80s and early

’90s with the foundational work of Gifford and Lucassen [7] and later Jouvelot, Talpin, Tofte and others [12, 19]. More recently, we have seen simple sets of effects evolve into more sophisticated effect specifications. For example, Igarashi and Kobayashi [11] use complex effects to specify the protocol for using a resource.

There are several differences between Igarashi and Kobayashi’s resource usage analysis and our own. First, in their system, values have types that contain resource usage specifications directly whereas we associate resource usage specifications indirectly with values using the combination of singleton types and regular expression effects. This level of indirection provides a convenient, precise, and relatively simple way to track identity and local aliasing of values. Second, since usages are associated directly with types and type declarations, their analysis propagates information top-down rather than synthesizing it bottom up via effects. Third, they give a very abstract characterization of the actual language for specifying usages whereas we have made our resource usage language concrete (we use regular expressions).

There have been a number of other research efforts on effect systems for object-oriented languages. For example, Greenhouse and Boyland [8] describe an effect system for Java that allows subclasses to extend the effects of their superclass. They show how to apply their effect system to aliasing problems rather than enforcement of object protocols. The ownership types of Clark, Potter and Noble [2] are another sort of effect system. Ownership types help prevent an object from revealing the implementation of its internal data structures by ensuring that internal pointers are not passed out to clients.

Another closely related project is Deline and Fähndrich’s Vault language [4]. Vault’s type system is built on top of the capability calculus [22]. Both use a similar sort of “key” to track the identity of resources, and the state of a resource determines which operations can be performed next. One difference between Vault and our scoped-method approach lies in the specification of usage protocols. In Vault, the annotations of all functions in an interface collectively define a resource protocol whereas we have centralized protocol specification at the definition of a scoped method. The advantage of centralized specification is easy maintenance and extension. It is less likely that non-obvious errors in the annotation break the whole protocol. To add a new usage protocol, one only needs to add a new scoped method instead of changing annotations of every method in the class.

Extended Static Checking (ESC) [14] addresses a broad range of programming errors from array index bounds errors to locking protocol violations to null pointer dereferences. By focusing on the problem of resource usage protocols, we hope to develop a specification language that allows simple, concise and comprehensible specifications in this domain. If we succeed, our simple sublanguage might be added to a more general program checking tool such as ESC.

8 Conclusions and Future Work

We have proposed a new language construct, the *scoped method*, which encapsulate resource usage protocols inside a resource module. We have also designed a type system to statically enforce such protocols and have proven the soundness of the type system.

We have modeled some simple resource protocols involving files, locks and sockets using scoped methods to validate our approach. The next step in this research involves tackling some of the more advanced features of Java and developing an implementation to test our results.

Acknowledgments

We would like to thank the FOOL 10 reviewers for their insightful comments on a previous version of this paper.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison Wesley, 2000.

[2] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, Kyoto, April 1998.

[3] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL' 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275. ACM Press, January 1999.

[4] Robert Deline and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 59–69, June 2001.

[5] Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming*, pages 91–108, 1999.

[6] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL' 98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1998.

[7] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986.

[8] Aaron Greenhouse and John Boyland. An object-oriented effect system. In *ECOOP*, number 1628 in LNCS, pages 205–229. Springer-Verlag, 1999.

[9] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[10] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, 1999.

[11] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *POPL '02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342. ACM Press, January 2002.

[12] Pierre Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *EighteenthACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[13] Naoki Kobayashi. Quasi-linear types. In *Twenty-SixthACM Symposium on Principles of Programming Languages*, pages 29–42, San Antonio, January 1999.

[14] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Proceedings of the Schloss Dagstuhl tenth-anniversary conference*, volume 2000 of Springer LNCS, 2001.

[15] John C. Reynolds. Syntactic control of interference. In *FifthACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, 1978.

[16] John C. Reynolds. Syntactic control of interference, part 2. In *Sixteenth International Colloquium on Automata, Languages, and Programming*, July 1989.

[17] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

[18] Bjarne Stroustrup. What is "object-oriented programming"? In *ECOOP '87: European Conference on Object-Oriented Programming*, Paris (France), 1987. Springer-Verlag, Lecture Notes in Computer Science no. 276.

[19] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[20] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[21] Philip Wadler. Is there a use for linear logic? In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991.

[22] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, May 2000.

[23] David Walker and Greg Morrisett. Alias types for recursive data structures. In Robert Harper, editor, *Workshop on Types in Compilation*, number 2071 in LNCS, Montreal, March 2001.

[24] David Walker and Kevin Watkins. On regions and linear types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, 2001.

[25] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.