Ownership Downgrading for Ownership Types

Yi Lu, John Potter, and Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, Sydney
{ylu,potter,jingling}@cse.unsw.edu.au

Abstract. Ownership types support information hiding by providing object-based encapsulation. However the static restrictions they impose on object accessibility can limit the expressiveness of ownership types. In order to deal with real applications, it is sometimes necessary to admit mechanisms for dynamically exposing otherwise encapsulated information. The need for policies and mechanisms to control such information flow, known as downgrading or declassification, has been well covered in the security literature.

This paper proposes a flexible ownership type system for object-level access control. It still maintains privacy of owned data, but allows information to be dynamically exposed where appropriate through an explicit declassification operation. The key innovation is an owners-as-downgraders policy, implemented via a simple language construct, which allows an object to be made more widely accessible by downgrading its ownership to its owner's owner.

1 Introduction

Traditional class-level private declarations in Java-like programming languages are inadequate for hiding object instances or restricting access to them. For instance, the underlying object in a private field can be accidentally exposed by assignments, calls or method returns. *Ownership types* [26, 11, 10] are a widely accepted technique for providing object-based encapsulation by partitioning all objects into a structure called the *ownership tree*. In such type systems, the *owners-as-dominators* property guarantees that an object is only accessible within the encapsulation provided by its owner. This prevents unwanted representation exposure and protects against deliberate attacks.

This form of strong encapsulation is too inflexible in practice; for example, it impedes the use of common coding idioms and design patterns such as iterators and observers. A number of proposals have been made for improving the expressiveness of ownership types (as reviewed in Section 5). They explore ways to increase object accessibility by exposing otherwise encapsulated objects. While these approaches vary in their detail, they all require object exposure to be statically determined, being fixed at the time of object creation. Hence object accessibility is fixed for the lifetime of the object.

However, to deal with real applications, it is sometimes necessary to admit mechanisms for dynamically exposing otherwise encapsulated information. A long-standing problem in information security is how to specify and enforce expressive security policies that hide information while also permitting information release (called downgrading or declassification) where appropriate [7].

In this paper, we present a new ownership type system for object-level access control. While it still protects object instances like other ownership type systems do, it can allow programmers to intentionally expose encapsulated objects at runtime through an explicit declassification operation. Compared to other ownership type systems where an object's accessibility is statically and solely determined by its creator, our type system allows object accessibility to be changed dynamically by its *dynamic owner*. Compared to other downgrading policies in information security (where downgrading is controlled based on code authority [25] or conditions [7] or actions [19]), our type system enforces a downgrading policy based on object ownership, called *owners-as-downgraders*. The owners-as-downgraders property highlights what is a natural role for an owner—access to an object must be sufficiently authorized by its owners. With the ability to encapsulate objects as well as to expose them, we provide a more flexible and expressive ownership type system for object access control, which allows programmers to express design intent more precisely.

The paper is organized as follows: Section 2 gives a brief introduction to ownership types. Section 3 provides an informal overview of our proposed model. An example is presented with a detailed description to illustrate the use of our type system. Section 4 presents a core object-oriented language to allow us to formalize the static semantics, dynamic semantics and some important properties. Section 5 discusses related work, and Section 6 briefly concludes the paper.

2 Object Encapsulation with Static Ownership

We first give a brief overview of the classic approach to *static ownership* and before introducing our concept of *dynamic ownership* in Section 3.

Earlier object encapsulation systems, such as *Islands* [16] and *Balloons* [3], use full encapsulation techniques to forbid both incoming and outgoing references to an object's internal representation. Ownership types [11, 10, 8] provide a more flexible mechanism than previous systems; instead of full encapsulation, they allow outgoing references while still restricting incoming ones. The work on ownership types emanated from some general principles for *Flexible Alias Protection* [26] and the use of dominator trees in structuring object graphs [28].

In ownership type systems, each object has one fixed owning object, called the owner or *static owner*, bound statically at object creation. Often, the term *context* is used in the ownership types literature [11, 9] to refer to ownership parameters in types. In the following example, the first context parameter `o` of class `Author` is the owner of an author object. Other context parameters such as `p` are optional; they are used to type references outside the current encapsulation. Types are formed by binding actual context parameters to the formal parameters of a class. In the following example, the owner of `book` can be declared to be the current object/context **this**.

```
class Author<o,p> {
  Book<this> book;
  Date<p> deadline;
}
// client code
Author<owner,world> author = new Author<owner,world>();
... = author.book; // error
```

Ownership partitions all objects in the heap into an *ownership tree*, rooted at the special context **world**. Ownership types support information hiding by providing statically enforceable object encapsulation based on the ownership tree. The key technical mechanism used here is the hiding of object identity; ownership type systems ensure that the identity of an object can only be propagated (via class parameters) to its owned objects. In order to declare a type for a reference, one must be able to identify the owner that encapsulates the object, so that objects outside an encapsulation boundary can never identify the owner of encapsulated objects. The **world** context is globally nameable, so that objects owned by **world** are accessible from anywhere in the program.

For example, in the `Author` class, the author may want to hide his book from unauthorized access. In Java, this can be done by declaring the `book` field to be private. However such a class-level private declaration does not prevent the underlying object instance from being referenced from outside, because a reference to the object may easily be leaked by assignments to fields of other objects, bindings to calls, or method returns. In ownership types, the owner of `book` can be declared to be the current context **this**, which is different from object to object. It is impossible to name **this** to give the correct type of an author's `book` from outside. Thus, any attempted access (reference) to the book from client code via the expression `author.book` is a type error.

Ownership types can be considered as an access control system where owners are security levels for object instances. The **world** context is the lowest security level, so that accesses to objects owned by **world** are not restricted. However, unlike pre-defined security levels in conventional security systems, owners are also normal objects (except **world**) which may be owned by other objects. The following *owners-as-dominators* property guarantees that any access from outside to an object's internal representation must be made via the object's owner.

Property 1 (Static Encapsulation)
If object A accesses object B then $A \preceq$ owner(B).

This property states that an object can only be accessed from within its static owner. The function *owner(B)* denotes the static owner of B. The symbol $\preceq$, often called *inside* (or *dominated by*), denotes the reflexive and transitive closure of the ownership relation. This property implies that both accessibility for an object (the set of objects that can reference it) and its reference capability (the set of objects it can reference) is determined by the object's position in the ownership tree. This, in turn is defined by the object's static owner, which remains fixed for its lifetime. This fixed ownership has limited utility in applications that require a more flexible access control policy, such as when information needs to be released at runtime.

3 Object Exposure with Dynamic Ownership

In this section, we give an informal introduction to our type system. We first introduce our concept of dynamic ownership in Section 3.1, and then explain how to safely expose objects using dynamic ownership in Section 3.2. Finally, we illustrate our type system with an example, showing that it is more flexible and expressive than static ownership types.

3.1 Dynamic Encapsulation

In order to support a more flexible and dynamic access policy, we introduce a concept of dynamic ownership which separates an object's *dynamic owner* from its static owner. This separation enables the object's dynamic owner to be changed (downgraded) at runtime while its static owner remains fixed after creation. Like previous ownership types, the static owner of an object defines its fixed position in the ownership tree, hence determines the capability of the object. Unlike previous ownership types, object accessibility is determined by its dynamic owner rather than static owner. Thus the accessibility of an object may be changed at runtime by downgrading its dynamic ownership. Here is a simple illustration.

```
class Author<o> {
  [this]Book<o> book = new Book<o>();
}
```

Each type in our system is a (classic) ownership type prefixed with a dynamic owner. The dynamic owner of the variable `book` is the current context `[`**this**`]`. The dynamic owner of a new object is not explicitly specified, because the initial dynamic owner is always the same as its static owner; in this example the implicit dynamic owner for the new book is `o`, the static owner of the new book (and of the current author).

The assignment on variable `book` being OK implies `[o]Book<o>` is a subtype of `[`**this**`]Book<o>`. This subtyping is safe because it never exposes objects; the set of objects that can access **this** is a subset of the objects that can access `o`. In this example, the new object becomes less accessible when it is assigned to the variable `book` (our type system requires **this** to be inside `o`). This is typical of access control mechanisms in security applications where it is safe to increase the security level by restricting the number of subjects that may access an object. We used a similar mechanism in previous work [20] on the variance of object access modifiers. In this paper, we focus on how to expose objects using dynamic ownership. We now state the encapsulation property for our new system with dynamic ownership.

Property 2 (Dynamic Encapsulation)
If object A accesses object B then $A \preceq$ downer(B).

The function *downer(B)* denotes the dynamic owner of B. This property is more general than static encapsulation, because the dynamic ownership of an object can be changed at runtime by controlled downgrading, which will be discussed in the next section.

3.2 Dynamic Exposure

An object may be exposed by downgrading its dynamic ownership, i.e. promoting its dynamic owner up the ownership tree so the object becomes more accessible. Like most information security systems [7], in our system downgrading only occurs via an explicit `expose` operation in order to avoid accidental leaking of sensitive information. Below is a simple example of the `expose` operation.

```
class Author<o> {
  [this]Book<this> bookInWriting;
  [o]Book<*> release() {
    ...
    return expose bookInWriting;
  }
}
```

The reference in the field `bookInWriting` is encapsulated within the current object as suggested by its dynamic owner (`[this]`). Classic ownership type systems would never allow such a reference to be leaked outside. However, in our type system, we allow programmers to release this reference via the explicit expose operation (for example, the author may wish to release the book after he has finished writing it). In the above code, **expose** `bookInWriting` downgrades the dynamic ownership of the book reference by promoting its dynamic owner from **this** to `o` (we know `o` owns **this**). This enables the reference to be accessed by the owner of the current object or other objects within it.

The `*` symbol (similar to Java's `?` wildcard for generics) can be thought of as an abstract owner; they have become common in ownership-based type systems [21, 20, 6]. Abstract owners are useful when the actual owners are unknown to the current context. Only context arguments (including the static owner) of a type may be abstract; dynamic owners can never be abstract as they determine if the object is accessible in the current context. In the example, the return type of the method `release` is `[o]Book<*>`, which implies that the receiver of the method call only needs to know `o` in order to access the released book (without having to know its static owner).

The following property states the effect of object exposure. An expose operation allows the exposed object to be accessed from within the downgrader's owner.

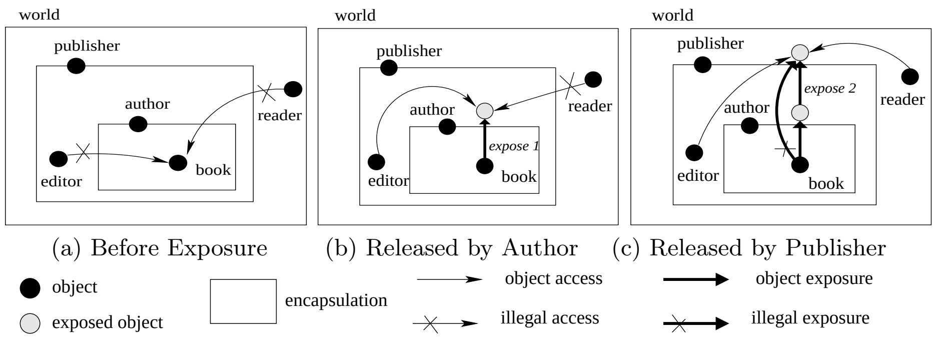
Property 3 (Object Exposure)
If object A has exposed object B, then owner(A) $\preceq$ downer(B).

Not surprisingly, our type system requires that the dynamic owner of the exposed object to be known to the downgrader (which is always the current object). This means that the object to be exposed is dynamically owned by the downgrader or the downgrader's owners. The static owner of an exposed object is not restricted by the type system; typically it can be abstracted by $*$. Note that, since the capability of an object is determined by its static ownership, its capability won't be affected by any downgrading. Moreover, an `expose` operation can promote an object's dynamic ownership by at most one level up the ownership tree. For an object to be exposed through more than one level, it must

```
class Publisher<o> {
 Author<this> author;
 Editor<this> editor;
 [this]Book book;
 [o]Book release() {
   book = author.release();
   editor.edit(book);
   ...
   if (finishedEditing(book))
     return expose book;
 }
}
```

```
class Author<p> {
 Book<this> book;
 [p]Book release() {
   ...
   if (finishedWriting(book))
     return expose book;
 }
}
// a reader client
Publisher<world> publisher;
[world]Book book;
book = publisher.release();
read(book);
```

Fig. 1. A Publisher Example



(a) Before Exposure (b) Released by Author (c) Released by Publisher

Fig. 2. Step by Step Exposure

go through a sequence of expose operations, authorized by a sequence of downgraders up an ownership branch. That *B is a downgrader of A* means either B has exposed A via an explicit expose operation, or B has the authority to expose A (i.e. it is the current dynamic owner of A). The *owners-as-downgraders* property states our authorization policy on object exposure.

Property 4 (Owners-as-Downgraders)
For two objects A and B, if owner(A) $\preceq$ B $\preceq$ downer(A), then B must be a downgrader of A.

Since *owner(A)* and *downer(A)* are identical when A is created, we know A has been exposed if *downer(A)* is outside *owner(A)*. That is, the object has become more accessible than was allowed by its creator. If A is exposed, then all A's transitive owners up to *downer(A)* (exclusively) must have authorized the exposure via explicit expose operations.

Our type system strictly subsumes classic ownership types; the owners-as-dominators property is the same owners-as-downgraders without downgrading. In other words, ownership programs are special cases of our programs where no expose operation is used; in that case, the dynamic owner of every object is fixed, remaining identical to its static owner.

3.3 A Publisher Example

In this section, we sketch a scenario for book publishing which involves multiple steps of exposure of an object. The sample code, given in Figure 1, is kept simple, to highlight the idea of ownership downgrading. Figure 2 illustrates the steps of exposure and the change of object accessibility in the system after each exposure.

In this example, there are two top-level objects, a publisher and a reader. The reader reads books published by the publisher. Inside the publisher, there are an author and an editor, whose jobs are, respectively, writing and reviewing books for the publisher. A book object is owned by the author. Their ownership relation is fixed as suggested by their static owners. But the accessibility of the book changes at runtime due to the use of object exposure.

In the code, we use some defaults for declaring types. Context arguments of a type may be omitted if they are all abstract; for instance, `[p]Book` is a shorthand for `[p]Book<*>`. Also, the dynamic owner of a type may be omitted if it is the same as the static owner (which must not be abstract); for instance, `Author<`**`this`**`>` is a shorthand for `[`**`this`**`]Author<`**`this`**`>`.

In the class `Author` the book is owned by the author, initially it cannot be accessed from outside the author. In Figure 2(a), the book object is encapsulated inside the author's context, so that accesses on it from the editor or the reader are prevented by the type system (as shown by crossed arrow lines in the figure). However, the author may release the book to the publisher (its owner) after he finishes writing it. An expose operation is used in the `release()` method to expose the book to the publisher. The book becomes dynamically owned by the publisher as suggested by the dynamic owner of the return type `[p]Book`.

After the first exposure, the book object is accessible within the publisher: it can be accessed by the editor but not the reader, as shown in Figure 2(b). The editor can now do some editing on the book. The second exposure is defined in the `Publisher` class. The publisher will release the book after the editor finishes editing it. Another expose operation is used by the publisher to expose the book; the book becomes dynamically owned by the **world**. After the second exposure, the book object is accessible by any object in the program so that the reader can start reading the book now, as shown in Figure 2(c).

In our type system, it is necessary to expose the book twice in order to release the book to the reader. Our owners-as-downgraders policy ensures that such exposure must be authorized (via the explicit expose operations) by both the author and the publisher. The crossed thick line in Figure 2(c) indicates that it is not possible to directly expose the book to the reader in a single step.

The ability to encapsulate objects and expose them allows our programs to express useful aspects of designer intent. The publisher example shows that we can use object exposure to enforce a logical sequence of actions in a certain order. This is like an assured pipeline in information flow security. When the author is writing the book, he does not want any interference from outside, so the book object is kept private by the author until writing is finished. After the book is released to the publisher, the editor may edit the book in collaboration with the author. The book remains private to the publisher until editing is finished. The reader may not read the book, until the publisher publishes it. In practice, as the book is exposed, the type of the reference should be restricted to a more limited interface, e.g. to prevent readers editing the book.

T	$::=$	$[\mathsf{D}]\mathsf{O}$	variable types
O	$::=$	$\mathsf{C}\langle\overline{\mathsf{K}}\rangle$	object types
K, D	$::=$	$\mathsf{X} \mid \mathsf{this} \mid \mathsf{world} \mid * \mid ? \mid e$	contexts
P	$::=$	$\overline{\mathsf{L}}\ e$	programs
L	$::=$	$\mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle \lhd \mathsf{O}\ \{\overline{\mathsf{T}\ \mathsf{f}};\ \overline{\mathsf{M}}\}$	classes
M	$::=$	$\mathsf{T}\ m(\overline{\mathsf{T}\ x})\ \{e\}$	methods
e	$::=$	$x \mid \mathsf{new}\ \mathsf{O} \mid e.\mathsf{f} \mid e.\mathsf{f} = e \mid e.m(\overline{e}) \mid \mathsf{expose}\ e$	expressions
Γ	$::=$	$\bullet \mid \Gamma, \mathsf{X} \preceq \mathsf{X} \mid \Gamma, x : \mathsf{T}$	environments

Table 1. Abstract Syntax for Source Language and Type System

4 The Formal System

In this section, we present a core object-oriented language which allows us to formalize the type system, semantics and the main results of the type system. We show that our owners-as-downgraders property can be checked syntactically by a type system. The abstract syntax and typing rules follow the conventions used in our previous papers [20, 22], which extend *Featherweight Java* [17] with field assignment and ownership information.

Table 1 gives the abstract syntax of the source language, with the extensions used by the type system having grey background as they are not available to programmers. The extensions include the syntax for type environments and existential contexts. Type environments Γ record the assumed ownership ordering between context parameters, and the types of variables. Existential contexts are used by the type system to name contexts which are hidden from the current environment. The expression e may be a concrete context only if e is this, otherwise e is an existential context (hence not a well-formed context by the rules in Table 4). The special existential context ? represents a concrete unknown context: an anonymous Skolemisation; it is only used in type instantiation in class member lookup (see `[L-DEF]` in Table 7) to prevent unsound binding. Types with existential context ? have restricted *bindability* (see `[T-SUB]` in Table 3), because ? is not a well-formed context in the current environment (see `[A-ID]`). Conventional existential types use some form of pack/unpack or close/open pairing to distinguish between typing contexts where the existential type is visible or not. Our use of the context abstraction $*$ in the language syntax and the existential context ? in the type system is somewhat akin to the use of pack/unpack mechanisms for existential types, but simpler. In particular, we avoid introducing new names for contexts into environments by keeping them anonymous. The details of existential contexts can be found in [20]; we do not discuss them in full detail in this paper.

4.1 Static Semantics

The judgements used in the type system are give in Table 2, together with their meanings. Table 3 provides rules for type well-formedness, subtyping rules for object (value) types and variable types, as well as the rules for context abstraction. Directly borrowed from [20], we use a separate judgement for bindability to handle types with existential contexts, which only occur after field or method lookup

Judgement	Meaning
$\Gamma \vdash \mathsf{O}$	O is a well-formed object type
$\vdash \mathsf{O} < \mathsf{O}'$	O is a subtype of O'
$\Gamma \vdash \mathsf{T}$	T is a well-formed variable type
$\Gamma \vdash \mathsf{T} <: \mathsf{T}'$	an expression of type T is bindable to a variable of type T'
$\Gamma \vdash \mathsf{K} \ll \mathsf{K}'$	K can be abstracted by K'
$\Gamma \vdash \mathsf{K}$	K is a well-formed context
$\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'$	K is inside K'
$\vdash \mathsf{P}$	P is a well-formed program
$\vdash \mathsf{L}$	L is a well-formed class
$\Gamma \vdash \mathsf{M}$	M is a well-formed method
$\Gamma \vdash_{\mathsf{K}} e : \mathsf{T}$	expression e is of type T
$\mathsf{H}; e \Downarrow_{\mathsf{K}} \mathsf{H}'; e'$	e is reduced to e' in context K, and heap H is updated to be H'

Table 2. Judgements in Type System

$$\texttt{[OBJECT]}\quad \frac{\begin{array}{c}|\overline{\mathsf{K}}| = arity(\mathsf{C}) \qquad \Gamma \vdash \overline{\mathsf{K}} \\ \overline{\mathsf{K}} = \mathsf{K}, ... \qquad \Gamma \vdash \mathsf{K} \preceq \overline{\mathsf{K}}\end{array}}{\Gamma \vdash \mathsf{C}\langle\overline{\mathsf{K}}\rangle}$$

$$\texttt{[O-SUB]}\quad \frac{\mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle \lhd \mathsf{O}\ ...}{\vdash \mathsf{C}\langle\overline{\mathsf{K}}\rangle < \mathsf{O}[\overline{\mathsf{K}/\mathsf{X}}]}$$

$$\texttt{[O-RFL]}\quad \frac{}{\vdash \mathsf{O} < \mathsf{O}}$$

$$\texttt{[O-TRA]}\quad \frac{\vdash \mathsf{O} < \mathsf{O}'' \qquad \vdash \mathsf{O}'' < \mathsf{O}'}{\vdash \mathsf{O} < \mathsf{O}'}$$

$$\texttt{[TYPE]}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{D} \qquad \Gamma \vdash \mathsf{O}' \\ \Gamma \vdash [\mathsf{D}]\mathsf{O}' <: [\mathsf{D}]\mathsf{O}\end{array}}{\Gamma \vdash [\mathsf{D}]\mathsf{O}}$$

$$\texttt{[T-SUB]}\quad \frac{\Gamma \vdash \overline{\mathsf{K} \ll \mathsf{K}'} \qquad \Gamma \vdash \mathsf{D}' \preceq \mathsf{D}}{\Gamma \vdash [\mathsf{D}]\mathsf{C}\langle\overline{\mathsf{K}}\rangle <: [\mathsf{D}']\mathsf{C}\langle\overline{\mathsf{K}'}\rangle}$$

$$\texttt{[T-TRA]}\quad \frac{\begin{array}{c}\vdash \mathsf{O} < \mathsf{O}'' \\ \Gamma \vdash [\mathsf{D}]\mathsf{O}'' <: [\mathsf{D}']\mathsf{O}'\end{array}}{\Gamma \vdash [\mathsf{D}]\mathsf{O} <: [\mathsf{D}']\mathsf{O}'}$$

$$\texttt{[ABSTRACT]}\quad \frac{}{\Gamma \vdash \mathsf{K} \ll *}$$

$$\texttt{[A-ID]}\quad \frac{\Gamma \vdash \mathsf{K}}{\Gamma \vdash \mathsf{K} \ll \mathsf{K}}$$

Table 3. Types, Subtype, Binding and Abstraction Rules

by the type system (see Table 7). The well-formedness [TYPE] rule is somewhat unconventional; it allows the introduction of context abstraction $*$ into types, via the bindability requirement. Well-formed object types, [OBJECT], only use concrete contexts; by requiring the owner context of an object type to be inside any other context parameters, we ensure that any type accessed from within has an owner that contains the current object, as required for the encapsulation property. [A-ID] ensures that the existential contexts abstract nothing, because they are not valid contexts by [CONTEXT] in Table 4. Combined with [T-SUB] this ensures that bindability is not reflexive, because existential types can only occur on the right-hand side of the binding relation. The use of bindability in typing field assignment and method argument binding (see [E-ASS], [E-CAL] in Table 6) prevents existential contexts from being associated with the target of a binding. Substitutions used in this type system are postfixed in order to be distinct from the prefixed dynamic owners. For instance, $\mathsf{O}[\overline{\mathsf{K}/\mathsf{X}}]$ substitutes $\overline{\mathsf{K}}$ for $\overline{\mathsf{X}}$ in O.

Table 4 defines well-formed contexts and context ordering for concrete contexts. Direct ownership is captured in [C-OWN] by looking up the static owner

[Context] $$\frac{\mathsf{K} \in \Gamma}{\Gamma \vdash \mathsf{K}}$$

[World] $$\frac{}{\Gamma \vdash \mathsf{world}}$$

[C-own] $$\frac{}{\Gamma \vdash \mathsf{K} \preceq \mathit{owner}_{\Gamma}(\mathsf{K})}$$

[C-env] $$\frac{\mathsf{K} \preceq \mathsf{K}' \in \Gamma}{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'}$$

[C-wld] $$\frac{}{\Gamma \vdash \mathsf{K} \preceq \mathsf{world}}$$

[C-tra] $$\frac{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'' \qquad \Gamma \vdash \mathsf{K}'' \preceq \mathsf{K}'}{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'}$$

[C-rfl] $$\frac{\Gamma \vdash \mathsf{K}}{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}}$$

Table 4. Context and Inside Rules

[Program] $$\frac{\vdash \overline{\mathsf{L}} \qquad \bullet \vdash e : \mathsf{T}}{\vdash \overline{\mathsf{L}}\ e}$$

[Class] $$\frac{\begin{array}{c}\overline{\mathsf{X}} = \mathsf{X}, \ldots \qquad \Gamma = \mathsf{X} \preceq \overline{\mathsf{X}}, \mathsf{this} : [\mathsf{X}]\mathsf{C}\langle\overline{\mathsf{X}}\rangle, \mathsf{super} : [\mathsf{X}]\mathsf{O} \qquad \Gamma \vdash [\mathsf{X}]\mathsf{O}, \overline{\mathsf{T}} \\ \Gamma \vdash \overline{\mathsf{M}} \qquad \mathit{owner}_{\Gamma}(\mathsf{this}) = \mathit{owner}_{\Gamma}(\mathsf{super}) \qquad \overline{\mathsf{f}} \cap \mathit{dom}(\mathit{flds}_{\Gamma}(\mathsf{super})) = \bullet\end{array}}{\mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle \lhd \mathsf{O}\ \{\overline{\mathsf{T}\ \mathsf{f}};\ \overline{\mathsf{M}}\}}$$

[Method] $$\frac{\begin{array}{c}\Gamma, \overline{x : \mathsf{T}} \vdash_{\mathsf{this}} e : \mathsf{T}'' \qquad \Gamma \vdash \mathsf{T}, \overline{\mathsf{T}} \qquad \Gamma \vdash \mathsf{T}'' <: \mathsf{T} \\ \mathit{mth}(\Gamma(\mathsf{super}), \mathsf{this}, m) = \mathsf{T}'\ \overline{\mathsf{T}'} \quad \Longrightarrow \quad \Gamma \vdash \mathsf{T} <: \mathsf{T}' \qquad \Gamma \vdash \overline{\mathsf{T}' <: \mathsf{T}}\end{array}}{\Gamma \vdash \mathsf{T}\ m(\overline{\mathsf{T}\ x})}$$

Table 5. Program, Class and Method Rules

from the type of the context via [L-own]. The only direct ownership relation available in static semantics is for the this context; it is owned by the first context parameter of its type (see [Class]). this is the only context that is given a static type; at runtime, this is bound to the location of the target object.

The rules for well-formed program definitions and declarations are in Table 5. In the [Class] rule, each class defines its own environment formed from its formal contexts and the type of this and super objects. The owner parameter (the first context parameter X) has to be inside all other context parameters, just like the classic ownership types [11, 9]. The only direct ownership relation known to the class, that is this $\preceq$ X, is not included in the class environment; instead, we capture it in the [C-own] rule to make it generally derivable. A bullet symbol is used to denote something null or empty. For instance, in $\bullet \vdash e : \mathsf{T}$ in [Program], the environment of the expression does not exist. For simplicity, we often simply omit the bullets.

Table 6 defines expression types. Each expression judgement is attached with a context ($\vdash_{\mathsf{K}}$), which is the current context where the expression is evaluated (the target object of the current call). In the static semantics, it is bound to the this context in class methods (see [Method]); in the dynamic semantics, it is bound to the location of the current object. It is omitted in [Program], because the main method does not have a target object; because world context is never used as a target object. The most interesting typing rule is [E-xpo], where an object may be exposed. Since a dynamic owner is only authorized to expose an object it directly owns, this rule insists an object can only be exposed, by promoting its dynamic owner one level up in the ownership tree, if its dynamic owner is the current context. Otherwise, there is no exposure.

$$[\texttt{E-VAR}]\quad \frac{}{\Gamma \vdash_K x : \Gamma(x)}$$

$$[\texttt{E-NEW}]\quad \frac{\Gamma \vdash O \qquad O = C\langle D, ...\rangle}{\Gamma \vdash_K \mathsf{new}\ O : [D]O}$$

$$[\texttt{E-SEL}]\quad \frac{flds^K_\Gamma(e)(f) = T \qquad \Gamma \vdash T}{\Gamma \vdash_K e.f : T}$$

$$[\texttt{E-ASS}]\quad \frac{\Gamma \vdash_K e.f : T \qquad \Gamma \vdash_K e' : T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash_K e.f = e' : T}$$

$$[\texttt{E-CAL}]\quad \frac{mth^K_\Gamma(e, m) = T\ \overline{T}\ ... \qquad \Gamma \vdash_K \overline{e : T'} \qquad \Gamma \vdash \overline{T' <: T} \qquad \Gamma \vdash T}{\Gamma \vdash_K e.m(\overline{e}) : T}$$

$$[\texttt{E-XPO}]\quad \frac{\begin{array}{c}\Gamma \vdash_K e : [D]O \qquad D = K \implies D' = owner_\Gamma(D) \\ D \neq K \implies D' = D\end{array}}{\Gamma \vdash_K \mathsf{expose}\ e : [D']O}$$

Table 6. Expression Rules

$$[\texttt{L-OWN}]\quad \frac{\Gamma \vdash_K e : [D]C\langle K', ..\rangle}{owner^K_\Gamma(e) = K'}$$

$$[\texttt{L-FLD}']\quad \frac{def(O, K) = ... \lhd O'\ \{\overline{T\ f};\ ...\}}{flds(O, K) = \overline{f\ T}, flds(O', K)}$$

$$[\texttt{L-DYN}]\quad \frac{\Gamma \vdash_K e : [D]O}{downer^K_\Gamma(e) = D}$$

$$[\texttt{L-MTH}]\quad \frac{\Gamma \vdash_K e : [D]O}{mth^K_\Gamma(e, m) = mth(O, e, m)}$$

$$[\texttt{L-DEF}]\quad \frac{\begin{array}{c}L = \mathsf{class}\ C\langle\overline{X}\rangle ... \qquad \overline{K'} = \overline{K}[?/*] \\ L' = L[\overline{K'/X}, K/\mathsf{this}]\end{array}}{def(C\langle\overline{K}\rangle, K) = L'}$$

$$[\texttt{L-MTH}']\quad \frac{def(O, K) = ...\ T\ m(\overline{T\ x})\ \{e\}\ ...}{mth(O, K, m) = T\ \overline{T}\ \overline{x}\ \ e}$$

$$[\texttt{L-FLD}]\quad \frac{\Gamma \vdash_K e : [D]O}{flds^K_\Gamma(e) = flds(O, e)}$$

$$[\texttt{L-MTH}'']\quad \frac{\begin{array}{c}def(O, K) = ... \lhd O'\ \{...\ ; \overline{M}\} \\ m \notin \overline{M}\end{array}}{mth(O, K, m) = mth(O', K, m)}$$

Table 7. Auxiliary Definitions for Lookup

Table 7 defines auxiliary definitions to be used by the type system. We put substitutions and lookups in the auxiliary definitions to keep the type rules simple. When accessing the fields or methods via an expression e, we determine their types, given the type of e. These in turn use [L-DEF] to find a correct substitution for parameters of T's class, where $*$ is replaced by ?. This is similar to the usual unpack/open for conventional existential types. The major difference is that we do not introduce fresh context variables into the current environment. Instead, we keep the existential context anonymous by using a special symbol ?. This technique eliminates the need for the pack/close operation, since anonymous contexts do not have to be bound to an environment, they naturally become global (see [20] for more details).

4.2 Dynamic Semantics and Properties

The extended syntax and features used by the dynamic semantics are given in Table 8. Ownership information, including both static and dynamic owners, is only needed in static type checking so that they may be erased after type checking. The expose operations may be erased too, because they only affect dynamic owners; $\mathsf{expose}\ e$ becomes just e after erasure. However, in order to formalize the key properties of the type system, we need to establish a connection

$l, l_O^{\overline{D}}$		locations	$o ::= \overline{f \mapsto l}$	objects
e	$::= \ldots \mid l$	expressions	$H ::= \overline{l \mapsto o}$	heaps

Table 8. Extended Syntax with Dynamic Features

$$[\texttt{E-LOC}]\quad \frac{}{\Gamma \vdash_K l_O^{\overline{D},D} : [D]O} \qquad\qquad [\textsc{Heap}]\quad \frac{\begin{array}{c}\forall l \in dom(H) \cdot \\ H(l) = \overline{f \mapsto l} \qquad flds(l) = \overline{f\ T} \\ \vdash \overline{l : T'} \qquad \vdash \overline{T' <: T}\end{array}}{\vdash H}$$

$$[\texttt{L-XPO}]\quad \frac{}{downgraders(l_O^{\overline{D}}) = \overline{D}}$$

Table 9. Auxiliary Definitions for Dynamic Semantics

between the static and dynamic semantics by including ownership and exposures in the dynamic semantics.

Locations are annotated with the type of the object they refer to, as well as a list of dynamic owners which records the history of exposures of this object. The dynamic owners of a location may be extended over time through exposures. The list is used to prove Theorem 4. The last element in the dynamic owners list of a location is the current dynamic owner of the object, i.e., the downgrader who currently has the authority to expose the object. The advantage of not storing the type information in the heap is that we can look up type information directly from the location itself without referring to the heap (see [E-LOC]). All locations are annotated with their owners and type; but we may omit them wherever that information is not used.

Expressions are extended with locations, which indirectly extends contexts with locations. A heap is a mapping from locations to objects; an object maps its fields to locations. A few auxiliary definitions are included in Table 9 to help formalize the properties. We have used some shorthand for simplicity. $\Gamma \vdash e <: T$ means $\Gamma \vdash_K e : T'$ and $\Gamma \vdash T' <: T$. We may omit $\bullet$ in judgements and lookup functions; for example, $\vdash_K e : T$ means $\bullet \vdash_K e : T$. We may also omit the current context K where it is not used.

The dynamic semantics are defined in a big step fashion in Table 10. The context K in $\Downarrow_K$ refers to the target object of the current call; the main method does not have a target object. At the time of method invocation in [R-CAL], the target object of the body of the invoked method is l. The variable this is not substituted in $e'[\overline{l/x}]$. Instead, this is replaced by l in the substitution provided by the lookup function $mth(l, m)$. In [R-XPO], if the current object K is the same as the last dynamic owner of l (i.e, K is the current authorizing downgrader for l), then the exposure is authorized and the dynamic owner of l is promoted to $owner(D)$. The exposure is completed by an appending of $owner(D)$ to the end of the dynamic owner list that is attached to the location of the object. Otherwise, the exposure has insufficient authorization; the object l and its dynamic owner list remains unchanged.

The operational semantics for field operations are obvious. In [R-NEW], to create an object, we adopt the default field initialization scheme from MOJO [6]. We first create a new object at a fresh address in the heap; then we initialize the fields with default object creation. Object creation is a recursive process in an atomic step. MOJO's default field initialization simplifies the formalism

$$\texttt{[R-SEL]} \quad \frac{H; e \Downarrow_K H'; l}{H; e.f \Downarrow_K H'; H'(l)(f)}$$

$$\texttt{[R-NEW]} \quad \frac{\begin{array}{c} l \notin dom(H) \qquad H_1 = H, l^D_O \mapsto \bullet \qquad \overline{f\ T} = flds(l) \\ \forall i \in 1..|\overline{f}| \quad \cdot \quad H_i; \mathsf{new}\ T_i \Downarrow_K H_{i+1}; l_i \\ O = C\langle D, ...\rangle \qquad H' = H_{|\overline{f}|+1}[l^D_O \mapsto \overline{f \mapsto l}] \end{array}}{H; \mathsf{new}\ O \Downarrow_K H'; l}$$

$$\texttt{[R-ASS]} \quad \frac{H; e \Downarrow_K H'; l \qquad H'; e' \Downarrow_l H''; l'}{H; e.f = e' \Downarrow_K H''[l \mapsto H''(l)[f \mapsto l']]; l'}$$

$$\texttt{[R-CAL]} \quad \frac{\begin{array}{c} H; e \Downarrow_K H_1; l \qquad mth(l, m) = ...\overline{x}\ e' \\ \forall i \in 1..|\overline{x}| \quad \cdot \quad H_i; e_i \Downarrow_K H_{i+1}; l_i \\ H_{|\overline{x}|+1}; e'[\overline{l/x}] \Downarrow_l H'; l' \end{array}}{H; e.m(\overline{e}) \Downarrow_K H'; l'}$$

$$\texttt{[R-XPO]} \quad \frac{\begin{array}{c} H; e \Downarrow_K H'; l \qquad \vdash_K l : [D]O \qquad D \neq K \implies H'' = H' \\ D = K \implies H'' = H'[l_O^{downgraders(l), owner(D)}/l] \end{array}}{H; \mathsf{expose}\ e \Downarrow_K H''; l}$$

Table 10. A Big Step Semantics

because it avoid nulls; otherwise we would need to allow for null errors. In this paper, we focus on the correctness of object exposure, rather than concerning ourselves with a more complex, but realistic model for object initialization.

Finally, we formalize some of the key properties of the type system, including a standard type preservation result in Theorem 1. Theorems 2-4 directly correspond to the properties introduced informally in Section 3.

Theorem 1 (Preservation) *Given* $\vdash H$ *and* $\vdash_K e : T$*, If* $H; e \Downarrow_K H'; l$ *then* $\vdash H'$ *and* $\vdash l <: T$.

Theorem 2 (Encapsulation)
Given $\vdash H$ *and* $\vdash_K e : T$*, if* $H; e \Downarrow_K H'; l$ *then* $\vdash K \preceq downer(l)$.

Theorem 3 (Exposure)
Given $\vdash H$ *and* $\vdash_K \mathsf{expose}\ e : T$*, if* $H; \mathsf{expose}\ e \Downarrow_K H'; l$*, then* $\vdash owner(K) \preceq downer(l)$.

Theorem 4 (Owners-as-downgraders)
Given $\vdash H$ *and* $l, l' \in H$*, if* $\vdash owner(l) \preceq l' \preceq downer(l)$ *then* $l' \in downgraders(l)$.

5 Discussion and Related Work

In this paper, we have focused on techniques for exposing objects safely at runtime in a statically typed language. Any object may become accessible to any other object, given sufficient downgrading. Our type system ensures that owners must explicitly authorize downgrading; this highlights the role of the owner – only the owner can authorize the objects it owns to be accessed from outside. Like security types with downgrading [30, 25, 7, 19], our type system does not track downgrading information. In other words, we cannot tell, from the types, whether an object has-been-exposed or may-be-exposed. However, in many applications where stronger static assumptions are needed (for instance, program verification and reasoning about object invariants), we may still want to retain fixed object ownership so that certain objects can never be exposed. A simple extension to our system would allow us to model (ownership-traditional) never-to-be-exposed objects in addition to may-be-exposed objects, but that is not a concern of this paper. Never-to-be-exposed objects are just a special case of may-be-exposed objects where downgrading is not allowed.

Our formalization is built upon classic tree-based ownership types [11, 10, 9] for simplicity. Modern extensions to ownership types have been studying richer structures than per-context ownership trees. For instance, *Ownership Domains* [1] allow programmers to further partition contexts into domains for finer-grained access control, and MOJO [6] replaces ownership tree with multiple ownership for more precise reasoning about effects. We believe that the object exposure technique introduced in this paper could be applied in those systems too.

Syntactic overhead for our types is that of ownership types plus an extra dynamic owner for each type. As we have seen in our earlier example, with carefully selected defaults type annotations can be reduced. Moreover, the ideas of *Generic Ownership* [27] can also be employed here to reduce the burden of type annotations in the presence of class type parameters. As for other ownership type systems, ours allows separate compilation. It is statically checkable and does not require any runtime support.

There have been a variety of ownership type systems seeking to improve the expressiveness of the classic ownership types by relaxing the owners-as-dominators property. JOE [9] allows internal contexts to be exposed through read-only local variables. Ownership Domains [1] uses final fields to expose internal contexts. Lexical scoping has been used to give inner class instances special access privileges to their outer objects [8, 5]. Both SOT [10] and [20] offer separation of accessibility and capability of an object. Such separation allows the creator of an object to decide its accessibility independently from its position in the ownership tree, and also implies object access need not be authorized by its owner. All these techniques increase nameability of internal contexts, but object accessibility remains statically determined, being fixed at creation for the object's lifetime. None of these approaches can dynamically expose an encapsulated object at runtime as in this paper.

We have focused on object instance access control with object ownership. Beside access control, the idea of object ownership has been proved to be useful in other applications. *OOFX* [14], JOE [9] and *MOJO* [6] combine effects and some forms of ownership. These systems focus on reasoning about read and write effects in order to syntactically control interference in the program. *Boogie* [18], *Universes* [13], *Effective Ownership* [21] and *Oval* [22] use the ownership structure to confine object dependency in order to support localized reasoning about object invariants. These systems typically do not hide information as they do not restrict read access or reference; object invariants can only be violated by mutations. They enforce an owners-as-modifiers property.

Ownership transfer has been supported in a number of systems which permits the owner of an object to be changed when necessary. However, they either cannot be enforced statically, or have to enforce some form of uniqueness of reference. *Islands* [16] uses strict unique references with destructive reads to permit transfer. *External uniqueness* [12] is a flexible form of uniqueness for ownership types which permits internal sharing without compromising uniqueness of external reference; it supports ownership transfer by combining external uniqueness with destructive reads and borrowing. *AliasJava* [2] supports ownership transfer using destructive field reads and lent variables. Beside these type systems, *UUT* [24] combines a type system and static analysis to enforce temporary uniqueness for ownership transfer. Program logics, such as Boogie [4, 18, 23], can allow dynamic ownership, but require program verification to check ownership properties.

Our owners-as-downgraders property essentially enforces a downgrading policy based on object ownership. Unlike most security types for secure information flow [7, 19], our type system does not consider covert channels [30], hence we do not enforce a conventional *noninterference* property (we could have done so, but it is not a concern of this paper). Our downgrading policy may be considered as a special case of intransitive noninterference [15, 29], where special downgrading paths exist in a security lattice. With our owners-as-downgraders property, such downgrading paths are upward branches in the ownership tree.

6 Conclusion

In this paper, we have proposed an expressive ownership type system, which encapsulates object privacy while also permitting information release where appropriate. In this new system, objects may expose their owned information via an explicit operation. We have shown an example where object encapsulation and exposure capture useful aspects of designer intent. With the ability to change object accessibility at runtime, we have provided a more dynamic and expressive model for object access control in ownership types.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 17th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–330, 2002.
3. P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.
4. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. In S. Eisenbach, G. T. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs (FTfJP)*, July 2003. Published as Technical Report 408 from ETH Zurich.
5. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–223. ACM Press, 2003.

6. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *OOPSLA*, October 2007.
7. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, 2004.
8. D. Clarke. *Object Ownership and Containment.* PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
9. D. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *OOPSLA*, 2002.
10. D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.
11. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
12. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, July 2003.
13. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005.
14. A. Greenhouse and J. Boyland. An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 205–229. Springer-Verlag, 1999.
15. J. T. Haigh and W. D. Young. Extending the noninterference version of mls for sat. *IEEE Trans. on Software Engineering, SE-13(2)*, pages 141–150, 1987.
16. J. Hogg. Islands: aliasing protection in object-oriented languages. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.
17. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, pages 132–146, 1999.
18. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.
19. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.
20. Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP*, pages 99–123. Springer-Verlag, 2006.
21. Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *POPL*. ACM Press, 2006.
22. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *ECOOP*. Springer-Verlag, 2007.
23. Microsoft Research. *Towards a Verifying Compiler: The Spec# Approach*, 2006.
24. P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478, 2007.
25. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
26. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, 1998.
27. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, 2006.
28. J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *ASWEC*. IEEE Press, 1998.
29. A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *CSFW*, pages 228–238, 1999.
30. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1)*, 2003.