On Ownership and Accessibility

Yi Lu and John Potter

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

Abstract. Ownership types support information hiding by providing statically enforceable object encapsulation based on an ownership tree. However ownership type systems impose fixed ownership and an inflexible access policy. This paper proposes a novel type system which generalizes ownership types by separating object accessibility and reference capability. With the ability to hide owners, it provides a more flexible and useful model of object ownership.

1 Introduction

The object-oriented community is paying increasing attention to techniques for object level encapsulation and alias protection. Formal techniques for modular verification of programs at the level of objects are being developed hand in hand with type systems and static analysis techniques for restricting the structure of runtime object graphs. Ownership type systems have provided a sound basis for such structural restrictions by being able to statically represent an extensible object ownership hierarchy. The trick to ownership systems is to hide knowledge of the identity of an object outside its owner. This form of information hiding is useful for modular reasoning, data abstraction and confidentiality.

Ownership types support instance-level information hiding by providing a statically enforceable object encapsulation model based on an ownership tree. Traditional class-level private fields are not enough to hide object instances. For example, an object in a private field can be easily returned through a method call. However, the encapsulation mechanism used by ownership types is still not flexible enough to express some common design patterns such as iterators and callback objects. Moreover, ownership types, to date, lack ownership variance. This means, for instance, that all elements stored in a list must be owned by the same owner due to the recursive structure of the list.

This paper proposes a novel type system which generalizes ownership types with an access control system in which *object accessibility* and *reference capability* are treated orthogonally. The rationale behind this mechanism is that one only needs to hold access permission for an object in order to use it; the capability of the object can be adapted to the current access context. This allows more flexible and expressive programming with ownership types.

Our system allows programmers to trade off flexibility/accessibility with useability/capability. We allow object accessibility to be variant; intuitively it is

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 99–123, 2006.

safe to allow accessibility to be reduced as computation proceeds. On the other hand, we allow reference capability associated with an object to be abstracted in contexts where it is used. Our resulting type system is flexible enough to encode iterator and callback-like design patterns. It also allows objects, such as recursive data structures, to hold references to elements owned by different objects.

This paper is organized as follows: Section 2 gives an introduction to object encapsulation and the mechanisms used in ownership types; it also discusses the limitations of ownership types. Section 3 proposes the variant ownership object model and its key mechanisms with some program examples. Section 4 presents a small object-oriented programming language to allow us to formalize the static semantics, dynamic semantics and some important properties. Section 5 follows with discussion and related work. Section 6 briefly concludes the paper.

2 Ownership Types

Earlier object encapsulation systems, such as *Islands* [13] and *Balloons* [2], use full encapsulation techniques to forbid both incoming and outgoing references to an object's representation. However, full encapsulation techniques are overly strong, because outgoing references from the representation are harmless and are often needed to express typical object-oriented idioms. Ownership types [11, 10, 7] provide a more flexible mechanism than previous systems; they weaken the restriction of full encapsulation by allowing outgoing references while still preventing representation exposure from outside of the encapsulation. The work on ownership types emanated from some general principles for *Flexible Alias Protection* [20] and the use of dominator trees in structuring object graphs [22].

Ownership type systems establish a fixed per object ownership tree, and enforce a reference containment invariant, so that objects cannot be referenced from outside of their owner — an object owns its representation and any other external object wanting to access the representation must do so via the owner. Ownership types are parameterized by names of runtime objects, called *context*s in the type system [10]. Contexts include all objects and a pre-defined context called world which is used to name the root of the ownership tree. The world context is in scope throughout the program. All root objects are created in the world context and all live objects are reachable from the root objects.

In defining ownership the first context parameter of a class is the owner of the object. An object's owner is fixed for its lifetime, thus naturally establishing an *ownership tree* in the heap. The rest of the context parameters are optional; they are used to type and reference the objects outside the current encapsulation, which is how ownership types free objects from being fully encapsulated, as in early approaches to alias protection.

Types are formed by binding the formal context parameters of a class. In order to declare a type for a reference, one must be able to name the owner that encapsulates the object. An encapsulation is protected from incoming references because the owners of objects inside the encapsulation cannot be named from the outside. The key mechanism is the use of variable this to name the current

object (or context), which is different from object to object — it is impossible to name `this` in an object from outside of the object. Only objects inside the current object can name the this context, because the name of this can only be propagated as context arguments for objects owned by this object.

Since an object can only name its dominators (either direct or indirect owners) using formal context parameters, the pre-defined context world or the local context this, it can never declare a correct type for any object not owned by these contexts. In the following simple example, the private `Data` object is in the `this` context. Since it is impossible to name the `this` variable in a `Personnel` object from outside, it is impossible to give a correct type for the private `Data` object and reference it.

```
class Personnel<o> {
  Data<this> privateData;
  Data<this> getData() { return privateData; } }
```

We now highlight two problems with the standard ownership example of a linked list in which the linked nodes are protected within their list owner.

```
class List<o, d> {
  Node<this, o> head;
  Iterator<this, d> getIter() { return new Iterator<this, d>(head); } }

class Node<o, d> {
  Node<o, d> next;
  Data<d> data;
  Node(Node<o, d> next, Data<d> data) {
    this.next = next; this.data = data; } }

class Iterator<o, d> {
  Node<o, d> current;
  Iterator(Node<o, d> node) { current = node; }
  Data<d> element() { return current.data; }
  void next() { current = current.next; }
  void add(Data<d> data) {
    current.next = new Node<o, d>(current.next, data); } }

class Data<o> { void useMe(){ ... } }
```

A list object is implemented by a sequence of linked node objects. The node objects form the representation of the list object, in other words, they are owned by the list object. The owner of the node objects is the `this` context in the `List` class which refers to the current list object. The `List` class provides iterator objects to be used by the client to read the elements stored by the list or add new elements.

The first problem for iterators with ownership is well known. Iterator objects need to be able to reference the internal data representation of the list in order to traverse it efficiently. In ownership types, this requires that iterators

are owned by the list, living within the list's internal representation. The problem is obvious — iterators cannot be referenced from outside of the list due to the encapsulation property. In the client code given below, `list` owns the iterator object returned by `list.getIter()`, but `iter` cannot be declared as `Iterator<list,o>`, because `list` is not a constant.

```
class Client<o> {
  void m() {
    List<this, o> list = new List<this, o>(); // OK
    Iterator<this, o> iter = list.getIter();  // ERROR, owner mismatch
    iter.add(new Data<o>());                  // OK
    iter.add(new Data<world>());    // ERROR, owner of Data mismatch
    iter.add(new Data<this>());     // ERROR, owner of Data mismatch
    iter.element().useMe(); } }               // OK
```

The second problem is a less well-known expressiveness problem due to the recursive nature of the `Node` class. All `Data` objects stored in the list must have the same type; in particular they must be owned by the same context. In the above example, the client can only add objects with type `Data<o>` into the list.

For the first problem with iterators, a number of solutions have been proposed (to be discussed in Section 5). Compared to these, our proposal is more flexible and somewhat less ad hoc. To solve the second problem, we employ a powerful mechanism which allows ownership contexts to be abstract or variant (that is, appearing as an abstraction with bounds) while still maintaining enough control on object access. The resulting type system is statically type checkable and more expressive than previous ownership type systems.

3 Variant Ownership Types

In this section, we give an informal overview of *variant ownership types* which are more flexible and expressive than ownership types. The two new concepts involved are the *accessibility context* and *context variance*.

3.1 The Accessibility Context

We separate the access permissions for an object from the definition of its class, by adding another context to ownership types (in addition to the normal context arguments) as an access modifier, which alone determines its accessibility. In comparison to conventional class-level field access modifiers, such as *public/protected/private* as used in Java and C++, our system provides instance-level object access modifiers that are dynamic contexts. This extra context is specified by the creator of the object in its creation type, and controls the accessibility of the object. Only objects which can name the access modifier have access permission. In the owners-as-dominators model, owners control access, but in our model, accessors do not need to have the owner in scope. For a given ownership hierarchy, our new approach has more flexibility, and strictly subsumes the owners-as-dominators model. In ownership types, if object `l` can

reference object $\mathtt{l}'$ then $\mathtt{l}$ must be inside $\mathit{owner}(\mathtt{l}')$. In our type system, we use a separate context $\mathtt{acc}(\mathtt{l})$ to determine the accessibility of an object; if object $\mathtt{l}$ can reference object $\mathtt{l}'$ then $\mathtt{l}$ must be inside $\mathtt{acc}(\mathtt{l}')$. This is the *accessibility invariant* for our system.

A typical type consists of three parts: $[\mathtt{access}]\ \mathtt{Class}\langle\mathtt{capabilities}\rangle$ comprising a class name with a list of contexts. In a type, the capability list binds the formal parameters of the type's class definition to actual contexts; the access modifier context (the prefix in square brackets) restricts accessibility to objects inside the given context.

Class definitions are parameterized by formal contexts. These formal parameters, together with `this` and `world`, define the contexts that are available for types within the class. Note that a class definition does not have a formal parameter for denoting the accessibility of its instances; there is no need.

Access modifiers are independent from the definition of their objects, that is, the access modifier is an annotation on a type rather than a context parameter in the class definition. Note that, in our type system, the ownership tree is still built from the owner context (the first context argument of a type). The modifier does not affect the ownership tree, instead, it generalizes the strong containment invariant of ownership, allowing more general reference structures.

We could use access modifiers to provide various levels of protection on objects. When an object's access modifier is the world context, it can be considered as a public object which is accessible by any other object. When an object's access modifier is the owner of the defining object (i.e., the owner parameter of the current class), it is partially protected — it can be only used by those within the owner context. When an object's access modifier is the defining object (i.e., the this context), it is private to the defining object and cannot be accessed from outside the defining object.

In the following example, variable `a` has `world` accessibility and owner `this`. We interpret this dynamically. Variable `a` is a field of the current `B` object that may hold references to `A` objects; any such `A` object must be inside (owned by) the current `B` object `this` because the owner is given as `this`. Having `world` accessibility implies that the referenced object can be used by any other object via `B`'s `a` field. The new expression creates a new `A` object owned by the current `B` object with `world` accessibility. Such an object may safely be assigned to the `a` field of the current object, but not to other `B` objects.

```
class A<o> { ... }

class B<o> {
  [world] A<this> a;
  m() { a = new [world]A<this>; } }
```

Access modifiers allow indirect exposure of internal states in a controlled manner. Objects in variable `a` can act as interface objects or proxy objects between the internal objects of the current `B` object and accessing objects on the outside. However, the type system ensures these interface objects cannot directly expose the internal objects to the accessing objects on the outside. This breaks the

strong owners-as-dominators containment invariant enforced in ownership types while still retaining enough control on object access.

The access modifier of an object is decided by its creator and does not change over the object's lifetime. However, to allow a more flexible programming style, our type system allows the access modifier to be varied inwards. In the following example, the assignment `a1 = a2` being OK implies that objects accessed via `a2` can also be accessed via `a1`. This variance is safe because the set of objects that can access `o` is a subset of the objects that can access `world`. In other words, an object in variable `a2` can become less accessible when it is assigned to variable `a1`; the converse does not apply. 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 use this variance in our type system, and its usage is *implicit* (via binding) and very similar to the way type subsumption works in any typed language. As with subtyping, this variance of accessibility applies to language-level expressions; at runtime, an object's accessibility is fixed, and is determined by its creation type.

```
class C<o> {
  [o] A<o> a1;
  [world] A<o> a2;
  [this] A<o> a3;
  m() {
    a1 = a2;        // OK, o inside world
    a1 = a3;        // ERROR, o outside this
    a2 = a1;        // ERROR, world outside o
    a2 = a3;        // ERROR, world outside this
    a3 = a1;        // OK, this inside o
    a3 = a2; } }    // OK, this inside world
```

When the owner context and access modifier are the same, the variant ownership type `[o] A<o>` is the same as the `A<o>` in ownership types. Hence our model with modifiers subsumes the owners-as-dominators model. The following example illustrates how separating access from the owner allows us to achieve different kinds of access protection.

```
class D<o> {
  [o] A<o> a1;      // partly protected
  [this] A<this> a2;// encapsulated
  [this] A<o> a3;   // encapsulated, but field is writable from outside
  [o] A<this> a4; } // not encapsulated, field is read-only from outside

class E<o> {
  [o] D<o> d;
  m() {
    [o]A<o> x = d.a1;   // OK to read the reference
    d.a1 = x;           // OK to update the field

    ... = d.a2;         // ERROR to read, cannot name context inside
                        // d.a2's access modifier ('this' in d)
```

```
d.a2 = ...  // ERROR to write, cannot name d.a2's owner ('this' in d)

... = d.a3; // ERROR to read, cannot name context inside 'this' in d
d.a3 = new [o]A<o>; // OK to write, 'this' in d inside o

[o]A<*> y = d.a4;   // OK to read, the owner of d.a4 is abstracted
d.a4 = ... } }      // ERROR to write, cannot name d.a4's owner
```

When an object is encapsulated by its defining object `this`, it is not accessible from outside. The reference in field `a3` is protected from being accessed from outside of the object. However a client can assign the field itself with an expression of type `[p]A<o>` where `this` is known to be inside `p` (`= o` in the example). This reference does not break the accessibility invariant because the modifier is variant inwards (from `p` to `this`). Such accessibility restrictions are useful but cannot be expressed by any of the existing ownership type systems which use an invariant owner to control access.

In contrast to `a3`, the field `a4` can be read from outside (via context abstraction, which will be discussed in the next subsection) but is not updatable from outside of the object. The field `a4` can be considered as an object-level (instead of traditional class-level) read-only field — it is not updatable by a client from outside, but can be updated from objects within the current context. Again, this kind of object-level read-only restriction is not supported by previous ownership type systems.

3.2 Context Variance

Type variance is a recently developed approach for increasing code genericity in typed programming languages. Use-site variance on parametric types [15] has been implemented in the new version of Java 1.5 with wildcards [25]. Our type system adds variance to context arguments rather than type arguments. By allowing context variance, not only do we achieve code genericity, but we also allow a much more flexible reference structure than the original ownership types by removing the naming restriction of object contexts. Technically, use-site variance is a form of existential types, in our case, existential contexts. The technical detail and formalization are discussed in Section 4. Here we introduce the context variances informally with examples.

Programmers may *explicitly* declare the variance of context arguments wherever they form types. For a concrete context K, we write K+ for inward variance which means any context inside K, K− for outward variance which means any context outside K and $*$ for full abstraction which means any context. Recall that concrete contexts are those of formal context parameters, the current context this and the world context. Figure 1 shows these variances for a given context K.

The following subtype relations show the variant ownership types with two contexts K and K′ where K dominates K′ (domination is the reflexive and transitive relation of ownership). For simplicity, access modifiers are all defaulted (to the world context) because they are orthogonal to the context argument variance.

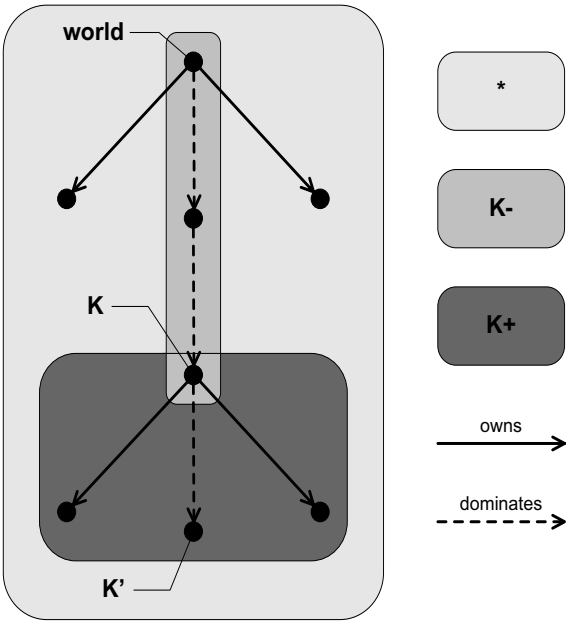

Fig. 1. Context Variances

$C\langle K'\rangle <: C\langle K'+\rangle <: C\langle K+\rangle <: C\langle *\rangle$
$C\langle K\rangle <: C\langle K-\rangle <: C\langle K'-\rangle <: C\langle *\rangle$

Variance on context arguments is not to be confused with variance on the access modifier which is variant implicitly. Moreover, access modifiers can never be abstracted while argument variances are a form of context abstraction. For convenience, when all context arguments are fully abstract, we will elide all the `*` contexts. For instance, in the following example, `[o] A` is short for `[o] A<*>`. Also because we use the `world` context as default access modifier, `A` is short for `[world] A<*>`.

```
class F<o> { [o] A a; }

class G<o> {
  F<this> f1;               // invariant on context argument
  F<*> f2;                  // fully abstract on context argument
  F<this+> f3;              // inward variance on context argument
  F<this-> f4;              // outward variance on context argument
  A<this> a;
  m() {
    [this] A x = f1.a;      // OK to read
    f1.a = x;               // OK to write

    ... = f2.a;// ERROR, cannot name context inside f2.a's
               // access modifier (which is abstracted)
```

```
f2.a = ... // ERROR (in most cases),
           // cannot name context outside f2.a's access modifier
f2.a = a;  // OK! the only exception, world outside any context!

...= f3.a;// ERROR, cannot name context inside f3.a's access modifier
f3.a = new [o]A<o>();   // OK, f3.a's access modifier inside o

x = f4.a;  // OK, 'this' inside f4.a's access modifier
f4.a = ... // ERROR (in most cases),
           // cannot name context outside f4.a's access modifier
f4.a = a; } } // OK! the only exception, world outside any context!
```

The choice of variance is made when types are used. Programmers can use any combination of invariance, inward/outward variance or full abstraction to express context arguments in a type. Invariant contexts are most usable but least flexible because one must be able to name the concrete context. `f1` is most usable, because `f1.a` is both readable and writable; but `f1` is less flexible because it can only be accessed from within the current context. Fully abstract contexts are most flexible but least usable because all information about the context is hidden; `f2` is least usable because `f2.a` is neither readable nor writable (except for the special case with the world context as shown in the example). The type of `f2.a` is `[?]A` where the `?` denotes an unknown context; the only thing we know about `?` is that it is inside `world`. Unknown contexts are not for programmer use, but are used in our semantics. They are simply shorthand for an anonymous context with given variance which is existentially quantified. However, the combinations of fully abstract and invariant context arguments are useful as we are about to see in revisiting the list example. Inward/outward variant contexts give a choice between invariance and full abstraction where some information of the context is available to give programmers just enough information they need to use the context, as we see with `f3.a` and `f4.a`. Within class `G` their types are `[this+?]A` (respectively `[this-?]A`) where the unknown contexts are bounded inside (respectively outside) `this`.

We extend the above example with some more complicated cases of variance which involve nested variances and mixed inward/outward variances. The type system is able to derive the ordering information in the presence of nested variances. Some of the types involved are:

`h1.f1 : F<o+?+>` and `h1.f1.a : [o+?+?]A<*>`

we can derive that `o+?+?` is inside `o`. Also we find:

`h1.f2 : F<o+?->` and `h2.f1 : F<o-?+>`

The variance `o+?-` contains contexts `o` and `world` but not `this`. Similarly `o-?+` contains `o` and `this` but not `world`.

```
class H<o> {
  F<o+> f1;
  F<o-> f2; }
```

```
class I<o> {
  H<o+> h1;
  H<o-> h2;
  [o] A a;
  m() {
    h1.f1.a = a;              // OK, h1.f1.a's access modifier inside o
    a = h2.f2.a;              // OK, o inside h2.f2.a's access modifier

    h1.f2 = new F<o>;         // OK
    h1.f2 = new F<world>;     // OK
    h1.f2 = new F<this>;      // ERROR
    h2.f1 = new F<o>;         // OK
    h2.f1 = new F<this>;      // OK
    h2.f1 = new F<world>; } } // ERROR
```

Now we are in a good position to revisit the list example we discussed in the previous section.

3.3 The List Example: Revisited

We revisit the list example with a solution to the two problems considered previously: iterator accessibility and fixed ownership of data.

```
class List<o, d> {
  [this] Node<this, d> head;
  [o]Iterator<this, d> getIter(){return new [o]Iterator<this, d>(head);}}

class Node<o, d> {
  [o] Node<o, d> next;
  [d] Data data;
  Node([o]Node<o, d> next, [d]Data data) {
    this.next = next;
    this.data = data; } }

class Iterator<o, d> {
  [o] Node<o, d> current;
  Iterator([o]Node<o, d> Node) { current = Node; }
  [d]Data element() { return current.data; }
  void next() { current = current.next; }
  void add([d]Data data) {
    current.next = new Node<o, d>(current.next, data); } }

class Data<o> { void useMe(){ ... } }
```

The implementation of the `List` and `Iterator` classes is almost the same as for ownership types except the type of iterators created by the list has the access modifier the same to the owner of the list, which essentially means anyone who can name the owner of the list is allowed to access its iterators. By creating iterators with accessibility as `o`, the list object authorizes the iterators to act as its interface objects and to be used by the client to manipulate on itself.

However, the list's representation (that is, the `Node` objects) is always protected from the client and never exposed to the outside directly. To access the nodes, the client must use either the list itself or the iterators created by the list.

In the `Node` class, the type of `data` field is `[d] Data`. As we have mentioned, this is shorthand for `[d] Data<*>` where the owner of these `Data` objects is abstract. The `Node` class is a recursive structure so all the node objects must have the same type. However, with our owner abstraction, each node may contain data objects owned by different contexts as shown in the client program.

```
class Client<o> {
  void m() {
    List<this, o> list = new List<this, o>();   // OK
    [this]Iterator<*, o> iter = list.getIter(); // OK
    iter.add(new [o]Data<o>());    // OK, o inside o, o matches *
    iter.add(new Data<world>());   // OK, o inside world, world matches *
    iter.add(new [this]Data<this>());           // ERROR, o outside this!
    iter.add(new [o]Data<this>()); // OK, o is inside o, this matches *
    iter.element().useMe();                     // OK
    iter.current = ... } }         // ERROR, access modifier abstracted
```

The client creates the list object as usual, but in order to obtain a reference to iterator objects returned by the list, it must declare a type which abstracts the owner of iterators (which is the list object, see the `List` class). However, in the type of iterators, the second context argument remains concrete, which is necessary in order to reference data objects returned by iterators. Moreover, with context variance, now the client can add data objects owned by various contexts into the list. Objects with type `[this] Data<this>` cannot be added into the list because the access modifier is variant outwards (from `this` to `o`) which is not sound hence not permitted by the type system. Note that the type system guarantees iterators cannot expose the node objects to the client.

4 The Formal Language

In this section, we formalize variant ownership types in a core language based on *Featherweight Java* [14] extended with field assignment. We incorporate contexts and formalize the main properties.

4.1 Syntax

The abstract syntax for the source languages is given in Table 1. The metavariable T ranges over types; N ranges over nameable contexts (or concrete contexts); K ranges over contexts; V ranges over context variances; L ranges over class definitions; M ranges over method definitions; e ranges over expressions; C, D range over class names; f and m range over field names and method names respectively; X, Y range over formal context parameters; and x ranges over variable names with this as a special variable name to reference the target object for the current call. The overbar is used for a sequence of constructs; for example,

Table 1. Abstract Syntax for Source Language

T	$::= [\mathsf{N}]\ \mathsf{C}\langle\overline{\mathsf{V}}\rangle$	types
N	$::= \mathsf{X} \mid \mathsf{this} \mid \mathsf{world}$	nameable contexts
K	$::= \mathsf{N}$	contexts
V	$::= \mathsf{K} \mid \mathsf{K}+ \mid \mathsf{K}- \mid *$	context variances
L	$::= \mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle \lhd \mathsf{D}\langle\overline{\mathsf{Y}}\rangle\ \{\overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \overline{\mathsf{M}}\}$	classes
M	$::= \mathsf{T}\ \mathsf{m}(\overline{\mathsf{T}}\ \overline{\mathsf{x}})\ \{e\}$	methods
e	$::=$	terms
	x	variable
	$\mid \mathsf{new}\ \mathsf{T}(\overline{e})$	new
	$\mid e.\mathsf{f}$	select
	$\mid e.\mathsf{m}(\overline{e})$	call
	$\mid e.\mathsf{f} = e$	assignment

Table 2. Extended Syntax for Type System

K	$::= \ldots \mid \mathsf{K}+_? \mid \mathsf{K}-_? \mid ?$	contexts
P	$::= \overline{\mathsf{L}}\ e$	programs
Γ	$::= \bullet \mid \Gamma, \mathsf{X} \preceq \mathsf{Y} \mid \Gamma, \mathsf{x} : \mathsf{T}$	environments

$\overline{e}$ is used for a possibly empty sequence $e_1..e_n$, $\overline{\mathsf{T}}\ \overline{\mathsf{x}}$ stands for a possibly empty sequence of pairs $\mathsf{T}_1\ \mathsf{x}_1..\mathsf{T}_n\ \mathsf{x}_n$, etc. In the class production, inheritance $\lhd\ \mathsf{D}\langle\overline{\mathsf{Y}}\rangle$ is optional because our type system does not need a top type.

The syntax distinguishes between concrete (nameable) contexts N and those contexts K allowing the abstract contexts. Table 2 shows the extended syntax used by the type system, which is not accessible by programmers. Abstract contexts $\mathsf{K}+_?$, $\mathsf{K}-_?$ and ? correspond to context variances $\mathsf{K}+$, $\mathsf{K}-$ and $*$. The difference between $\mathsf{K}+$ and $\mathsf{K}+_?$ is that $\mathsf{K}+$ means all contexts inside K while $\mathsf{K}+_?$ is one context in the set of $\mathsf{K}+$. Actually, $\mathsf{K}+_?$ is a bounded existential context whose name is anonymous; but we do know it is inside K. The unbound existential context ? is an arbitrary context; we know nothing about it (except it is inside the upper bound context world in the context hierarchy). Figure 2 shows the concept of existential contexts. A program P is a pair consisting of a fixed sequence of class definitions and an expression e which is the body of the main method. The environment Γ may contain the types of variables and domination relations between formal context parameters.

4.2 Static Semantics

The same syntactical abbreviation for sequences is used in the typing rules. A sequence of judgements can be simplified with an overbar on the argument, such as $\Gamma; \mathsf{N} \vdash \overline{\mathsf{T}}$. Substitution $[\overline{\mathsf{V}}/\overline{\mathsf{X}}]\mathsf{T}$ is used to substitute $\overline{\mathsf{V}}$ for $\overline{\mathsf{X}}$ in T; this substitution also requires $|\overline{\mathsf{V}}| = |\overline{\mathsf{X}}|$. Sometimes we use implications, denoted by $\Longrightarrow$, to avoid repeating rules with similar structure. Other symbols used in the type system are: $\bullet$ means empty set; _ and ... match any single or multiple things; $1..n$ means an enumeration from 1 to n.

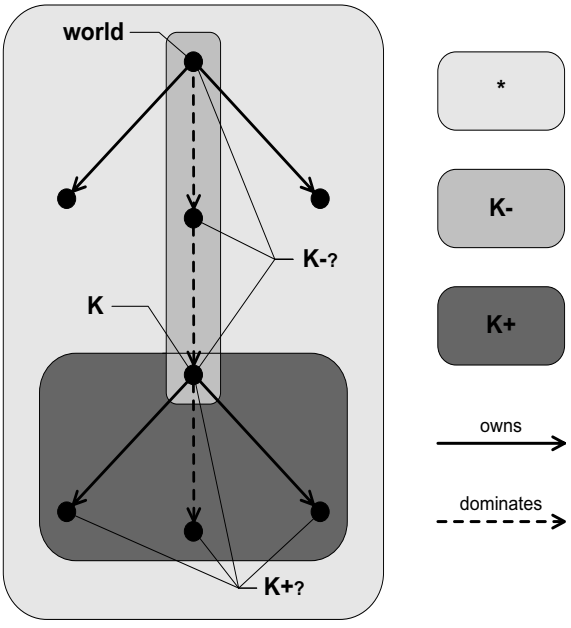

Fig. 2. Existential Contexts

In ownership type systems the contexts used to form types are actual runtime objects. In order to prove the desired dynamic properties, we need to incorporate the bindings of context parameters into the type system. Typically, the expression judgement $\Gamma; \mathsf{N} \vdash e : \mathsf{T}$ holds for the current context N. The context N is bound to the current object (the target object of current call); in the static semantics N is always bound to the variable this or world for the top-level program expression, while in the dynamic semantics N is bound to the location of the actual object in heap or world. Note that the bindings for all context parameters in the current environment can be determined from the type of N at runtime. To simplify the dynamic semantics we will annotate locations with their object type.

$$\texttt{[VAR-ANY]}\ \frac{}{\Gamma \vdash \mathsf{V} \subseteq *} \qquad \texttt{[VAR-IN']}\ \frac{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'}{\Gamma \vdash \mathsf{K}+ \subseteq \mathsf{K}'+}$$

$$\texttt{[VAR-CRT]}\ \frac{}{\Gamma \vdash \mathsf{N} \subseteq \mathsf{N}} \qquad \texttt{[VAR-OUT']}\ \frac{\Gamma \vdash \mathsf{K}' \preceq \mathsf{K}}{\Gamma \vdash \mathsf{K}- \subseteq \mathsf{K}'-}$$

$$\texttt{[VAR-IN]}\ \frac{}{\Gamma \vdash \mathsf{K}+_? \subseteq \mathsf{K}+} \qquad \texttt{[VAR-TRA]}\ \frac{\Gamma \vdash \mathsf{V} \subseteq \mathsf{V}'' \qquad \Gamma \vdash \mathsf{V}'' \subseteq \mathsf{V}'}{\Gamma \vdash \mathsf{V} \subseteq \mathsf{V}'}$$

$$\texttt{[VAR-OUT]}\ \frac{}{\Gamma \vdash \mathsf{K}-_? \subseteq \mathsf{K}-}$$

The [VAR] rules define the valid context variances. Since context variances represent sets of contexts, the [VAR] rules really just define the subset relations between them. Contexts can be considered as singleton sets containing only one element. The only rule that can be applied to the unbound existential context ? is the [VAR-ANY] rule. By inspection it is also clear that we cannot have anything

as a subset of any existential context, bound or unbound. This is a key property of the system.

The [TYPE] rule states that expressible types are those that are supertypes of object types; this introduces the context variances into valid types. In order to maintain an ownership tree on the heap, objects must be constructed using new with concrete contexts. By [TYPE-OBJ] a well-formed object type must satisfy the standard context ordering constraints for classes, namely that the first argument, the owner context for the type is within other context arguments; all contexts must be valid concrete contexts by [CONTEXT]. The [SUB-VAR] rule allows the access modifier to be varied outwards in a subtype; context arguments can also be narrowed according to the [VAR] rules. Note that the class definitions are global so that we simply use $\texttt{class}\ \mathsf{C}[\overline{\mathsf{X}}] \lhd \mathsf{D}[\overline{\mathsf{Y}}]$ ... to hypothesize a valid class definition in the [SUB-EXT] rule and some other rules in the type system.

$$\texttt{[TYPE]}\quad \frac{\Gamma; \mathsf{N} \vdash_o \mathsf{T}_o \qquad \Gamma \vdash \mathsf{T}_o <: \mathsf{T}}{\Gamma; \mathsf{N} \vdash \mathsf{T}}$$

$$\texttt{[TYP-OBJ]}\quad \frac{|\overline{\mathsf{N}}| = \mathtt{arity}(\mathsf{C}) \qquad \Gamma; \mathsf{N} \vdash \mathsf{N}', \overline{\mathsf{N}} \qquad \Gamma \vdash \mathsf{N}_1 \preceq \overline{\mathsf{N}}}{\Gamma; \mathsf{N} \vdash_o [\mathsf{N}']\ \mathsf{C}\langle\overline{\mathsf{N}}\rangle}$$

$$\texttt{[SUB-VAR]}\quad \frac{\Gamma \vdash \mathsf{N}' \preceq \mathsf{N} \qquad \Gamma \vdash \overline{\mathsf{V}} \subseteq \overline{\mathsf{V}'}}{\Gamma \vdash [\mathsf{N}]\ \mathsf{C}\langle\overline{\mathsf{V}}\rangle <: [\mathsf{N}']\ \mathsf{C}\langle\overline{\mathsf{V}'}\rangle}$$

$$\texttt{[SUB-EXT]}\quad \frac{\mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle \lhd \mathsf{D}\langle\overline{\mathsf{Y}}\rangle\ \ldots \qquad \mathsf{T} = [\mathsf{N}]\ \mathsf{D}\langle[\overline{\mathsf{V}}/\overline{\mathsf{X}}]\overline{\mathsf{Y}}\rangle \qquad \Gamma \vdash \mathsf{T} <: \mathsf{T}}{\Gamma \vdash [\mathsf{N}]\ \mathsf{C}\langle\overline{\mathsf{V}}\rangle <: \mathsf{T}}$$

Our subtyping rules need to handle context abstraction correctly, and avoid breaking accessibility constraints through assignments to fields, or method parameters, with some of their types' contexts hidden. The main idea of our system is to substitute existential contexts for any variant contexts in the type of an object (via an opening process as we see later) when we determine the types of its fields/methods, and to prohibit binding to fields or method parameters which include existential contexts in their types. Let us use the phrase *existential type* to describe a type containing an existential context. By guaranteeing that existential types cannot be supertypes, we achieve the desired prohibition (note the subtyping premise in all [EXP] rules involve binding). We now explain how the [SUB] rules achieve this. We cannot use [SUB-VAR] to find an existential supertype because its premise would require there to be some subset of an existential context, which the [VAR] rules preclude. It follows that no existential type can be a subtype of itself, because the alternative [SUB-EXT] is not applicable for the reflexive case. Finally any type T judged to be a supertype by [SUB-EXT] must be a supertype of itself according to the last premise of the rule. It follows that no type judged to be a supertype by these rules can contain an existential context.

Legal concrete contexts include formal context parameters, the current context this and the world context. Recall that the current context N, in the static semantics, is always bound to this or world. Context ordering rules define the domination relation between contexts. Domination is the reflexive and transitive closure of ownership. Direct ownership is captured in the [ORD-OWN] rule by

looking up the owner from the type of the context via [LKP-OWN] (appearing at the end of this subsection). The only direct ownership relation available in the static semantics is for the this context; it is owned by the first context parameter of its type (see [LKP-OWN] and [LKP-OWN′]). The this context is the only context that is given a static type; this is both a context parameter and a variable naming the current object. At runtime, this is bound to the location of the target object. The ordering on existential contexts is not surprising; $? \preceq \mathsf{world}$ by the [ORD-WLD] rule, but no other ordering is derivable for ?.

$$\texttt{[CTX-LCL]}\ \frac{\Gamma; \mathsf{N} \vdash \mathsf{N} : [\mathsf{N}'']\ \mathsf{C}\langle\overline{\mathsf{N}}\rangle \qquad \mathsf{N}' \in \overline{\mathsf{N}} \cup \{\mathsf{N}\}}{\Gamma; \mathsf{N} \vdash \mathsf{N}'}$$

$$\texttt{[CTX-WLD]}\ \frac{}{\Gamma; \mathsf{N} \vdash \mathsf{world}}$$

$$\texttt{[ORD-OWN]}\ \frac{}{\Gamma \vdash \mathsf{N} \preceq \mathit{owner}_\Gamma(\mathsf{N})}$$

$$\texttt{[ORD-TRA]}\ \frac{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'' \qquad \Gamma \vdash \mathsf{K}'' \preceq \mathsf{K}'}{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}'}$$

$$\texttt{[ORD-RFL]}\ \frac{}{\Gamma \vdash \mathsf{N} \preceq \mathsf{N}}$$

$$\texttt{[ORD-ENV]}\ \frac{\mathsf{X} \preceq \mathsf{X}' \in \Gamma}{\Gamma \vdash \mathsf{X} \preceq \mathsf{X}'}$$

$$\texttt{[ORD-WLD]}\ \frac{}{\Gamma \vdash \mathsf{K} \preceq \mathsf{world}}$$

$$\texttt{[ORD-IN]}\ \frac{}{\Gamma \vdash \mathsf{K}{+_?} \preceq \mathsf{K}}$$

$$\texttt{[ORD-OUT]}\ \frac{}{\Gamma \vdash \mathsf{K} \preceq \mathsf{K}{-_?}}$$

The [PROGRAM] rule simply checks the expression in the main method; world is the only concrete context available at this level.

$$\texttt{[PROGRAM]}\ \frac{\vdash \overline{\mathsf{L}} \qquad \bullet; \mathsf{world} \vdash e : \mathsf{T}}{\vdash \overline{\mathsf{L}}\ e}$$

Class well-formedness is checked in the [CLASS] rule. Each class defines its own environment formed from its formal contexts and the type of this object. In the original ownership type system, the owner parameter X_1 had to be dominated by all other context parameters, we follow the same convention here. Note that the only direct ownership relation known to the class, that is $\mathsf{this} \preceq \mathsf{X}_1$, is not included in the class environment; instead we capture it in the [ORD-OWN] rule to make it generally derivable, in particular for its use in the dynamic semantics. Furthermore, field types and methods need to be checked for well-formedness.

If a class is extended from another then the supertype needs to be valid in the environment formed from the class, and the owner of the supertype must be the same as the owner of the current context. Not surprisingly, new field names need to be distinguished from the field names used in the supertype. Moreover, the supertype is bound to a super variable in the environment that is used only by the [METHOD] rule to check the correctness of overridden methods. We implicitly assume the access modifier for the types of both this and super variables is the default access modifier world.

$$\texttt{[CLASS]}\ \frac{\begin{array}{c}\Gamma = \mathsf{X}_1 \preceq \overline{\mathsf{X}}, \mathsf{this} : \mathsf{C}\langle\overline{\mathsf{X}}\rangle, \mathsf{super} : \mathsf{D}\langle\overline{\mathsf{Y}}\rangle \qquad \mathsf{X}_1 = \mathsf{Y}_1 \qquad \Gamma; \mathsf{this} \vdash \mathsf{D}\langle\overline{\mathsf{Y}}\rangle, \overline{\mathsf{T}} \\ \Gamma \vdash \overline{\mathsf{M}} \qquad \overline{\mathsf{f}} \cap \mathit{dom}(\mathit{fields}(\mathsf{D}\langle\overline{\mathsf{Y}}\rangle, \mathsf{this})) = \bullet\end{array}}{\vdash \mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle \lhd \mathsf{D}\langle\overline{\mathsf{Y}}\rangle\ \{\overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \overline{\mathsf{M}}\}}$$

In the [METHOD] rule, all types are checked for well-formedness and a new environment is constructed by extending the class environment with method parameters and their types. The method body is checked in the new environment

and the current context this. Methods can be overridden in the traditional way — covariant on the return type and contravariant on the types of method parameters.

$$\texttt{[METHOD]}\ \frac{\begin{array}{c}\Gamma;\mathsf{this}\vdash \mathsf{T},\overline{\mathsf{T}} \qquad \Gamma,\overline{x}:\overline{\mathsf{T}};\mathsf{this}\vdash e:\mathsf{T}'' \qquad \Gamma\vdash \mathsf{T}''<:\mathsf{T}\\ \mathtt{method}(\Gamma(\mathsf{super}),\mathsf{this},m)=\mathsf{T}'\ m(\overline{\mathsf{T}'}\ \_)\ldots \implies\\ \Gamma\vdash \mathsf{T}<:\mathsf{T}' \qquad \Gamma\vdash\overline{\mathsf{T}'}<:\overline{\mathsf{T}}\end{array}}{\Gamma\vdash \mathsf{T}\ m(\overline{\mathsf{T}}\ \overline{x})\ \{e\}}$$

In the [EXP-NEW] rule, new objects are created using concrete contexts (according to [TYP-OBJ]) in order to establish an ownership tree in heap. For simplicity, we force all the fields of the object to be initialized at creation time. The internal context of the newly created object is an anonymous context inside its owner. The [EXP-SEL] and [EXP-CAL] rules lookup the types of fields or methods for the target expression e. They need to decide if they are able to name the internal context of e by using the auxiliary function $\mathtt{rep}_{\mathsf{T}}()$ which simply checks if e is the current context (i.e. this). If e is the current context then it is used as the internal context of e in the lookup functions $\mathtt{fields}()$ and $\mathtt{method}()$; otherwise, an anonymous context is used instead thus hiding the internal context (see [LKP-REP]).

All rules for expressions that involve some form of binding, such as [EXP-ASS], [EXP-CAL] and [EXP-NEW], use a subtype constraint to ensure that the type of the target of the binding does not involve any existential contexts (recall the [SUB] rules). A more conventional formulation of these rules would shift the subtype check onto a subsumption rule, but that cannot be done here — we need to use distinct types for the source and target of the binding in the rules.

$$\texttt{[EXP-VAR]}\ \frac{\Gamma(x)=\mathsf{T}}{\Gamma;\mathsf{N}\vdash x:\mathsf{T}}$$

$$\texttt{[EXP-NEW]}\ \frac{\Gamma;\mathsf{N}\vdash_{\mathsf{o}}\mathsf{T} \quad \Gamma;\mathsf{N}\vdash\overline{e}:\overline{\mathsf{T}'} \quad \mathtt{fields}(\mathsf{T},\mathtt{owner}(\mathsf{T})+_?)=\overline{f}\ \overline{\mathsf{T}} \qquad \Gamma\vdash\overline{\mathsf{T}'}<:\overline{\mathsf{T}}}{\Gamma;\mathsf{N}\vdash \mathsf{new}\ \mathsf{T}(\overline{e}):\mathsf{T}}$$

$$\texttt{[EXP-SEL]}\ \frac{\Gamma;\mathsf{N}\vdash e:\mathsf{T} \qquad \mathtt{fields}(\mathsf{T},\mathtt{rep}_{\mathsf{T}}(\mathsf{N},e))(f)=\mathsf{T}'}{\Gamma;\mathsf{N}\vdash e.f:\mathsf{T}'}$$

$$\texttt{[EXP-ASS]}\ \frac{\Gamma;\mathsf{N}\vdash e':\mathsf{T}' \qquad \Gamma;\mathsf{N}\vdash e.f:\mathsf{T} \qquad \Gamma\vdash\mathsf{T}'<:\mathsf{T}}{\Gamma;\mathsf{N}\vdash e.f=e':\mathsf{T}'}$$

$$\texttt{[EXP-CAL]}\ \frac{\begin{array}{c}\Gamma;\mathsf{N}\vdash e:\mathsf{T} \qquad \Gamma;\mathsf{N}\vdash\overline{e}:\overline{\mathsf{T}} \qquad \Gamma\vdash\overline{\mathsf{T}}<:\overline{\mathsf{T}'}\\ \mathtt{method}(\mathsf{T},\mathtt{rep}_{\mathsf{T}}(\mathsf{N},e),m)=\mathsf{T}'\ m(\overline{\mathsf{T}'}\ \_)\ldots\end{array}}{\Gamma;\mathsf{N}\vdash e.m(\overline{e}):\mathsf{T}'}$$

When accessing the fields or methods via an expression e, we determine their types, given the type of e. These in turn use [LKP-DEF] to find a correct substitution for parameters of T's class. The opening process requires the replacement of context variances with corresponding existential contexts. This process is similar to the usual unpack/open for conventional existential types. The major difference is that our open process does not introduce fresh context variables into the current environment. Instead, we keep the existential context anonymous by

annotating context variances with a special symbol ?. This technique not only eliminates the need for the pack/close operation (since anonymous contexts do not have to be bound to an environment, they naturally become global), but also makes the proofs simpler. Moreover, this technique is capable of handling complicated variances which would need nested open/close operations.

$$\texttt{[LKP-DEF]}\quad \frac{\mathsf{L} = \mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle\ \ldots \qquad \mathit{open}(\mathsf{T}) = \mathsf{C}\langle\overline{\mathsf{K}}\rangle}{\mathit{defin}(\mathsf{T},\mathsf{K}) = [\overline{\mathsf{K}}/\overline{\mathsf{X}}, \mathsf{K}/\mathtt{this}]\mathsf{L}}$$

$$\texttt{[LKP-FLD]}\quad \frac{\mathit{defin}(\mathsf{T},\mathsf{K}) = \mathsf{class}\ \ldots \lhd \mathsf{T}'\ \{\overline{\mathsf{T}}\ \overline{\mathsf{f}};\ \ldots\}}{\mathit{fields}(\mathsf{T},\mathsf{K}) = \overline{\mathsf{f}}\ \overline{\mathsf{T}}, \mathit{fields}(\mathsf{T}',\mathsf{K})}$$

$$\texttt{[LKP-MTH]}\quad \frac{\mathit{defin}(\mathsf{T},\mathsf{K}) = \mathsf{class}\ \ldots\ \mathsf{T}'\ \mathsf{m}(\overline{\mathsf{T}}\ \overline{\mathsf{x}})\{e\}\ \ldots}{\mathit{method}(\mathsf{T},\mathsf{K},\mathsf{m}) = \mathsf{T}'\ \mathsf{m}(\overline{\mathsf{T}}\ \overline{\mathsf{x}})\{e\}}$$

$$\texttt{[LKP-MTH']}\quad \frac{\mathit{defin}(\mathsf{T},\mathsf{K}) = \mathsf{class}\ \ldots \lhd \mathsf{T}'\ \{\ \ldots\ ;\overline{\mathsf{M}}\} \qquad \mathsf{m} \notin \overline{\mathsf{M}}}{\mathit{method}(\mathsf{T},\mathsf{K},\mathsf{m}) = \mathit{method}(\mathsf{T}',\mathsf{K},\mathsf{m})}$$

$$\texttt{[LKP-ARI]}\quad \frac{\mathsf{class}\ \mathsf{C}\langle\overline{\mathsf{X}}\rangle\ \ldots}{\mathit{arity}(\mathsf{C}) = |\overline{\mathsf{X}}|}$$

$$\texttt{[OPEN]}\quad \frac{}{\mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}\rangle) = \mathsf{C}\langle\overline{\mathsf{K}}\rangle}$$

$$\texttt{[OPN-ANY]}\quad \frac{}{\mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}, *, \overline{\mathsf{V}}\rangle) = \mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}, ?, \overline{\mathsf{V}}\rangle)}$$

$$\texttt{[OPN-IN]}\quad \frac{}{\mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}, \mathsf{K}+, \overline{\mathsf{V}}\rangle) = \mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}, \mathsf{K}+_?, \overline{\mathsf{V}}\rangle)}$$

$$\texttt{[OPN-OUT]}\quad \frac{}{\mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}, \mathsf{K}-, \overline{\mathsf{V}}\rangle) = \mathit{open}(\mathsf{C}\langle\overline{\mathsf{K}}, \mathsf{K}-_?, \overline{\mathsf{V}}\rangle)}$$

$$\texttt{[LKP-OWN]}\quad \frac{\Gamma; \bullet \vdash e : \mathsf{T}}{\mathit{owner}_\Gamma(e) = \mathit{owner}(\mathsf{T})}$$

$$\texttt{[LKP-OWN']}\quad \frac{}{\mathit{owner}([\mathsf{N}]\ \mathsf{C}\langle\overline{\mathsf{V}}\rangle) = \mathsf{V}_1}$$

$$\texttt{[LKP-REP]}\quad \frac{e \neq \mathsf{N}}{\mathit{rep}_\mathsf{T}(\mathsf{N}, e) = \mathit{owner}(\mathsf{T})+_?}$$

$$\texttt{[LKP-REP']}\quad \frac{}{\mathit{rep}_\mathsf{T}(\mathsf{N}, \mathsf{N}) = \mathsf{N}}$$

4.3 Dynamic Semantics and Properties

The extended syntax and features used by the dynamic semantics are given in Table 3. The ownership information is usually only used in static type checking. However, in order to obtain a formal proof for some of the key properties of the type system, we need to establish a connection between the static and dynamic semantics by including ownership relations in the dynamic semantics. Terms and contexts are extended with locations, which are annotated with the type of object they refer to. A heap is a mapping from locations to objects; an object maps fields to locations. Object creation extends the heap, introducing a new location which is then forever associated with its object; field assignment updates an object but does not directly affect the heap.

Table 3. Extended Syntax with Dynamic Features

$\mathsf{l}, \mathsf{l}_\mathsf{T}$		typed locations
e	$::= \ldots \mid \mathsf{l}$	terms
N	$::= \ldots \mid \mathsf{l}$	nameable contexts
o	$::= \overline{\mathsf{f}} \mapsto \overline{\mathsf{l}}$	objects
H	$::= \overline{\mathsf{l}} \mapsto \overline{\mathsf{o}}$	heaps

There are also a few auxiliary definitions to help formalize the properties. Locations are annotated with their type. From this we can lookup the accessibility context for an object stored at that location. The objects in the heap form an ownership tree just as in other ownership type systems. However, the reference containment invariant is different. An object needs to be inside another object's modifier in order to access it.

[EXP-LOC] $$\frac{}{\Gamma; \mathsf{N} \vdash \mathsf{l}_\mathsf{T} : \mathsf{T}}$$

[LKP-ACC] $$\frac{\bullet; \bullet \vdash \mathsf{l} : [\mathsf{N}]\ \mathsf{C}\langle\overline{\mathsf{V}}\rangle}{\mathsf{acc}(\mathsf{l}) = \mathsf{N}}$$

[HEAP] $$\frac{\forall \mathsf{l} \in \mathsf{dom}(\mathsf{H})\ \cdot\ \bullet; \bullet \vdash \mathsf{l} : \mathsf{T} \qquad \mathsf{H}(\mathsf{l}) = \overline{\mathsf{f}} \mapsto \overline{\mathsf{l}} \qquad \mathsf{fields}(\mathsf{T}, \mathsf{l}) = \overline{\mathsf{f}}\ \overline{\mathsf{T}} \qquad \bullet; \bullet \vdash \overline{\mathsf{l}} : \overline{\mathsf{T}'} \qquad \Gamma \vdash \overline{\mathsf{T}'} <: \overline{\mathsf{T}} \qquad \bullet \vdash \mathsf{l} \preceq \mathsf{acc}(\overline{\mathsf{l}})}{\vdash \mathsf{H}}$$

The reduction rules are defined in a big step fashion. The context N in $\Downarrow_\mathsf{N}$ refers to the target object of the current call, or the world context in case of the main method. At the time of method invocation in [RED-CAL], the target object of the body of the invoked method is l. Notice that the variable this is not substituted in $[\overline{\mathsf{l}}/\overline{\mathsf{x}}]e'$. Instead, this is replaced by l in the substitution provided by the lookup function $\mathsf{method}(\mathsf{T}, \mathsf{l}, \mathsf{m})$.

[EXECUTION] $$\frac{\bullet; e \Downarrow_{\mathsf{world}} \mathsf{H}; \mathsf{l}}{\overline{\mathsf{L}}\ e \Downarrow \mathsf{l}}$$

[RED-CAL] $$\frac{\mathsf{H}; e \Downarrow_\mathsf{N} \mathsf{H}'; \mathsf{l} \qquad \mathsf{H}'; \overline{e} \Downarrow_\mathsf{N} \mathsf{H}''; \overline{\mathsf{l}} \qquad \bullet; \mathsf{N} \vdash \mathsf{l} : \mathsf{T} \qquad \mathsf{method}(\mathsf{T}, \mathsf{l}, \mathsf{m}) = \ldots(\_\ \overline{\mathsf{x}})\{e'\} \qquad \mathsf{H}''; [\overline{\mathsf{l}}/\overline{\mathsf{x}}]e' \Downarrow_\mathsf{l} \mathsf{H}'''; \mathsf{l}'}{\mathsf{H}; e.\mathsf{m}(\overline{e}) \Downarrow_\mathsf{N} \mathsf{H}'''; \mathsf{l}'}$$

[RED-NEW] $$\frac{\mathsf{H}; \overline{e} \Downarrow_\mathsf{N} \mathsf{H}'; \overline{\mathsf{l}} \qquad \mathsf{l}_\mathsf{T} \notin \mathsf{dom}(\mathsf{H}') \qquad \overline{\mathsf{f}} = \mathsf{dom}(\mathsf{fields}(\mathsf{T}, \mathsf{l}_\mathsf{T})) \qquad \mathsf{H}'' = \mathsf{H}', \mathsf{l}_\mathsf{T} \mapsto \{\overline{\mathsf{f}} \mapsto \overline{\mathsf{l}}\}}{\mathsf{H}; \mathtt{new}\ \mathsf{T}(\overline{e}) \Downarrow_\mathsf{N} \mathsf{H}''; \mathsf{l}_\mathsf{T}}$$

[RED-ASS] $$\frac{\mathsf{H}; e \Downarrow_\mathsf{N} \mathsf{H}'; \mathsf{l} \qquad \mathsf{H}'; e' \Downarrow_\mathsf{N} \mathsf{H}''; \mathsf{l}'}{\mathsf{H}; e.\mathsf{f} = e' \Downarrow_\mathsf{N} \mathsf{H}''[\mathsf{l} \mapsto \mathsf{H}''(\mathsf{l})[\mathsf{f} \mapsto \mathsf{l}']]; \mathsf{l}'}$$

[RED-SEL] $$\frac{\mathsf{H}; e \Downarrow_\mathsf{N} \mathsf{H}'; \mathsf{l}}{\mathsf{H}; e.\mathsf{f} \Downarrow_\mathsf{N} \mathsf{H}'; \mathsf{H}'(\mathsf{l})(\mathsf{f})}$$

Finally we formalize some of the key properties of the type system. We present a standard subject reduction result in Theorem 1, together with a statement that goodness of a heap is invariant through expression reductions. This implies that the heap invariants are maintained through program execution.

Theorem 1 (Subject Reduction). *Given* $\vdash \mathsf{P}$ *and* $\vdash \mathsf{H}$, *if* $\bullet; \mathsf{N} \vdash \mathsf{e} : \mathsf{T}$ *and* $\mathsf{H}; \mathsf{e} \Downarrow_{\mathsf{N}} \mathsf{H}'; \mathsf{l}$ *then* $\bullet; \mathsf{N} \vdash \mathsf{l} : \mathsf{T}'$ *for some* T' *such that* $\bullet \vdash \mathsf{T}' <: \mathsf{T}$ *and* $\vdash \mathsf{H}'$.

Proof. The proof proceeds by induction on the form of $\mathsf{H}; \mathsf{e} \Downarrow_{\mathsf{N}} \mathsf{H}'; \mathsf{l}$. Notice the heap needs to be well-formed over reduction to maintain the accessibility invariant.

Theorem 2 is the accessibility invariant enforced by the type system, which is proved as part of Theorem 1.

Theorem 2 (Accessibility Invariant). *Given* $\vdash \mathsf{P}$ *and* $\vdash \mathsf{H}$, *if* $(\mathsf{f} \mapsto \mathsf{l}') \in \mathsf{H}(\mathsf{l})$ *then* $\bullet \vdash \mathsf{l} \preceq \mathsf{acc}(\mathsf{l}')$.

Proof. This property is enforced by the `[HEAP]` rule and proved in Theorem 1.

5 Discussion and Related Work

Object encapsulation enforces a separation between the internal state of an object, and external dependencies. Ownership types achieve object encapsulation by establishing an object ownership tree, and in the owners-as-dominators model, prevent object references from breaching the encapsulated state. Ownership types use the ability to name objects as owners, to permit access to the objects they own. Ownership types can be considered as an access control system where other objects are permitted to access an object if they can name all of its context arguments, including its owner. The reference capability of an object is determined by its actual context arguments; these are used by the object as permissions for accessing other objects. In ownership types an object's accessibility and capability are essentially the same thing — as determined by the actual context arguments of the object's ownership type.

In this paper, we have separated accessibility and capability by introducing the concept of access modifier. The capability of an object remains the same as in ownership types, although now the context arguments can be abstract or variant from the site of use. However accessibility to the object now requires the ability of other objects to name its access modifier. Moreover, to completely free accessibility from capability, the access modifier is not declared in the object's class definition, that is, it is not part of its formal capability. Accessibility to an object is therefore independent of the reference capability of the object. The access context is the only context that must be named in order to access the object; this yields a much more flexible access control policy. Note that the access modifier cannot be abstracted — it must be named to gain access. The capability (context arguments) can be abstract or variant to express less rigid reference structures as we have seen from the examples. Moreover, an object's accessibility also implies its lifetime. The separation of accessibility from capability naturally means an object's lifetime is independent from its capability, but solely dependent on its accessibility.

The soundness of our approach lies in the fact that an object can only be accessible to those objects created (directly or indirectly) by the owner of the

object. This is because the owner's internal context can only be named from within the owner. This highlights the role of the creator — only the creator can authorize the created objects to access its own representation by defining their accessibility and capability appropriately.

Obviously, our type system subsumes ownership types; ownership types are special cases of our type system where the access modifier is the same as the owner context and no context argument is abstracted. Moreover, the techniques used in our type system may be applicable in other similar type systems for more flexibility and expressiveness, such as Effective Ownership [17], Acyclic Types [16] and Ownership Domains [1].

Aldrich and Chambers noted that ownership types cannot express the event callback design pattern [1]. Typically, a callback object is created by a listener object to observe some event. In the event, the callback object is invoked and will notify the listener object. Callback objects share some of the problems of iterators. The problem occurs when the callback object needs to directly mutate the listener's internal representation rather than use the listener's interface. The callback problem does not have such a serious performance issue as iterators do. The issue here is really about adding some flexibility to the callback classes. For example, instead of adding more methods (to be called by callback objects in different events) into the listener class, each callback class may implement its own code to mutate its listener. In our system, callback objects can be expressed in exactly the same way as iterators — we may simply promote the access modifier (permitted by the listener object) of callback objects high enough in the ownership hierarchy so that they can be named by the user of the callback objects.

Syntactic overhead for our types is that of ownership types plus an extra access modifier for each type. As we have seen, with carefully selected defaults type annotations can be reduced significantly. For instance, access modifiers can be omitted for globally accessible objects; abstract contexts can be omitted completely. Moreover, the ideas of *Generic Ownership* [21] can also be employed here to reduce the amount of type annotations in the presence of class type parameters.

As for ownership types, our type system allows separate compilation. It is statically checkable and does not require any runtime support. Our dynamic semantics can easily handle typecasting with runtime checks because it incorporates full owner information. However, in practice, the overhead for having owner information available at runtime may be significant for systems with a large number of small objects because each object will have two extra fields to identify its owner and access contexts. In security sensitive applications, this cost may well be worthwhile.

5.1 Related Work

Ownership Type Systems Without Owners-As-Dominators. There have been a number of proposals made to improve the expressiveness of ownership types. Some of them tend to break the strong owners-as-dominators encapsula-

tion of ownership types. Some of them tend to retain the owners-as-dominators encapsulation by using harmless cross-encapsulation references. We will discuss each of them individually.

Most proposals to break strong encapsulation of owners-as-dominators are essentially methods to increase nameability of internal contexts. Our proposal is also an owners-as-dominators encapsulation breaking technique. The difference is that we do not expose internal names, but use abstraction to hide the names of internal contexts. Compared to previous attempts, our type system appears to be more flexible and less ad hoc.

JOE [9] allows internal contexts to be named through read-only local variables (variables that cannot be assigned after initialization) so that internal representation can be accessed from outside; the justification for this approach is that encapsulation breeches are localized on the stack. The following code shows a simple example of JOE, where a method parameter is used to name the owner context of the `Node` object.

```
void joe(List<o, d> list) { Node<list, d> node = list.head; }
```

Ownership Domains [1] use a similar method where read-only fields (final fields in Java) are used to name internal domains (partitions of contexts) instead of read-only variables. The effect of moving variables to fields allows ownership domains types to have a more flexible reference structure than ownership types and JOE. To provide some safety with this approach, only domains declared as public can be named via final fields. Access policy between domains is explicitly declared and public domains are typically linked to private domains (which are unnameable from outside). For soundness, object creation is restricted to the owner domain of the current object or its locally defined subdomains. The following code shows a simple example of ownership domains. Iterator objects are created in a public domain of the list object and used as interface objects by the client. Note that a subclass of `Iterator` is needed to propagate the name of the private domain `owned` (as an extra domain parameter) to the iterator objects. We consider this to be a limited version of our context abstraction: essentially the `Iterator` interface hides the `Node` owner that is a required capability for the `ListIterator` object.

```
class List<o, d> assumes o->d {
  domain owned; link owned->d;
  public domain iters; link iters->owned, iters->d;
  Node<owned, d> head;
  Iterator<iters, d> getIter() {
    return new ListIterator<iters, d, owned>(head); } }

// in client class
final Link<some, world> list = ...
Iterator<list.iters, world> iter = list.getIter();
```

In practice, some problems may arise with read-only variables/fields. For example, in order to access an object in a context/public domain, it must firstly obtain a reference to the owner object of the context and then must place the

owner in a read-only variable. Only in this way can the context be named through the name of the read-only variable and a valid type be constructed for the object to be accessed. When accessing an object buried deep in the ownership tree, the programmer may need to declare many read-only variables and obtain references to each object along an ownership branch.

Moreover, the restriction on where objects can be created may limit some common programming practices, the factory design pattern for instance, where objects need to be created in various contexts/domains given by clients. The explicitly defined domains add finer-grained structures to the system at the cost of more domain and link annotations. Domains can be used to express some architectural constraints more precisely than ownership types do, because these constraints can be expressed directly as links which defines access policy between each pair of domains.

The *inner class* solution was suggested for ownership types by Clarke in his PhD thesis [7] and adopted by Boyapati et al. [4]. The idea of inner classes is very simple; inner classes can name the outer object directly. The following example shows the `Iterator` class is written as an inner class of the `List` class, who can name the list object's internal context directly via `List.this`.

```
class List<o, d> {
  Node<this, d> head;
  class Iterator<o, d> { Node<List.this, d> current; ... } }
```

Inner classes are lexically scoped and can only be used in limited places where the usage of the objects are specific and can be foreseen by the programmer. In general they are not as flexible as our type system.

The closest work to our type system may be the model of *Simple Ownership Types* [10]. In this model, there is a separation of *owner* and *rep* contexts. The owner context of the object determines which other objects may access it (like our accessibility context), while the rep context determines those contexts it may access (like `this`). The containment invariant states that if $\mathtt{l}$ references $\mathtt{l}'$ then $\mathtt{rep}(\mathtt{l}) \preceq \mathtt{owner}(\mathtt{l}')$. There are a number of major differences between the two models. The owner context in simple ownership types is a formal parameter of the class definition; it controls access to the object rather than defining the containment structure of objects/contexts — we prefer to reserve the notion of owner for the latter role. To preserve soundness this prevents the owner context from being variant. Moreover, as for all context parameters, the owner context must be a dominator of the rep context (which can be thought of as the object itself). Although simple ownership types use an explicit form of existential types to hide the internal contexts, it does not support variance of context arguments, as our system does.

Ownership Type Systems With Owners-As-Dominators. The proposals to retain the owners-as-dominators encapsulation allow some references to cross encapsulation boundary but ensure these reference cannot update the internal states directly (called *observational representation exposure* in [3]), that is, any update still has to be initialized by the owner object.

The *Universes* System [18, 19] uses read-only references to cross the boundary of encapsulation. Its read-only references are restricted and can only return read-only references. For example, they are able to express iterators by using a read-only reference to access the internal implementation of the list object. However, these iterators can only return data elements in read-only mode, that is, the elements in the list cannot be updated in this way (unless using dynamic casting with its associated runtime overheads [19]).

Effective Ownership [17] employs an encapsulation-aware effect system which allows arbitrary reference structure but still retain an owners-as-dominators encapsulation on object representation. It guarantees that any update to an object's internal state must occur (directly or indirectly) via a call on a method of its owner. In contrast to Universes, effective ownership's cross-encapsulation references can be used to mutate data elements via references held by a list object, while still protecting the list's representation from being modification. One limitation of effective ownership is that the iterator objects cannot be used to update the list's internal implementation, such as adding or removing elements from the list, because of the strong owners-as-dominators effect encapsulation.

Other Type Systems for Alias Protection. Many type systems have been proposed for alias protection. Confined types [26] manage aliases based on a package level encapsulation which provides a lightweight but weaker constraint than instance-level object encapsulation. Uniqueness techniques [27, 13] allow local reasoning and can prevent representation exposure by forbidding sharing; a reference is unique if it is the only reference to an object. External uniqueness combines the benefit of ownership types with uniqueness [8]. Boyland et al designed a system to unify uniqueness and read-only capability [5] where a reference is combined with a capability. Adoption [12, 6] can be used to provide information hiding similar to object encapsulation, but it is not clear how common object-oriented patterns such as iterators can be expressed in this approach. Alias types [23, 28] allow fine control on aliases at the cost of more complex annotations.

Variance on Parametric Types. The idea of use-site variance on type arguments of parametric class was first introduced informally by Thorup and Torgersen but only for covariance [24]. Igarashi and Viroli added contravariance and bivariance (complete abstraction) in their *Variant Parametric Types* and formalized type variances as bounded existential types. Our type system uses a similar technique, where, instead of types with subtyping, we rely on the containment ordering of ownership. Usually type systems with existential types would need some form of pack/unpack or close/open pairing to distinguish between typing contexts where the existential type is visible or not. Igarashi and Viroli used this idea directly in their type system, without exposing the existential types in the language syntax. Our use of the *, K+ and K− context variances in the language syntax and ?, K+? and K−? 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 (for example, K+? denotes an anonymous context which is inside K). Moreover since anonymous existential contexts are not bound to an environment they naturally become global, in other words, there is no need to pack/close them.

6 Conclusion

This paper has presented *variant ownership types* that generalize ownership types by separating accessibility of a type from its capability. Combined with context variance, the resulting type system significantly improves the expressiveness and utility of ownership types. The authors wish to acknowledge the support of the Australian Research Council Grant DP0665581.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *In European Conference on Object-Oriented Programming (ECOOP)*, July 2004.
2. P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.
3. A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 35–49. ACM Press, 2004.
4. 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.
5. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *In European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, 2001.
6. J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 283–295, New York, NY, USA, 2005. ACM Press.
7. D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
8. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *In European Conference on Object-Oriented Programming (ECOOP)*, July 2003.
9. D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
10. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.
11. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, 1998.

12. M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, 2002. ACM Press.
13. J. Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.
14. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.
15. A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 441–469. Springer-Verlag, 2002.
16. Y. Lu and J. Potter. A type system for reachability and acyclicity. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 479–503. Springer-Verlag, 2005.
17. Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2006.
18. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. *Programming Languages and Fundamentals of Programming*, 1999.
19. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
20. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.
21. A. Potanin, J. Noble, and R. Biddle. Generic ownership: practical ownership control in programming languages. In *OOPSLA Companion*, pages 50–51, 2004.
22. J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*. IEEE Press, 1998.
23. F. Smith, D. Walker, and G. Morrisett. Alias types. *Lecture Notes in Computer Science*, 1782:366–381, 2000.
24. K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP*, pages 186–204, 1999.
25. M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the java programming language. In *SAC*, pages 1289–1296, 2004.
26. J. Vitek and B. Bokowski. Confined types. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 82–96. ACM Press, 1999.
27. P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.
28. D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177–206, 2001.