Ownership Types for Object Encapsulation

Chandrasekhar Boyapati
Laboratory for Computer Science
Massachusetts Intitute of Technology
Cambridge, MA 02139
chandra@lcs.mit.edu

Barbara Liskov
Laboratory for Computer Science
Massachusetts Intitute of Technology
Cambridge, MA 02139
liskov@lcs.mit.edu

Liuba Shrira
Department of Computer Science
Brandeis University
Waltham, MA 02454
liuba@cs.brandeis.edu

Abstract

Ownership types provide a statically enforceable way of specifying object encapsulation and enable local reasoning about program correctness in object-oriented languages. However, a type system that enforces strict object encapsulation is too constraining: it does not allow efficient implementation of important constructs like iterators. This paper argues that the right way to solve the problem is to allow objects of classes defined in the same module to have privileged access to each other's representations; we show how to do this for inner classes. This approach allows programmers to express constructs like iterators and yet supports local reasoning about the correctness of the classes, because a class and its inner classes together can be reasoned about as a module. The paper also sketches how we use our variant of ownership types to enable efficient software upgrades in persistent object stores.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs;
D.2.4 [**Software Engineering**]: Program Verification

General Terms

Languages, Verification, Theory

Keywords

Ownership Types, Object Encapsulation, Software Upgrades

1 Introduction

The ability to reason locally about program correctness is crucial when dealing with large programs. Local reasoning allows correctness to be dealt with one module at a time. Each module has a specification that describes its expected behavior. The goal is to prove that each module satisfies its specification, using only the specifications but not code of other modules. This way the complexity of the proof effort (formal or informal) can be kept under control.

This local reasoning approach is sound if separate verification of individual modules suffices to ensure the correctness of the composite program [43, 28]. The key to sound local reasoning in object-oriented languages is object encapsulation. Consider, for example, a Stack object s that is implemented using a linked list. Local reasoning about the correctness of the Stack implementation is possible if objects outside s do not directly access the list nodes, i.e., the list nodes are *encapsulated* within the s.

This paper presents a variant of ownership types for specifying and statically enforcing object encapsulation. With ownership types, a program can declare that s *owns* all the list nodes. The type system then statically ensures that the list nodes are encapsulated within s.

A type system that enforces strict object encapsulation, however, is too constraining [55]: it does not allow efficient implementation of important constructs like iterators [48, 32]. Consider, for example, an iterator over the above-mentioned Stack object s. If the iterator is encapsulated within s, it cannot be used outside s. If the iterator is *not* encapsulated within s, it cannot directly access the list nodes in s, and hence cannot run efficiently.

Previous ownership type systems were either too constraining to support constructs like iterators [22, 21], or too permissive to support local reasoning [20, 14, 11]; for example they allowed objects outside the above-mentioned Stack object s to temporarily get direct access to the list nodes.

This paper argues that the right way to solve the problem is to provide special access privileges to objects belonging to classes in the same module; we show how to do this for inner classes [50, 38]. Our variant of ownership types allows inner class objects to have privileged access to the representations of the corresponding outer class objects. This principled violation of encapsulation allows programmers to express constructs like iterators using inner classes, yet supports local reasoning about the correctness of the classes. Our system supports local reasoning because a class and its inner classes can be reasoned about together as a module.

The paper also describes how our variant of ownership types enables efficient software upgrades in persistent object stores.

The research was supported in part by DARPA Contract F30602-98-1-0237, NSF Grant IIS-98-02066, and NTT.

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright 2003 ACM 1-58113-628-5/03/0001 ...$5.00

Our interest in software upgrades led us to work on ownership types. The paper shows how our ownership types can be used to ensure that code for upgrading objects does not observe broken invariants or interfaces unknown at the time it was written; this makes it possible for programmers to reason about the correctness of their upgrades.

This paper is organized as follows. Section 2 discusses object encapsulation. Section 3 describes our variant of ownership types for enforcing object encapsulation. Section 4 presents a formal description of the type system. Section 5 shows how ownership types can be used to enable modular upgrades. Section 6 discusses related work and Section 7 concludes.

2 Object Encapsulation

Object encapsulation is important because it provides the ability to reason locally about program correctness. Reasoning about a class in an object-oriented program involves reasoning about the behavior of objects belonging to the class. Typically objects point to other *subobjects*, which are used to represent the containing object.

Local reasoning about class correctness is possible if the subobjects are *fully encapsulated*, that is, if all subobjects are accessible only within the containing object. This condition supports local reasoning because it ensures that outside objects cannot interact with the subobjects without calling methods of the containing object. And therefore the containing object is in control of its subobjects.

However, full encapsulation is often more than is needed. Encapsulation is only required for subobjects that the containing object *depends* on [43, 28]:

D1. An object x *depends* on subobject y if x calls methods of y and furthermore these calls expose mutable behavior of y in a way that affects the invariants of x.

Thus, a Stack object s implemented using a linked list depends on the list but not on the items contained in the list. If code outside could manipulate the list, it could invalidate the correctness of the Stack implementation. But code outside can safely use the items contained in s because s doesn't call their methods; it only depends on the identities of the items and the identities never change. Similarly, a Set of immutable elements does not depend on the elements even if it invokes a.equals(b) to ensure that no two elements a and b in the Set are equal, because the elements are immutable.

Local reasoning about a class is possible if objects of that class encapsulate every object they depend on. But strict object encapsulation is too constraining [55]: it prevents efficient implementation of important constructs like iterators. For example, to run efficiently, an iterator over the abovementioned Stack object s needs access to the list nodes in s. To provide this access, we have to allow objects like iterators to violate encapsulation.

O1. Every object has an owner.

O2. The owner can either be another object or world.

O3. The owner of an object does not change over time.

O4. The ownership relation forms a tree rooted at world.

Figure 1: Ownership Properties

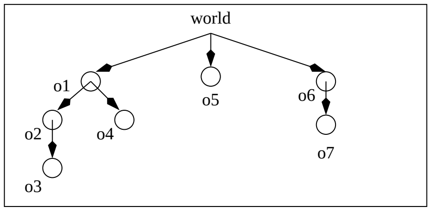

Figure 2: An Ownership Relation

Local reasoning is still possible provided all violations of encapsulation are limited to code contained in the same module. For example, if both the Stack and its iterator are implemented in the same module, we can still reason about their correctness locally, by examining the code of that module.

3 Ownership Types for Encapsulation

Ownership types [22, 20, 14, 11] provide a statically enforceable way of specifying object encapsulation. The idea is that an object can *own* subobjects it depends on, thus preventing them from being accessible outside. This section presents our ownership type system. This system is similar to the one described in [20]—the main difference is that to support constructs like iterators, the type system in [20] allows temporary violations of encapsulation. We disallow this violation. Instead, we support constructs like iterators using inner classes.

The key to the type system is the concept of object ownership. Every object has an owner. The owner can either be another object or a special owner called world. Our type system statically guarantees the ownership properties shown in Figure 1. Figure 2 presents an example ownership relation. We draw an arrow from x to y if x owns y. In the figure, the special owner world owns objects o1, o5, and o6; o1 owns o2 and o4; o2 owns o3; and o6 owns o7.

Ownership allows a program to statically declare encapsulation boundaries that capture dependencies:

D2. An object should own all the objects it depends on.

The system then enforces encapsulation: if y is inside the encapsulation boundary of z and x is outside, then x cannot access y. (An object x *accesses* an object y if x has a pointer to y, or methods of x obtain a pointer to y.) In Figure 2, o7 is inside the encapsulation boundary of o6 and o1 is outside, so o1 cannot access o7. An object is only allowed to access: 1) itself and objects it owns, 2) its ancestors in the ownership tree and objects they own, and 3) globally accessible objects, namely objects owned by world.[1] Thus, o1 can access all objects in the figure except for o3 and o7.

[1] Note the analogy with nested procedures: proc P_1 {var x_2;

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode =
      new TNode<this, TOwner>(value, head);
    head = newNode;
  }
  T<TOwner> pop() {
    if (head == null) return null;
    T<TOwner> value = head.value();  head = head.next();
    return value;
  }
}

class TNode<nodeOwner, TOwner> {
  TNode<nodeOwner, TOwner> next;  T<TOwner> value;

  TNode(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;  this.next  = n;
  }
  T<TOwner> value()                { return value; }
  TNode<nodeOwner, TOwner> next() { return next;  }
}

class T<TOwner> { }

class TStackClient<clientOwner> {
  void test() {
    TStack<this,  this>  s1 = new TStack<this,  this> ();
    TStack<this,  world> s2 = new TStack<this,  world>();
    TStack<world, world> s3 = new TStack<world, world>();
 /* TStack<world, this>  s4 = new TStack<world, this> (); */
}}
```

Figure 3: Stack of T Objects

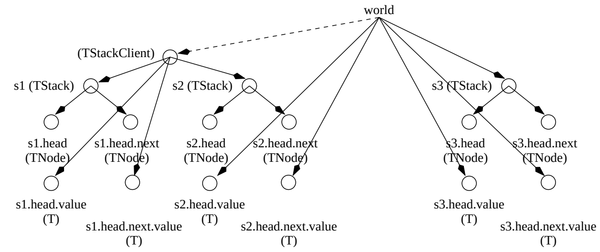

Figure 4: Ownership Relation for TStacks s1, s2, s3

3.1 Owner Polymorphism

We present our type system in the context of a Java-like language augmented with ownership types. Every class definition is parameterized with one or more owners. The first owner parameter is special: it identifies the owner of the corresponding object. The other owner parameters are used to propagate ownership information. Parameterization allows programmers to implement a generic class whose objects have different owners. This parameterization is similar to parametric polymorphism [54, 16, 1, 61] except that our parameters are owners, not types.

An owner can be instantiated with this, with world, or with another owner parameter. Objects owned by this are encapsulated objects that cannot be accessed from outside. Objects owned by world can be accessed from anywhere.

```
class C<cOwner, sOwner, tOwner> where (sOwner <= tOwner) {
    ...
    TStack<sOwner, tOwner> s;
}
```

Figure 5: Using Where Clauses to Constrain Owners

Figure 3 shows an example.[2] A TStack is a stack of T objects. It is implemented using a linked list. The TStack class is parameterized by stackOwner and TOwner. stackOwner owns the TStack object; TOwner owns the T objects contained in the TStack. The code specifies that the TStack object owns the nodes in the list; therefore the list nodes cannot be accessed from outside the TStack object.

The type of TStack s1 is instantiated using this for both the owner parameters. This means that TStack s1 is owned by the TStackClient object that created it and so are the T objects in s1. TStack s2 is owned by the TStackClient object, but the T objects in s2 are owned by world. TStack s3 is owned by world and so are the T objects in s3. The ownership relation for s1, s2, and s3 is depicted in Figure 4 (assuming the stacks contain two elements each). (The dotted line indicates that every object is directly or indirectly owned by world.)

3.2 Constraints on Owners

For every type $T\langle x_1, ..., x_n\rangle$ with multiple owners, our type system statically enforces the constraint that $(x_1 \preceq x_i)$ for all $i \in \{1..n\}$. Recall from Figure 1 that the ownership relation forms a tree rooted at world. The notation $(y \prec z)$ means that y is a descendant of z in the ownership tree. The notation $(y \preceq z)$ means that y is either the same as z, or y is a descendant of z in the ownership tree. Thus, the type of TStack s4 in Figure 3 is illegal because (world $\not\preceq$ this).

The above constraint is the same as in [20]. However, we extend it to parameterized methods as well. For a method $m\langle x_{n+1}, ..., x_k\rangle(...)\{...\}$ of an object of type $T\langle x_1, ..., x_n\rangle$, the restriction is that $(x_1 \preceq x_i)$ for all $i \in \{1..k\}$. (These constraints are needed to provide encapsulation in the presence of subtyping. [11] illustrates this point with an example.)

To check ownership constraints modularly, it is sometimes necessary for programmers to specify additional constraints on class and method parameters. For example, in Figure 5, the type of s is legal only if (sOwner $\preceq$ tOwner). We allow programmers to specify such additional constraints using where clauses [25, 54], and our type system enforces the constraints. For example, in Figure 5, class C specifies that (sOwner $\preceq$ tOwner). An instantiation of C that does not satisfy the constraint is illegal.

3.3 Subtyping

The rule for declaring a subtype is that the first owner parameter of the supertype must be the same as that of the subtype; in addition, of course, the supertype must satisfy the constraints on owners. The first owners have to match

proc P$_2$ {var x$_3$; proc P$_3$ {...}}}. Say x$_{n+1}$ and P$_{n+1}$ are children of P$_n$. P$_n$ can only access: 1) P$_n$ and its children, 2) the ancestors of P$_n$ and their children, and 3) global variables and procedures.

[2] The example shows type annotations written explicitly. However, many of them can be automatically inferred. See Section 4.3 for details.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TStackEnum<enumOwner, TOwner> elements<enumOwner>()
    where (enumOwner <= TOwner) {
    return new TStackEnum<enumOwner, TOwner>();
  }
  class TStackEnum<enumOwner, TOwner>
    implements TEnumeration<enumOwner, TOwner> {

    TNode<TStack.this, TOwner> current;

    TStackEnum() { current = TStack.this.head; }

    T<TOwner> getNext() {
      if (current == null) return null;
      T<TOwner> t = current.value();
      current = current.next();
      return t;
    }
    boolean hasMoreElements() {
      return (current != null);
    }
  }
}

class TStackClient<clientOwner> {
  void test() {
    TStack<this, world> s = new TStack<this, world>();
    TEnumeration<this,  world> e1 = s.elements();
    TEnumeration<world, world> e2 = s.elements();
}}

interface TEnumeration<enumOwner, TOwner> {
  T<TOwner> getNext();
  boolean   hasMoreElements();
}
```

Figure 6: TStack With Iterator

because they are special, in that they own the corresponding objects. Thus, TStack⟨stackOwner, TOwner⟩ is a subtype of Object⟨stackOwner⟩. But T⟨TOwner⟩ is not a subtype of Object⟨world⟩ because the first owners do not match.

3.4 Inner Classes

Our inner classes are similar to the member inner classes in Java. Inner class definitions are nested inside other classes. Figure 6 shows an example. The inner class TStackEnum implements an iterator for TStack; the elements method of TStack provides a way to create an iterator over the TStack. The TStack code is otherwise similar to that in Figure 3.

Recall from before that an owner can be instantiated with this, with world, or with another owner parameter. Within an inner class, an owner can also be instantiated with C.this, where C is an outer class. This feature allows an inner object to access the objects encapsulated within its outer objects. In Figure 6, the owner of the current field inTStackEnum is instantiated with TStack.this. The current field accesses list nodes encapsulated within its outer TStack object.

An inner class is parameterized with owners just like a regular class. In our system, the outer class parameters are not automatically visible inside an inner class. If an inner class uses an outer class parameter, it must explicitly include the outer class parameter in its declaration. In Figure 6, the TStackEnum declaration includes the owner parameter TOwner from its outer class. TOwner is therefore visible inside TStackEnum. But the TStackEnum declaration does not include stackOwner. Therefore, stackOwner is not visible inside TStackEnum.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  class TStackEnum<enumOwner, TOwner>
   implements TEnumeration<enumOwner, TOwner> {

    TNode<TStack.this, TOwner> current;
    ...
    T<TOwner> getNext() writes(this) reads(TStack.this){...}
    boolean   hasMoreElements()      reads(this){...}
  }
}

interface TEnumeration<enumOwner, TOwner> {
  T<TOwner> getNext() writes(this) reads(world);
  boolean   hasMoreElements()      reads(this);
}
```

Figure 7: TStack Iterator With Effects

Note that in this example, the elements method is parameterized by enumOwner. This allows a program to create different iterators that have different owners. elements returns an iterator of type TStackEnum⟨enumOwner, TOwner⟩. For this type to be legal, it must be the case that (enumOwner $\preceq$ TOwner). This requirement is captured in the where clause.

Note also that TStack⟨stackOwner, TOwner⟩.TStackEnum⟨enumOwner, TOwner⟩ is declared to be a subtype of TEnumeration⟨enumOwner, TOwner⟩. This allows TStackClient to create an unencapsulated iterator e2 over an encapsulated TStack s; the program can then pass e2 to objects outside the TStackClient. In general, inner classes can be used to implement wrappers [32] that expose a limited interface to an outer object. A program can then create a wrapper around an encapsulated subobject, and pass the wrapper object outside the encapsulation boundary.

3.5 Encapsulation Theorem

Our system provides the following encapsulation property:

THEOREM 1. *x can access an object owned by o only if:*

1. *($x \preceq o$), or*
2. *x is an inner class object of o.*

PROOF. Consider the code: *class* $C\langle f, ...\rangle\{...\ T\langle o, ...\rangle\ y\ ...\}$. Variable y of type $T\langle o, ...\rangle$ is declared within the static scope of class C. Owner o can therefore be either 1) this, or 2) world, or 3) a formal class parameter, or 4) a formal method parameter, or 5) C'.this, where C' is an outer class. We will show that in the first four cases, the constraint (this $\preceq o$) holds. In the first two cases, the constraint holds trivially. In the last two cases, ($f \preceq o$) and (this $\prec f$), so the constraint holds. In the fifth case, (C'.this $= o$). Therefore an object x of a class C can access an object y owned by o only if either 1) ($x \preceq o$), as in the first four cases, or 2) x is an inner object of o, as in the fifth case. □

```
class IntVector<vOwner> {
  int  elementCount = 0;
  int  size()     reads (this) { return elementCount; }
  void add(int x) writes(this) { elementCount++;  ... }
  ...
}

class IntStack<sOwner> {
  IntVector<this> vec = new IntVector<this>();
  void push(int x) writes (this) { vec.add(x); }
  ...
}

void m<sO,vO> (IntStack<sO> s, IntVector<vO> v)
  writes (s) reads (v) where !(v <= s) !(s <= v) {

  int n = v.size(); s.push(3); assert(n == v.size());
}
```

Figure 8: Reasoning About Aliasing and Side Effects

3.6 Discussion

Our variant of ownership types supports local reasoning provided the programmer declares that all depended-on objects are owned. The above theorem implies that owned objects can only be accessed from inside the owner, and by inner objects. Therefore if ownership captures the depends relation described in Section 2, local reasoning about the correctness of a class is possible, because the class and its inner classes together can be reasoned about as a module.

Our ownership types are also expressive. They allow efficient implementation of constructs like iterators and wrappers [32]. Furthermore, they also allow programs to create wrappers that can be used in contexts where the underlying object is inaccessible. This ability was illustrated in Figure 6; iterator e2 can be used globally even though the TStack it is iterating over can only be used in TStackClient.

Ours is the first ownership type system to support constructs like iterators and generally accessible wrappers while also ensuring local reasoning. We discuss this further in Section 6.

3.7 Effects Clauses

Our system also contains effects clauses [49] because they are useful for specifying assumptions that hold at method boundaries and enable modular reasoning and checking of programs. We use effects with ownership types to enable modular upgrades; we describe this in Section 5.

Our system allows programmers to specify *reads* and *writes* clauses. Consider a method that specifies that it writes $(w_1, ..., w_n)$ and reads $(r_1, ..., r_m)$. The method can write an object x (or call methods that write x) only if $(x \preceq w_i)$ for some $i \in \{1..n\}$. The method can read an object y (or call methods that read y) only if $(y \preceq w_i)$ or $(y \preceq r_j)$, for some $i \in \{1..n\}$, $j \in \{1..m\}$. We thus allow a method to both read and write objects named in its writes clause.

Figure 7 shows a TStack iterator that uses effects, but is otherwise similar to the TStack iterator in Figure 6. In the example, the hasMoreElements method reads the this object. The getNext method reads objects owned by TStack.this and writes (and reads) the this object.

$$\begin{array}{rcl}
P & ::= & \mathit{defn}^* \; e \\
\mathit{defn} & ::= & \textsf{class } \mathit{cn}\langle \mathit{formal}{+}\rangle \textsf{ extends } c \textsf{ where } \mathit{constr}^* \; \mathit{body} \\
\mathit{body} & ::= & \{\mathit{innerclass}^* \; \mathit{field}^* \; \mathit{meth}^*\} \\
c & ::= & \mathit{cn}\langle \mathit{owner}{+}\rangle \mid \textsf{Object}\langle \mathit{owner}{+}\rangle \mid c.\mathit{cn}\langle \mathit{owner}{+}\rangle \\
\mathit{owner} & ::= & \mathit{formal} \mid \textsf{world} \mid \mathit{cn}.\textsf{this} \\
\mathit{constr} & ::= & (\mathit{owner} \preceq \mathit{owner}) \mid (\mathit{owner} \not\preceq \mathit{owner}) \\
\mathit{innerclass} & ::= & \mathit{defn} \\
\mathit{meth} & ::= & t \; \mathit{mn}\langle \mathit{formal}^*\rangle(\mathit{arg}^*) \; \mathit{effects} \textsf{ where } \mathit{constr}^* \; \{e\} \\
\mathit{effects} & ::= & \textsf{reads } (\mathit{owner}^*) \textsf{ writes } (\mathit{owner}^*) \\
\mathit{field} & ::= & t \; \mathit{fd} \\
\mathit{arg} & ::= & t \; x \\
t & ::= & c \mid \textsf{int} \\
\mathit{formal} & ::= & f \\
e & ::= & \textsf{new } c \mid x.\textsf{new } c \mid x \mid \textsf{let } (\mathit{arg}{=}e) \textsf{ in } \{e\} \mid \\
 & & x.\mathit{fd} \mid x.\mathit{fd} = y \mid x.\mathit{mn}\langle \mathit{owner}^*\rangle(y^*) \\
 & & \\
\mathit{cn} & \in & \text{class names} \\
\mathit{fd} & \in & \text{field names} \\
\mathit{mn} & \in & \text{method names} \\
x,y & \in & \text{variable names} \\
f & \in & \text{owner names}
\end{array}$$

Figure 9: Grammar

When effects clauses are used in conjunction with subtyping, the effects of an overridden method must subsume the effects of the overriding method. This sometimes makes it difficult to specify precisely all the effects of a method. For example, it is difficult to specify precisely all the read effects in the getNext method of the TEnumeration class because TEnumeration is expected to be a supertype of subtypes like TStackEnum and TEnumeration cannot name the specific objects used in the getNext methods of these subtypes. To accommodate such cases, we allow an escape mechanism, where a method can include world in its effects clauses.

Ownership types and effects can be used to locally reason about the side effects of method calls. Consider, for example, the code in Figure 8, which shows an IntStack implemented using an IntVector vec. (We adopted this example from [44].) The example has a method m that receives two arguments: an IntStack s and an IntVector v. The condition in the assert statement in m can be true only if v is not aliased to s.vec.

In the example, the method m uses a where clause to specify that (v $\not\preceq$ s) and (s $\not\preceq$ v). Since the ownership relation forms a tree (see Figure 1), this constraint implies that v cannot be aliased to s.vec. Furthermore, IntVector.size declares that it only reads objects owned by the IntVector, and IntStack.push declares that it only writes (and reads) objects owned by the IntStack. Therefore, it is possible to reason locally that v.size and s.push cannot interfere, and thus the condition in the assert statement in m must be true.

4 The Type System

This section presents a formal description of our type system. To simplify the presentation of key ideas, we describe our type system in the context of a core subset of Java [33] known as Classic Java [31]. We add inner classes to Classic Java and augment its type system with ownership types. Our approach, however, extends to the whole of Java and other similar languages.

4.1 Type Checking

Figure 9 presents our grammar. The core of our type system is a set of rules for reasoning about the typing judgment: P;

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{WFClasses(P)} \quad \mathit{ClassOnce(P)} \quad \mathit{IClassOnce(P)} \\ \mathit{FieldsOnce(P)} \quad \mathit{MethodsOnce(P)} \quad \mathit{OverridesOK(P)} \\ P = \mathit{defn}_{1..n}\ e \quad P \vdash \mathit{defn}_i \in \emptyset \quad P;\ \emptyset;\ \mathsf{world};\ \mathsf{world} \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash \mathit{defn} \in c}$

[CLASS]

$$\frac{\begin{array}{c} \mathrm{OFields}(c.cn\langle f_{1..n}\rangle) \overset{\mathrm{def}}{=} c.cn\langle f_{1..n}\rangle\ cn.\mathsf{this},\ \mathrm{OFields}(c) \qquad \mathrm{OFields}(\emptyset) \overset{\mathrm{def}}{=} \emptyset \\ E = \mathrm{OFields}(c.cn\langle f_{1..n}\rangle),\ \mathrm{owner}\ f_{1..n},\ \mathit{constr}* \qquad P;\ E \vdash \mathit{wf} \\ P;\ E \vdash c' \quad P \vdash \mathit{iclass}_i \in c.cn\langle f_{1..n}\rangle \quad P;\ E \vdash \mathit{field}_i \quad P;\ E \vdash \mathit{meth}_i \end{array}}{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c'\ \mathsf{where}\ \mathit{constr}*\ \{\mathit{iclass}*\ \mathit{field}*\ \mathit{meth}*\} \in c}$$

$\boxed{P;\ E \vdash \mathit{constr}}$

[$\preceq$ WORLD]

$$\frac{P;\ E \vdash_{\mathrm{owner}} o}{P;\ E \vdash (o \preceq \mathsf{world})}$$

[CONSTR ENV]

$$\frac{E = E_1,\ \mathit{constr},\ E_2}{P;\ E \vdash \mathit{constr}}$$

[$\preceq$ OWNER]

$$\frac{P;\ E \vdash e : cn\langle o_{1..n}\rangle}{P;\ E \vdash (e \preceq o_1)}$$

[$\preceq$ REFL]

$$\frac{P;\ E \vdash_{\mathrm{owner}} o}{P;\ E \vdash (o \preceq o)}$$

[$\preceq$ TRANS]

$$\frac{P;\ E \vdash (o_1 \preceq o_2) \quad P;\ E \vdash (o_2 \preceq o_3)}{P;\ E \vdash (o_1 \preceq o_3)}$$

$\boxed{P;\ E \vdash X \preceq Y}$

[$X \preceq Y$]

$$\frac{\begin{array}{c} X = x_{1..n} \quad Y = y_{1..m} \\ \forall_{i \in \{1..n\}}\ \exists_{j \in \{1..m\}}\ (x_i \preceq y_j) \end{array}}{P;\ E \vdash (X \preceq Y)}$$

$\boxed{P;\ E \vdash_{\mathrm{owner}} o}$

[OWNER WORLD]

$$\frac{}{P;\ E \vdash_{\mathrm{owner}} \mathsf{world}}$$

[OWNER FORMAL]

$$\frac{E = E_1,\ \mathrm{owner}\ f,\ E_2}{P;\ E \vdash_{\mathrm{owner}} f}$$

[OWNER THIS]

$$\frac{E = E_1,\ c\ cn.\mathsf{this},\ E_2}{P;\ E \vdash_{\mathrm{owner}} cn.\mathsf{this}}$$

$\boxed{P;\ E \vdash t}$

[TYPE INT]

$$\frac{}{P;\ E \vdash \mathsf{int}}$$

[TYPE OBJECT]

$$\frac{P;\ E \vdash_{\mathrm{owner}} o}{P;\ E \vdash \mathsf{Object}\langle o\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle ...\ \mathsf{where}\ \mathit{constr}*\ ... \in \overline{cn}\langle\overline{f}\rangle \qquad P;\ E \vdash \overline{cn}\langle\overline{o}\rangle \\ P;\ E \vdash_{\mathrm{owner}} o_i \quad P;\ E \vdash o_1 \preceq o_i \quad P;\ E \vdash \mathit{constr}\ [o_1/f_1]..[o_n/f_n] \\ (f_{ij} \in \overline{f}) \wedge (o_{ij} \in \overline{o}) \wedge (f_k = f_{ij}) \Longrightarrow (o_k = o_{ij}) \end{array}}{P;\ E \vdash \overline{cn}\langle\overline{o}\rangle.cn\langle o_{1..n}\rangle}$$

$\boxed{P;\ E \vdash t_1 <: t_2}$

[SUBTYPE C]

$$\frac{\begin{array}{c} P;\ E \vdash \overline{cn}\langle\overline{o}\rangle.cn\langle o_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ \overline{cn'}\langle\overline{o'}\rangle.cn'\langle f_1\ o*\rangle\ ... \in \overline{cn}\langle\overline{f}\rangle \end{array}}{P;\ E \vdash \overline{cn}\langle\overline{o}\rangle.cn\langle o_{1..n}\rangle <: \overline{cn'}\langle\overline{o'}\rangle.cn'\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE REFL]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t <: t}$$

[SUBTYPE TRANS]

$$\frac{\begin{array}{c} P;\ E \vdash t_1 <: t_2 \\ P;\ E \vdash t_2 <: t_3 \end{array}}{P;\ E \vdash t_1 <: t_3}$$

$\boxed{P;\ E \vdash \mathit{wf}}$

[ENV $\emptyset$]

$$\frac{}{P;\ \emptyset \vdash \mathit{wf}}$$

[ENV X]

$$\frac{\begin{array}{c} P;\ E \vdash \mathit{wf} \\ x \notin \mathrm{Dom}(E) \\ P;\ E \vdash t \end{array}}{P;\ E,\ t\ x \vdash \mathit{wf}}$$

[ENV OWNER]

$$\frac{\begin{array}{c} P;\ E \vdash \mathit{wf} \\ f \notin \mathrm{Dom}(E) \end{array}}{P;\ E,\ \mathrm{owner}\ f \vdash \mathit{wf}}$$

[ENV CONSTR]

$$\frac{\begin{array}{c} \mathit{constr} = (o \preceq o') \vee \mathit{constr} = (o \not\preceq o') \\ P;\ E \vdash \mathit{wf} \quad P;\ E \vdash_{\mathrm{owner}} o,\ o' \\ E' = E,\ \mathit{constr} \\ \nexists_{x,y}\ (P;\ E' \vdash x \preceq y) \wedge (P;\ E' \vdash x \not\preceq y) \end{array}}{P;\ E,\ \mathit{constr} \vdash \mathit{wf}}$$

$\boxed{P \vdash \mathit{meth} \in c}$

[METHOD DECLARED]

$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle ...\ \{...\ \mathit{meth}\ ...\} \in c}{P \vdash \mathit{meth} \in c.cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{\begin{array}{c} P \vdash \mathit{meth} \in \overline{cn}\langle\overline{f}\rangle.cn\langle f_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn'\langle g_{1..m}\rangle\ \mathsf{extends}\ \overline{cn}\langle\overline{o}\rangle.cn\langle o_{1..n}\rangle ... \in c' \end{array}}{P \vdash \mathit{meth}\ [o_1/f_1]..[o_n/f_n] \in c'.cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\ E \vdash \mathit{method}}$

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ \mathrm{owner}\ f_{1..n},\ \mathit{constr}*,\ \mathit{arg}* \\ P;\ E' \vdash \mathit{wf} \quad P;\ E';\ r*, w*;\ w* \vdash e : t \end{array}}{\begin{array}{r} P;\ E \vdash t\ mn\langle f_{1..n}\rangle(\mathit{arg}*)\ \mathsf{reads}(r*) \\ \mathsf{writes}(w*)\ \mathsf{where}\ \mathit{constr}*\ \{e\} \end{array}}$$

$\boxed{P \vdash \mathit{field} \in c}$

[FIELD DECLARED]

$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle ...\ \{...\ \mathit{field}\ ...\} \in c}{P \vdash \mathit{field} \in c.cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{\begin{array}{c} P \vdash \mathit{field} \in \overline{cn}\langle\overline{f}\rangle.cn\langle f_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn'\langle g_{1..m}\rangle\ \mathsf{extends}\ \overline{cn}\langle\overline{o}\rangle.cn\langle o_{1..n}\rangle ... \in c' \end{array}}{P \vdash \mathit{field}\ [o_1/f_1]..[o_n/f_n] \in c'.cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\ E \vdash \mathit{field}}$

[FIELD]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t\ fd}$$

$\boxed{P;\ E \vdash e : t}$

[EXP TYPE]

$$\frac{P;\ E;\ \mathsf{world};\ \mathsf{world} \vdash e : t}{P;\ E \vdash e : t}$$

$\boxed{P;\ E;\ R;\ W \vdash e : t}$

[EXP SUB]

$$\frac{\begin{array}{c} P;\ E;\ R;\ W \vdash e : t' \\ P;\ E;\ R;\ W \vdash t' <: t \end{array}}{P;\ E;\ R;\ W \vdash e : t}$$

[EXP REF]

$$\frac{\begin{array}{c} P;\ E;\ R;\ W \vdash x : \overline{cn}\langle\overline{o}\rangle \quad P \vdash (t\ fd) \in \overline{cn}\langle\overline{f}\rangle \\ R = R_1,\ r,\ R_2 \quad x \preceq r \end{array}}{P;\ E;\ R;\ W \vdash x.fd : t\ [\overline{o}/\overline{f}]}$$

[EXP REF ASSIGN]

$$\frac{\begin{array}{c} P;\ E;\ R;\ W \vdash x : \overline{cn}\langle\overline{o}\rangle \quad P \vdash (t\ fd) \in \overline{cn}\langle\overline{f}\rangle \\ W = W_1,\ w,\ W_2 \quad x \preceq w \quad P;\ E;\ R;\ W \vdash y : t\ [\overline{o}/\overline{f}] \end{array}}{P;\ E;\ R;\ W \vdash x.fd = y : t\ [\overline{o}/\overline{f}]}$$

[EXP NEW]

$$\frac{P;\ E \vdash cn\langle o_{1..n}\rangle}{P;\ E;\ R;\ W \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle : cn\langle o_{1..n}\rangle}$$

[EXP X.NEW]

$$\frac{P;\ E \vdash x : c \quad P;\ E \vdash c.cn\langle o_{1..n}\rangle}{P;\ E;\ R;\ W \vdash x.\mathsf{new}\ cn\langle o_{1..n}\rangle : c.cn\langle o_{1..n}\rangle}$$

[EXP VAR]

$$\frac{E = E_1,\ t\ x,\ E_2}{P;\ E;\ R;\ W \vdash x : t}$$

[EXP LET]

$$\frac{\begin{array}{c} \mathit{arg} = t\ x \quad P;\ E;\ R;\ W \vdash e : t \\ P;\ E,\ \mathit{arg};\ R;\ W \vdash e' : t' \end{array}}{P;\ E;\ R;\ W \vdash \mathsf{let}\ (\mathit{arg} = e)\ \mathsf{in}\ \{e'\} : t'}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j \in 1..k})\ \mathsf{reads}(r_{1..r})\ \mathsf{writes}(w_{1..w})\ \mathsf{where}\ \mathit{constr}*\ \{e\}) \in \overline{cn}\langle\overline{f}\rangle.cn\langle f_{1..n}\rangle \\ \begin{array}{ll} P;\ E;\ R;\ W \vdash x : \overline{cn}\langle\overline{o}\rangle.cn\langle o_{1..n}\rangle & P;\ E;\ R;\ W \vdash x_j : t_j\ [o_1/f_1]..[o_m/f_m] \\ P;\ E \vdash_{\mathrm{owner}} o_i \quad P;\ E \vdash o_1 \preceq o_i & P;\ E \vdash r_{1..r}\ [o_1/f_1]..[o_m/f_m] \preceq R \\ P;\ E \vdash \mathit{constr}\ [o_1/f_1]..[o_m/f_m] & P;\ E \vdash w_{1..w}\ [o_1/f_1]..[o_m/f_m] \preceq W \end{array} \end{array}}{P;\ E;\ R;\ W \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$

Figure 10: Type Checking Rules

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash \mathit{defn} \in c$	*defn* is a well-formed class in class c
$P;\, E \vdash_{\text{owner}} o$	o is an owner
$P;\, E \vdash \mathit{constr}$	constraint *constr* is satisfied
$P;\, E \vdash X \preceq Y$	effect X is subsumed by effect Y
$P;\, E \vdash t$	t is a well-formed type
$P;\, E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P;\, E \vdash \mathit{wf}$	typing environment E is well-formed
$P \vdash \mathit{field} \in c$	class c declares/inherits *field*
$P \vdash \mathit{meth} \in c$	class c declares/inherits *meth*
$P;\, E \vdash \mathit{field}$	*field* is a well-formed field
$P;\, E \vdash \mathit{meth}$	*meth* is a well-formed method
$P;\, E \vdash e : t$	expression e has type t
$P;\, E;\, R;\, W \vdash e : t$	expression e has type t and read/write effects of e are subsumed by R/W

Figure 11: Typing Judgments

$E;\, R;\, W \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. R and W must subsume the read and write effects of e. t is the type of e. We define a typing environment as $E ::= \emptyset \mid E,\, t\ x \mid E,\, \text{owner}\ f \mid E,\, \mathit{constr}$. We define effects as $R,\, W ::= o_{1..n}$. We define the type system using the judgments in Figure 11. We present the rules for these judgments in Figure 10. The rules use a number of predicates that are shown in Figure 12. These predicates are based on similar predicates from [31]. For simplicity, we sometimes treat outermost classes in our rules as if they were inner classes of class $\emptyset$. We also sometime use $\overline{cn}\langle\overline{f}\rangle$ to denote $cn_1\langle f_{11..1n_1}\rangle.cn_2\langle f_{21..2n_2}\rangle..cn_k\langle f_{k1..kn_k}\rangle$.

4.2 Soundness of the Type System

Our type checking rules ensure that for a program to be well-typed, the program respects the properties described in Figure 1. A complete syntactic proof [63] of type soundness can be constructed by defining an operational semantics (by extending the operational semantics of Classic Java [31]) and then proving that well-typed programs do not reach an error state and that the generalized subject reduction theorem holds for well-typed programs. The subject reduction theorem states that the semantic interpretation of a term's type is invariant under reduction. The proof is straightforward but tedious, so it is omitted here.

4.3 Type Inference

Although our type system is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information. Instead, we can use a combination of inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. [14, 11] describe an intraprocedural type inference algorithm and some default types; we can use a similar approach. (In [14, 11], about one in thirty lines of code had to be changed to express Java programs in an ownership type system.) We emphasize that this approach to inference is purely intraprocedural and does not infer method signatures or types of instance variables. Rather, it uses a default completion of partial type specifications in those cases to minimize the required annotations. This approach permits separate compilation.

Predicate	Meaning
WFClasses(P)	There are no cycles in the class hierarchy
ClassOnce(P)	No class is declared twice in P
IClassesOnce(P)	No class contains two inner classes with same name, either declared or inherited
FieldsOnce(P)	No class contains two fields with same name, either declared or inherited
MethodsOnce(P)	No class contains two methods with same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden. The read and write effects of an overriding method must be superseded by those of the overridden methods

Figure 12: Predicates Used in Type Checking Rules

4.4 Runtime Overhead

The system we described is a purely static type system. The ownership relations are used only for compile-time type checking and are not preserved at runtime. Consequently, our programs have no runtime overhead compared to regular (Java) programs. In fact, one way to compile and run a program in our system is to convert it into a regular program after type checking, by removing the owner parameters, the constraints on owners, and the effects clauses.

A language like Java, however, is not purely statically-typed. Java allows downcasts that are checked at runtime. Suppose an object with declared type Object⟨o⟩ is downcast to Vector⟨o,e⟩. Since the result of this operation depends on information that is only available at runtime, our type checker cannot verify at compile-time that e is the right owner parameter even if we assume that the object is indeed a Vector. To safely support downcasts, a system has to keep some ownership information at runtime. This is similar to keeping runtime information with parameterized types [54, 61]. [10] describes how to do this efficiently for ownership by keeping runtime information only for objects that can be potentially involved in downcasts into types with multiple parameters.

5 Upgrades in Persistent Object Stores

This section shows how ownership types and effects clauses can be used to enable modular reasoning about the correctness of upgrades in a persistent object store. The desire to achieve such reasoning was the motivation for our work on ownership types for encapsulation.

A persistent object store [46, 5, 9, 17, 18, 29, 56] contains conventional objects similar to what one might find in an object-oriented language such as Java. Applications access persistent objects within atomic transactions, since this is necessary to ensure consistency for the stored objects; transactions allow for concurrent access and they mask failures. Upgrades are needed in such a system to improve object implementations, to correct errors, or even to change interfaces in the face of changing application requirements; this includes incompatible changes to interfaces where the new interface does not support the same methods as the old one. Providing a satisfactory solution for upgrades in persistent object stores has been a long-standing challenge.

An upgrade for a persistent object store is defined as a set of *class-upgrades*, one for each class whose objects need to change. A class-upgrade is a triple: ⟨old-class, new-class, TF⟩. It indicates that all objects belonging to old-class should be transformed, through the use of the *transform function* TF provided by the programmer, into objects of new-class. TF takes an old-class object and a newly allocated new-class object and initializes the new-class object from the old-class object. The upgrade infrastructure causes the new-class object to take over the identity of the old-class object, so that all objects that used to point to the old-class object now point to the new-class object.

An upgrade is executed by transforming all objects whose classes are being replaced. However, transforms must not interfere with application access to the store, and must be performed efficiently in both space and time. In addition, they must be done safely so that important persistent state is not corrupted. Previous approaches [4, 7, 29, 45, 56, 57] do not provide a satisfactory solution to these challenges; they either stop application access to the database while running the upgrade, or they keep copies of the database, or they limit the expressive power of transforms (e.g., transform functions are not allowed to make method calls).

Our system provides an efficient solution. It performs upgrades *lazily*. An object is transformed just before an application accesses it: the application transaction is interrupted to run the transform function. The transform runs in its own transaction; when this transaction commits, the application transaction is resumed. Our system also allows later upgrades to run in parallel with earlier ones. If an object has several pending transforms, they are run one after another, in upgrade order. Furthermore, if a transform transaction T encounters an object with a pending transform from an earlier upgrade, T is interrupted (just like an application transform) to run the pending transform, and continues execution after the pending transform commits.

More details can be found in [13, 12, 47].

5.1 Ownership Types for Safe Upgrades

Our upgrade system is efficient and expressive: it does not delay application transactions, avoids the use of versions (copies of objects), and does not limit the expressive power of transform functions. But it also needs to support modular reasoning about the correctness of transform functions. This is possible if each transform function encounters only object interfaces and invariants that existed when its upgrade started, even though in reality the transform function might run much later, after application transactions and other transform transactions.

We use our variant of ownership types to enable modular reasoning about the correctness of transform functions. Our system checks statically whether transform functions satisfy the following constraint, using ownership and effects declarations (effects clauses state what objects TFs access):

S1. TF(x) only accesses objects that x owns (directly or transitively).

Transform functions often satisfy S1 because ownership frequently captures the *depends* relation discussed in Section 2, and typically transform functions only access the depended-on objects. (We discuss in [13] how we support modular reasoning of transform functions when S1 does not hold.)

Our implementation also ensures the following:

S2. For any object x affected by an upgrade, x is accessed before any object owned by x.

We ensure S2 using two mechanisms. If the owned object is encapsulated within x, the type system guarantees that x is accessed first. If the owned object is shared with an inner class object of x, our system causes x to be accessed just before the inner class object is first used after the upgrade. This latter mechanism is described in more detail in [13].

When S1 holds, we can prove that out-or-order processing of transforms does not cause problems. In particular, we can show that: applications do not interfere with transform functions, transform functions of unrelated objects do not interfere with each other, and transform functions of related objects run in a pre-determined order (namely an object is transformed before its owned subobjects). (The proofs are given in [13]).

Thus when S1 holds, we can ensure that transform functions encounter the expected interfaces and invariants. This supports modular reasoning: each transform function can be reasoned about as extra method of its old class.

6 Related Work

Euclid [41] is one of the first languages that considered the problem of aliasing. [37] stressed the need for better treatment of aliasing in object-oriented programs. Early work on Islands [36] and Balloons [3] focused on *fully encapsulated* objects where all subobjects an object can access are not accessible outside the object. Universes [53] also enforces full encapsulation, except for read-only references. However, full encapsulation significantly limits expressiveness, and is often more than is needed. The work on ESC/Java pointed out that encapsulation is required only for subobjects that the containing object *depends* on [43, 28], but ESC/Java was unable to always enforce encapsulation.

6.1 Ownership Types and Encapsulation

Ownership types provide a statically enforceable way of specifying object encapsulation. They were proposed in [22] and formalized in [21]. These systems enforce strict object encapsulation, but do so by significantly limiting expressiveness. They require that a subtype have the same owners as a supertype. So TStack⟨stackOwner, TOwner⟩ cannot be a subtype of Object⟨stackOwner⟩. Moreover, they do not support iterators.

PRFJ [14], SCJ [11], and JOE [20] extended ownership types to support a natural form of subtyping. To do so without violating encapsulation, JOE introduces the constraint that in every type with multiple owners, the first owner $\preceq$ all other owners. As a result, in JOE, a program can create a pointer from object x to an object owned by o only if ($x \preceq o$). PRFJ and SCJ allow an object to contain pointers to subobjects owned by a different object, but they have effects clauses that prevent a program from following such pointers. The above systems effectively enforce encapsulation for object fields. However, to support constructs like iterators, they allow method local variables to violate encapsulation. Therefore they do not support local reasoning.

AliasJava [2] uses ownership types to aid program understanding. Like other ownership type systems, AliasJava allows programmers to use ownership information to reason about aliasing. AliasJava is also more flexible than other ownership type systems. However, unlike other ownership type systems, AliasJava does not enforce any encapsulation properties. (This is illustrated with an example in [11].)

Ownership types have been extended to inner classes in [19, 2]. However, these systems do not enforce the property stated in Section 3.5, and do not support local reasoning.

Ownership types have also been used to enforce other properties. Parameterized Race-Free Java (PRFJ) [14] uses an ownership based type system to prevent data races in multithreaded programs. Safe Concurrent Java (SCJ) [11] extends this to prevent both data races and deadlocks. These systems can be combined with our approach to enforce object encapsulation as well as prevent data races and deadlocks. [11] sketches a way of doing this.

Recent work [15, 59] combines region types [60, 24, 35] with our type system to statically ensure both object encapsulation and safe region-based memory management.

6.2 Related Type Systems

Linear types [62] and unique pointers [51] can also be used to control object aliasing. Linear types have been used in low level languages to support safe explicit memory deallocation [24] and to track resource usage [26, 27]. Linear types and unique pointers are orthogonal to ownership types, but the two can be used in conjunction to provide more expressive systems. PRFJ [14] is the first system that combines ownership types with conventional unique pointers. Recent work [23] proposes a better approach that allows a program to specify a unique *external* pointer to an object; there can be other internal pointers to the object from its subobjects.

Effects clauses [49] are useful for specifying assumptions that must hold at method boundaries. Effects enable modular checking of programs. PRFJ [14] is the first system to combine effects with ownership types to statically prevent data races. [11] and [20] also combine effects with ownership for preventing deadlocks and for program understanding. This paper uses effects with ownership to enable lazy upgrades.

Data groups [42, 44] can be used to name groups of objects in an effects clause to write modular specifications in the presence of subtyping. Ownership types provide an alternate way of writing modular specifications. Ownership types can also be used to name groups of objects in an effects clause—the name of an owner can be used to name all the objects transitively owned by the owner. Figure 8 presents an example from [44] expressed using ownership types. Data groups are implemented using a theorem prover, and in principle, they can be very flexible. However, *pivot uniqueness* in [44] imposes drastic restrictions on pivot fields. Ownership types do not impose such restrictions; they only require that the owner of an object be unique. In [44], the *owner exclusion* constraint is hard coded. In our system, programmers can specify arbitrary constraints on owners using where clauses; owner exclusion can be used as a default.

Systems such as TVLA [58], PALE [52], and Roles [40] specify the shape of a local object graph in more detail than ownership types. TVLA can verify properties such as when the input to the program is a tree, the output is also a tree. PALE can verify all the data structures that can be expressed as graph types [39]. Roles can verify global properties such as the participation of objects in multiple data structures. Roles also support compositional interprocedural analysis. In contrast to these systems that take exponential time for verification, ownership types provide a lightweight and practical way to constrain aliasing.

7 Conclusions

Object encapsulation enables sound local reasoning about program correctness in object-oriented languages. Ownership types provide a way of specifying and statically enforcing object encapsulation. However, a type system that enforces strict object encapsulation is too constraining: it does not allow efficient implementation of important constructs like iterators.

This paper argues that the right way to solve the problem is to allow objects of classes defined in the same module to have privileged access to each other's representations. We show how to do this for inner classes. Our variant of ownership types allows objects of inner classes to have privileged access to the representations of the corresponding objects of outer classes. This principled violation of encapsulation allows programmers to express constructs like iterators and wrappers using inner classes. Our system also allows wrappers to be used in more global contexts than the objects they wrap. Yet our system supports local reasoning about the correctness of classes, because a class and its inner classes can be reasoned about together as a module.

Thus the paper describes the first ownership type system that is expressive enough to support iterators and wrappers, while also supporting local reasoning. In addition, the paper describes an application of the technique to enable modular reasoning about upgrades in persistent object stores. Ownership types have been used for other purposes as well, such as for preventing data races and deadlocks, and for safe region-based memory management. Since ownership types require little programming overhead, their type checking is fast and scalable, and they provide several benefits, they offer a promising approach to making object-oriented programs more reliable.

Acknowledgments

We thank Daniel Jackson, Viktor Kuncak, Greg Nelson, Martin Rinard, and Alexandru Salcianu for useful discussions and comments on earlier drafts of this paper.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[3] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1997.

[4] M. P. Atkinson, M. A. Dmitriev, C. Hamilton, and T. Printezis. Scalable and recoverable implementation of object evolution for the PJama 1 platform. In *Persistent Object Systems (POS)*, September 2000.

[5] M. P. Atkinson, M. J. Jordan, L. Daynes, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Persistent Object Systems (POS)*, May 1996.

[6] A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, January 2002.

[7] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD International Conference on Management of Data*, May 1987.

[8] B. Bokowski and J. Vitek. Confined types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1999.

[9] C. Boyapati. JPS: A distributed persistent Java system. SM thesis, Massachusetts Institute of Technology, September 1998.

[10] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[12] C. Boyapati, B. Liskov, and L. Shrira. Ownership types and safe lazy upgrades in object-oriented databases. Technical Report TR-858, MIT Laboratory for Computer Science, July 2002.

[13] C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. Submitted for publication, November 2002.

[14] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[15] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. Submitted for publication, November 2002.

[16] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[17] R. Bretl et al. The GemStone data management system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. 1989.

[18] M. J. Carey et al. Shoring up persistent applications. In *ACM SIGMOD International Conference on Management of Data*, May 1994.

[19] D. G. Clarke. Object ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.

[20] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[21] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, June 2001.

[22] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[23] D. G. Clarke and T. Wrigstad. External uniqueness. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2003.

[24] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, January 1999.

[25] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1995.

[26] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[27] R. DeLine and M. Fahndrich. Adoption and focus: Practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[28] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research Report 156, Compaq Systems Research Center, July 1998.

[29] O. Deux et al. The story of O2. In *IEEE Transactions on Knowledge and Data Engineering (TKDE) 2(1)*, March 1990.

[30] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[31] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, January 1998.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[33] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[34] A. Greenhouse and J. Boyland. An object-oriented effects system. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1999.

[35] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[36] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1991.

[37] J. Hogg, D. Lea, A. Wills, and D. de Champeaux. The Geneva convention on the treatment of object aliasing. In *OOPS Messenger 3(2)*, April 1992.

[38] JavaSoft. Inner class specification, February 1997. Available at http://java.sun.com/products/JDK/1.1.

[39] N. Klarlund and M. I. Schwartzbach. Graph types. In *Principles of Programming Languages (POPL)*, January 1993.

[40] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Principles of Programming Languages (POPL)*, January 2002.

[41] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. In *Sigplan Notices, 12(2)*, February 1977.

[42] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[43] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.

[44] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[45] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1990.

[46] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1999.

[47] B. Liskov, C. Moh, S. Richman, L. Shrira, Y. Cheung, and C. Boyapati. Safe lazy software upgrades in object-oriented databases. Technical Report TR-851, MIT Laboratory for Computer Science, June 2002.

[48] B. Liskov, A. Snyder, R. R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. In *Communications of the ACM (CACM) 20(8)*, August 1977.

[49] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[50] O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[51] N. Minsky. Towards alias-free pointers. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1996.

[52] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[53] P. Muller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming.* 1999.

[54] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[55] J. Noble. Iterators and encapsulation. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, June 2000.

[56] Object Design Inc. *ObjectStore Advanced C++ API User Guide Release 5.1*, 1997.

[57] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1987.

[58] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems (TOPLAS) 20(1)*, January 1998.

[59] A. Salcianu, C. Boyapati, W. Beebee, Jr., and M. Rinard. A type system for safe region-based memory management in Real-Time Java. Technical Report TR-869, MIT Laboratory for Computer Science, November 2002.

[60] M. Tofte and J. Talpin. Region-based memory management. In *Information and Computation 132(2)*, February 1997.

[61] M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[62] P. Wadler. Linear types can change the world. In M. Broy and C. Jones, editors, *Programming Concepts and Methods.* 1990.

[63] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. In *Information and Computation 115(1)*, November 1994.