Ownership Types and Safe Lazy Upgrades in Object-Oriented Databases

Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
{chandra,liskov,liuba}@lcs.mit.edu

Abstract

This paper describes a novel mechanism for upgrading objects in an object-oriented database. Unlike earlier systems, our mechanism is expressive, supporting a rich set of upgrades; it is efficient and does not stop application access to run an upgrade; it avoids making copies of the database; yet it provides good semantics. Expressive efficient upgrades can lead to problems for the code that upgrades objects. For example, the code might observe broken invariants or interfaces unknown at the time it was written. The paper shows how to use a variant of ownership types to avoid such problems and enable programmers to reason about the correctness of their upgrades. Our approach to correctness is novel, and is a significant contribution of this paper.

This paper also presents a new ownership type system. Previous ownership type systems only supported a weak encapsulation property. Enforcing object encapsulation, while supporting subtyping and paradigms like iterators, has been an open problem. Our type system provides a satisfactory solution to this open problem. The novel idea is that we handle inner classes specially—our type system allows objects of inner classes to have privileged access to the representations of the corresponding objects of the outer classes. This principled violation of encapsulation allows programmers to express paradigms like iterators, yet they can reason locally about program correctness.

1 Introduction

Object-oriented databases (OODBs) provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. The database acts as an extension of an object-oriented programming language such as Java, allowing programs access to long-lived objects in a manner analogous to how they manipulate ordinary objects whose lifetime is determined by that of the program [37, 24, 13, 43, 12, 5].

The objects stored in an OODB may live a long time and as a result there may be a need to *upgrade* them, that is, change their code and storage representation. An upgrade can improve an object's implementation, to make it run faster or to correct an error; extend the object's interface, e.g., by providing it with additional methods; or even change the interface in an incompatible way, so that the object no longer behaves as it used to, e.g., by removing one of its methods or redefining what a method does. Incompatible upgrades are probably not common but they can be important in the face of changing application requirements. But providing a satisfactory way of upgrading objects in an OODB has been a long-standing challenge.

1.1 Upgrades in an OODB

This paper describes a mechanism for upgrading objects in an OODB. The approach is object-oriented: an upgrade definition describes what to do with each class that is changing, by providing a replacement class and a *transform function* that is used to initialize the new form of the object using the object's current state.

An upgrade is executed by transforming all objects belonging to classes that are being changed. Some systems [4, 43] stop application access to the OODB while the upgrade is performed. But such a stop-the-world approach can make the system unavailable to users for potentially long periods of time. The unavailability may not be a serious issue if the OODB is small, but if it is large (e.g., trillions of objects residing at thousands of servers), the time during which the system is unavailable to applications can be very long.

Our system avoids delaying applications by running the upgrade lazily. An object is transformed just before an application accesses it, and therefore applications that run after the upgrade starts never see non-upgraded objects. In spite of being lazy, our system provides good semantics by ensuring that when a transform function executes, it encounters object interfaces that existed when its upgrade started and states that satisfy its object's invariants. These guarantees make it easy for programmers to write transform functions and to reason about their correctness.

A lazy system might violate the semantics we wish to provide, because the work of doing an upgrade is interleaved with application accesses to stored objects. For example, a delayed transform function of upgrade U might access an object that has been modified by an application transaction that ran *after* upgrade U started, thus violating our semantics. Also, if the transform function of an object x accesses an object y that has already been transformed either within the same upgrade or a later one, y's interface may be different than expected (if the upgrade was incompatible).

Previous systems do not provide a satisfactory solution to these problems. Stop-the-world systems guarantee that applications and later upgrades cannot interfere with transform functions of an upgrade U, but they have difficulty ordering transform functions within the same upgrade. Some systems [45, 6, 36] avoid problems by severely limiting the expressive power of transform functions, not allowing them to make any method calls. Others (e.g., [4]) make the execution of transform functions order-independent by maintaining two copies of the database during the upgrade. The transform functions initialize the new copy of the database using the old copy; when the upgrade is complete, the new copy replaces the old one. Neither of these approaches is desirable. The loss of expressive power means that many upgrades cannot be expressed using the mechanism, and the two-copy approach consumes huge amounts of space.

This paper describes a lazy upgrade mechanism that supports expressive transform functions and avoids database copies. Our approach is efficient, yet it provides good semantics. For many upgrades, our system ensures statically that transform functions run before any objects they access are modified—either by applications or other transform functions. We also outline solutions for the remaining cases. Our approach is based on the observation that in most cases, a transform function of object x only accesses x and subobjects encapsulated within x. Ownership types provide a way of statically enforcing object encapsulation. We use a variant of ownership types to enforce our mechanism.

Our approach to providing good semantics for upgrades is novel, and is a significant contribution of this paper. Our correctness conditions, which apply to both lazy and eager upgrades, are new, and so is our provision of an efficient lazy system that nevertheless satisfies the conditions.

1.2 Ownership Types for Encapsulation

Ownership types [17, 16, 15] were introduced with the goal of statically enforcing object encapsulation. The idea is that an object may *own* other subobjects that are part of its representation and these subobjects should not be accessible outside the object that owns them. Encapsulation is important because it gives programmers the ability to reason locally about program correctness. However, previous ownership type systems are either not expressive enough (they do not support subtyping and paradigms like iterators), or they do not enforce object encapsulation (they enforce weaker restrictions instead). Enforcing object encapsulation, while supporting subtyping and paradigms like iterators, has been an open problem.

This paper describes a new ownership type system that provides a satisfactory solution to this open problem. The novel idea is that we handle inner classes specially—our type system allows objects of inner classes to have privileged access to the representations of the corresponding objects of the outer classes. This principled violation of encapsulation allows programmers to express paradigms like iterators and yet allows them to reason locally about the correctness of their programs. Our type system also improves upon previous ownership type systems by allowing programmers to specify constraints on formal owner parameters and letting them write parameterized methods. We also combine ownership with effects clauses [39].

Our ownership type system for enforcing encapsulation is useful in upgrade systems as well as in regular programs, and is another significant contribution of this paper.

1.3 Outline

The rest of the paper is organized as follows. Section 2 describes how we perform lazy upgrades in an OODB; it also presents the semantics we wish to provide, that help programmers reason about the correctness of upgrades. Section 3 presents our ownership type system that statically enforces object encapsulation. Section 4 shows how ownership types can be used to enforce our upgrade semantics; it also discusses the limitations of the ownership approach and describes mechanisms that can be used in the remaining cases. Section 5 presents related work, and Section 6 concludes.

2 Upgrades in an OODB

This section defines our approach to providing upgrades for an object-oriented database and explains how we execute upgrades lazily. Then it defines the conditions that an upgrade system ought to support, and it explains why these conditions make it easy for programmers to reason about upgrades. The conditions apply to all upgrade systems, both eager and lazy.

We assume the object-oriented database contains conventional objects similar to what one might find in an object-oriented programming language such as Java. Objects can refer to one another and can interact by calling one another's methods. The objects in the database belong to classes that define their representation and methods. Each class implements a type. Types can be arranged in a hierarchy, so that a type can be a subtype of one or more types. A class that implements a type implements all supertypes of that type.

2.1 Defining Upgrades

Our approach to defining upgrades is object-oriented: an upgrade is defined by describing what should happen to the classes that need to be changed. The information for a class that is changing is captured in a *class-upgrade*. Each class-upgrade is a tuple: ⟨old-class, new-class, TF⟩. The meaning of a class-upgrade is that all objects belonging to old-class will be transformed, through use of the *transform function*, TF, 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. At some point after TF returns (e.g., immediately) 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 refer to the old-class object now refer to the new-class object.

The mechanism preserves object state and identity across an upgrade. This preservation is crucial, because the whole point of the database is to preserve object state. When objects are upgraded, their state must survive, albeit in a modified form as needed in the new class. Furthermore, a great deal of object state is captured in the web of object relationships. This information is expressed by having objects refer to other objects. When an object is upgraded it must retain its identity so that all the objects that referred to it prior to the upgrade still refer to it.

An upgrade is a set of one or more class-upgrades. When a class C is upgraded incompatibly, this may affect other classes, including subclasses of C and classes that use C. All affected classes have to be upgraded as well, so that the new system as a whole remains type correct. A *complete upgrade* contains class-upgrades for all classes that need to change due to some class-upgrade already in the upgrade [53, 21, 4, 25]. Completeness is checked using rules analogous to type checking. Our system accepts an upgrade only if it is complete. At this point we say the upgrade is *installed*. Once an upgrade has been installed, it is ready to run. An upgrade is executed by running transform functions on all affected objects, i.e., all objects belonging to the old classes.

2.2 Running Upgrades

We assume applications access objects in the database within atomic transactions, since this is necessary to ensure consistency for the stored objects; transactions allow for concurrent access and they mask failures. An application transaction consists of calls on methods of persistent objects as well as local computation. A transaction terminates by committing or aborting. If the commit succeeds, all changes to database objects become persistent. If instead the transaction aborts, none of its changes affect the database.

One could imagine running an upgrade as a single transaction that ran all the transform functions in some order but, as mentioned in the introduction, this approach is undesirable since it can make the system unavailable to users for a long time. We avoid delaying application transactions by running the upgrade incrementally and lazily. We run each transform function as an individual transaction. These transactions are interleaved with application transactions.

Our system runs as follows. When an application transaction A is about to use an object that is due to be upgraded, we interrupt A and run the transform function at that point. (If transform functions from several upgrades are pending for that object, we run them one after another in upgrade order.) The transform function runs in its own transaction T. This transaction must be serialized *before* A since A uses the transformed object initialized by T. If T requires access to an old version of some object modified by A, we provide this access, taking advantage of the fact that A has not yet committed (and therefore the old version still exists). As soon as T finishes executing we commit it. Then we continue running A *unless* T modified some object that A read, in which case we abort A and rerun it. (Section 4.1 describes how we avoid aborting A in most cases.) While running T we might encounter an object that has pending transforms. If the pending transform function is for T's upgrade or a later upgrade, we do not do anything. Otherwise, we interrupt T (just as we interrupted A) to run the pending transform function.

We implemented this approach in the Thor OODB [37, 7]; the implementation is described in [38].

2.3 Upgrade Semantics

As we mentioned in the introduction, an upgrade system should guarantee that when a transform function runs, it encounters only interfaces that existed at the time its upgrade was installed and states that satisfy its object's invariants. This guarantee means the transform function writer need not be concerned, when reasoning about correctness of upgrades, with object interfaces and object invariants that existed in the past or will exist in the future. Instead, the transform function can be thought of as an extra method of its class: the writer can assume the same invariants and interfaces as are assumed for the other methods.

An upgrade system provides the guarantee if the following conditions hold (the notation $[A1;\ A2]$ means that $A1$ ran before $A2$):

L1. If we have $[A;\ \mathrm{TF}(x)]$, where A is either an application transaction that ran after TF's upgrade was installed, or A is a transform function from a later upgrade, this has the same effect as running $\mathrm{TF}(x)$ before A.

L2. If $\mathrm{TF}(x)$ and $\mathrm{TF}(y)$ are from the same upgrade and $\mathrm{TF}(x)$ (transitively) uses y and we have $[\mathrm{TF}(y);\mathrm{TF}(x)]$, this is equivalent to running the transform functions in the opposite order.

L3. If $\mathrm{TF}(x)$ and $\mathrm{TF}(y)$ are from the same upgrade and $\mathrm{TF}(x)$ does not (transitively) use y and $\mathrm{TF}(y)$ does not (transitively) use x, then $[\mathrm{TF}(x);\ \mathrm{TF}(y)]$ is equivalent to $[\mathrm{TF}(y);\ \mathrm{TF}(x)]$.

L1 states that the behavior of the system is equivalent to running upgrades eagerly, before later application transactions or later upgrades. L2 and L3 define the expected behavior for transforms within a single upgrade. L2 defines an ordering on transform functions within the same upgrade and thus ensures that transform functions encounter expected interfaces; L2 and L3 together ensure that transform functions encounter expected invariants. The three conditions ensure that transform functions encounter the expected interfaces and object invariants because they ensure that upgrades run in upgrade order, application transactions 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 subobjects).

The implementation described above transforms objects before they are used by applications or transform functions from later upgrades, but does not prevent transform functions from observing unexpected interfaces or broken invariants. Thus it does not automatically provide conditions L1-L3. Our approach to providing these conditions is based on the observation that a transform function of object x usually accesses only x and subobjects encapsulated within x. Ownership types provide a way of specifying and statically enforcing object encapsulation; they are the subject of the next section. We show how ownership types can be used to obtain the desired conditions in Section 4.

3 Ownership Types for Encapsulation

The possibility of aliasing between objects constitutes one of the primary challenges in understanding and reasoning about the correctness of object-oriented programs [30]. Unexpected aliasing can lead to broken invariants, mistaken assumptions, security holes, and surprising side effects, all of which may lead to defective software.

Ownership types provide a statically enforceable way of specifying object encapsulation that restricts object aliasing. Encapsulation is important because it gives programmers 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 easy to do 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 [34]:

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

Thus, if a stack of items is implemented using a linked list, the stack only depends on the linked list but not on the items contained in the linked list. This is because, if code outside could manipulate the list, it could invalidate the correctness of the stack implementation. But code outside can safely access the items contained in the stack because the stack 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.

Ownership types [17, 16, 15] were introduced with the goal of statically enforcing object encapsulation.[1] However, previous ownership type systems are either not expressive enough (they do not support subtyping and paradigms like iterators), or they do not enforce object encapsulation (they enforce weaker restrictions instead). Enforcing object encapsulation, while supporting subtyping and paradigms like iterators, has been an open problem.

This section describes a new ownership type system that provides a satisfactory solution to this open problem. Section 3.1 introduces ownership types by presenting our basic type system. Section 3.2 extends the basic system to allow us to express paradigms like iterators. Section 3.3 describes how we combine effects clauses with ownership.

A formal description of our type system is given in the appendix. The appendix also discusses how we can use type inference to make the ownership declarations less onerous to write, and discusses the impact of ownership types on runtime behavior (ownership information is not needed at runtime and thus has no impact on performance except when downcasts are used).

[1] Ownership types have also been used for statically enforcing the absence of data races and deadlocks in multithreaded programs [10, 9], and to aid program understanding [2].

O1. The owner of an object does not change over time.

O2. The ownership relation forms a tree rooted at `world`.

Figure 1: Ownership Properties

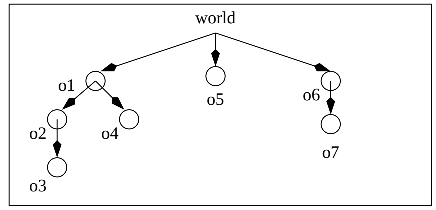

Figure 2: An Ownership Relation

3.1 Basic Type System

This section describes our basic ownership type system. This system is similar to the one described in [15]—the difference is that the type system in [15] allows stack aliases to violate encapsulation, so that it can be expressive enough to support paradigms like iterators. We disallow this kind of violation of encapsulation. We support iterators using the approach described in Section 3.2.

Object Ownership: The key to the type system is the concept of object ownership. Every object in our system has an owner. An object can be owned by another object, or by a special owner called `world`. Our type system statically verifies that a program respects the ownership properties shown in Figure 1. Figure 2 presents an example ownership relation. We draw an arrow from object `x` to object `y` in the figure if object `x` owns object `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`.

The key to local reasoning about program correctness is to have the ownership relation capture the depends relation. If a class is defined so that each object of that class owns every object it depends on, it will be possible to reason locally about correctness of the class.

Parameterizing with Owners: We describe our basic type system using the `TStack` example shown in Figure 3. A `TStack` is a stack of `T` objects. It is implemented using a linked list. This example (and other examples in the paper) use a Java-like language augmented with ownership types.

Every class definition in our system is parameterized with one or more owners.[2] The first owner parameter is special: it owns the `this` object. The other owner parameters are

[2] Our way of parameterizing is similar to the proposals for parametric types for Java [42, 11, 1, 50]. The difference is that our parameters are values and not other types.

```
class TStack<stackOwner, TOwner> {

  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>;
    newNode.init(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> {

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

  void init(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

used to propagate ownership information. In Figure 3, the TStack class is parameterized with stackOwner and TOwner. stackOwner owns the TStack object and TOwner owns the T objects contained in the TStack. Parameterization allows programmers to write generic code, so that different objects of the class can have different owners.

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

Figure 3 contains several illustrations of ownership. The TStack objects own the linked list objects used to store the stack elements. This enables local reasoning about the TStack class. The type of TStack s1 is instantiated using this for both the owner parameters. This means that the TStack s1 is owned by the TStackClient object that created it and so are the T objects in the TStack. 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 is included to indicate that every object is directly or indirectly owned by world.)

For every type $T\langle x_1, ..., x_n\rangle$ with multiple owners, our type

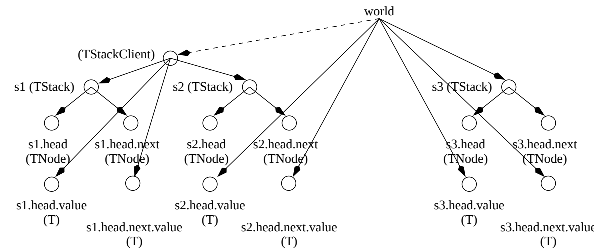

Figure 4: Ownership Relation for TStacks s1, s2, s3

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. This constraint prevents encapsulated objects from being passed on to unencapsulated objects; we prove this property below. Thus, the type of TStack s4 in Figure 3 is illegal because (world $\not\preceq$ this).

Subtyping: The rule for subtyping is that the parameters of the supertype must be instantiated with owners in context and the first owners have to match. The reason the first owners have to match is that the first owners in our system 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.

Object Encapsulation: The property we want to enforce is as follows:

E1. If object z owns object y, but z does not own object x directly or transitively, then x cannot access y.

The underlying reasoning is that y is *inside* the encapsulation boundary of z and x is *outside* the encapsulation boundary, so x must not be able to access y. An object x *accesses* an object y if x contains a pointer to y, or if the methods of x obtain a pointer to y. Consider Figure 2 for an illustration. o1 owns o2. But o1 does not own o5 directly or transitively. So o5 cannot access o2. The only objects that can access o2 are o1 and objects that o1 owns directly or transitively. (In the figure, the only objects that can access o2 are o1, o2, o3, and o4.) Property E1 can thus be restated as follows:

E2. Object x can access object y only if the owner of y is the same as or an ancestor of x.

Note that statements E1 and E2 are equivalent. Property E2 states which object accesses are legal and it rules out precisely the bad accesses that Property E1 rejects. The type system presented in this section guarantees the encapsulation property stated above. Below, we provide an informal proof that our type system provides E2 (and therefore E1):

Proof: Consider the code: *class* $C\langle f_1, ...\rangle\{... \ T\langle o_1, ...\rangle \ y \ ...\}$. Variable y of type $T\langle o_1, ...\rangle$ is declared within the static scope of class C. Owner o_1 can therefore be either 1) this, or 2) world, or 3) a formal parameter of the class, in which case $(f_1 \preceq o_1)$. Note that (this $\prec f_1$). Therefore, we always have (this $\preceq o_1$). The above constraint implies that an object x of a class C can only access an object y if the owner of y is the same as or an ancestor of x. The type system therefore provides Property E2.

The above proof shows that owned objects cannot be accessed from outside their owner. Therefore if ownership captures the depends relation, it will be possible to reason locally about program correctness.

3.2 Extensions to the Basic Type System

The basic type system ensures encapsulation and allows subtyping, but it is not expressive enough to support paradigms like iterators. This is because such paradigms require a violation of ownership: an outside object needs to access subobjects encapsulated in some other object. This section extends our basic type system to make it expressive enough, while preventing arbitrary violations of object encapsulation. We allow the objects of inner classes to have privileged access to the representations of objects of outer classes. This principled violation of encapsulation lets programmers express paradigms like iterators, yet allows them to reason locally about program correctness because the code for inner classes is contained within the code for outer classes. In other words, the entire definition, consisting of the outermost class and its inner classes, is reasoned about as a unit; it is a program module that can be reasoned about locally.

To make the type system more expressive, our extended system also allows programmers to specify constraints on formal owner parameters and lets them write parameterized methods. The typing rules presented in this section are generalizations of the rules in Section 3.1. We summarize the rules in Figure 5. We explain our type system using Figure 6, which contains part of a TStack implementation. The TStack has an iterator, but is otherwise similar to the TStack in Figure 3. The TStack and the iterator are *not* in an ownership relation. If the TStack owned the iterator, it would not be possible for the iterator to be used outside the TStack object. If the iterator owned the TStack it would not be possible to have more than one iterator for a given TStack object.

Inner Classes: Our inner classes are similar to the member inner classes in Java. Inner class definitions are nested inside other classes. An inner class object can only be created by code of an object of the outer class. The resulting object is considered to be inside the object that created it. The outer object might itself be an inner object, nested inside some other outer object. An inner object can access all its outer objects. It can use the syntax C.this to refer to the outer object of class C. (An implementation provides an inner object access to the outer objects via hidden fields. The hidden fields are initialized by the system from hidden arguments to constructors.)

An inner class is parameterized with owners just like a regular class. However, the parameters of an inner class must include all the parameters of all its outer classes. (We could have made the outer class parameters implicitly available to inner classes, but we feel making them explicit enhances program clarity.) As usual, the first owner owns the corresponding object, and (first owner $\preceq$ other owners).

T1. An owner can be instantiated with: an owner parameter, world, this, and C.this, where C is an outer class.

T2. If o is of type $T\langle x_1, ..., x_n\rangle$ then x_1 owns o.

T3. In type $T\langle x_1, ..., x_n\rangle$, $(x_1 \preceq x_i)$ $\forall i \in \{1..n\}$.

T4. For method $m\langle y_1, ..., y_k\rangle$ of an object of type $T\langle x_1, ...\rangle$, $(x_1 \preceq y_i)$ $\forall i \in \{1..k\}$

T5. Class and method instantiations must satisfy the constraints on owner parameters specified in the class and method declarations.

T6. For subtyping, first owners must match.

Figure 5: Rules for Type Checking in our System

Recall from Section 3.1 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 gives an inner object special privileges to access the objects encapsulated within its outer objects.

For example, the TStackEnum class in Figure 6 is an inner class. An object of this class is created by the elements method of TStack. The TStackEnum object has three owner parameters: the one defining its owner, and the two it inherits from its outer object. The TStackEnum constructor accesses a field of the outer TStack object using the syntax TStack.this. Note also that the object referred to by current is owned by TStack.this.

Parameterized Methods: Our type system allows methods to be parameterized to enable the writing of owner-polymorphic code. Recall that for every type with multiple owners, $T\langle x_1, ..., x_n\rangle$, our type system statically enforces the constraint that $(x_1 \preceq x_i)$ for all $i \in \{1..n\}$. For a parameterized method $m\langle y_1, ..., y_k\rangle(...)\{...\}$ of an object of type $T\langle x_1, ..., x_n\rangle$, the restriction is that $(x_1 \preceq y_i)$ for all $i \in \{1..k\}$. This restriction prevents encapsulated objects from being passed on to unencapsulated objects.

Constraints on Owners: Our type system allows classes and methods to specify constraints on owner parameters using where clauses. This is somewhat analogous to the use of where clauses in [20, 42]. To see an illustration, consider the parameterized method elements in Figure 6. The method specifies that (enumOwner $\preceq$ stackOwner) using a where clause. It is therefore legal for the method to instantiate the type TStackEnum⟨enumOwner, stackOwner, TOwner⟩. Without the where clause, the above instantiation would have violated our typing rule that (firstowner $\preceq$ other owners).

Note that in this example we also have (stackOwner $\preceq$ enum-

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

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

Figure 6: TStack With Iterator

Owner) since elements is a method of TStack⟨stackOwner, TOwner⟩ and (first owner of an object $\preceq$ its method's parameters). Therefore enumOwner and stackOwner must be the same. In general, whenever an inner object is accessible to objects not transitively owned by its outer object, the inner object and the outer object will have the same owner.

Object Encapsulation: Property E1 in Section 3.1 stated that if object z owns object y, but z does not own object x directly or transitively, then x cannot access y. Our system enforces the above property except for the case where x is an inner object of y. This is a principled violation of encapsulation—it is useful to implement paradigms like iterators, but it still allows programmers to reason locally about the correctness of their programs. We relax Properties E1 and E2 from Section 3.1 to E1$'$ and E2$'$ as follows, to allow this principled violation:

E1$'$. If object z owns object y, but z does not own object x directly or transitively, then x cannot access y, unless x is an inner object of y.

E2$'$. Object x can access object y only if the owner of y is either 1) the same as or an ancestor of x, or 2) an outer object of x.

Note that statements E1$'$ and E2$'$ are equivalent. Below, we provide an informal proof that our type system provides E2$'$ (and therefore E1$'$):

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

Proof: Consider the code: *class* $C\langle f_1, ...\rangle\{... \; T\langle o_1, ...\rangle \; y \; ...\}$. Variable y of type $T\langle o_1, ...\rangle$ is declared within the static scope of class C. Owner o_1 can therefore be either 1) this, or 2) world, or 3) a formal class parameter, in which case $(f_1 \preceq o_1)$, or 4) a formal method parameter, in which case also $(f_1 \preceq o_1)$, or 5) C'.this, where C' is an outer class. In the first four cases, (this $\preceq o_1$). In the fifth case, (C'.this $\preceq o_1$). This implies that an object x of class C can only access an object y if y is owned by either: 1) x or an ancestor of x, or 2) an outer object of x. The type system therefore provides Property E2$'$.

3.3 Extensions to Support Effects

Effects [39, 10, 9] are orthogonal to ownership and encapsulation, but effects clauses are useful for specifying assumptions that must hold at method boundaries, thus enabling modular reasoning and checking of programs. In this paper, we use effects with ownership types to provide safe lazy upgrades; we describe this in Section 4.

Our system allows programmers to specify *reads* and *writes* clauses. Consider a method that specifies that it writes $(w_1, ..., w_n)$ and reads $(r_1, ..., r_m)$. Then 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 an example that uses effects. The figure contains 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. (A complete TStack example with effects clauses is in the appendix.)

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 all the effects of a method. For example, it is difficult to specify all the read effects in the getNext method of the TEnumeration class because TEnumeration is a supertype of TStackEnum and we cannot name TStack.this in the getNext method of TEnumeration. To accommodate such cases, we allow an escape mechanism, where a method can include world in its effects clauses.

4 Safe Lazy Upgrades

In this section we show how ownership types and effects allow us to enforce L1-L3, the upgrade correctness conditions defined in Section 2. We also discuss the limitations of the ownership approach and outline our solutions for the remaining cases.

4.1 Enforcing Upgrade Semantics

This section shows how we can ensure conditions L1-L3 using ownership types and effects. The conditions were stated in terms of objects that transform functions use. In this section we limit transform functions to using only owned objects:

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

Our type system statically checks and enforces S1 using the ownership and effects declarations. Transform functions will often satisfy S1 because ownership frequently captures the *depends* relation discussed in Section 3, and typically the transform functions will only need to access the depended-on objects. Section 4.2 analyzes the situations where a transform function violates S1 and explains how we deal with such upgrades.

Our ownership type system guarantees that the owner of an object is accessed before the object is accessed (since the owned object is encapsulated within the owner). Our runtime system uses this property to ensure the following conditions:

S2. TF(x) runs before A accesses x or any object x owns either directly or transitively, where A is either an application transaction that ran after TF's upgrade was installed, or A is a transform function from a later upgrade.

S3. If TF(x) and TF(y) are in the same upgrade and x owns y directly or transitively, then TF(x) runs before TF(y).

Now we give informal proofs that when S1 holds, S1-S3 provide the semantics stated in L1-L3. Our proofs consider only adjacent transactions, but this is sufficient because the three conditions together can be used to reorder sequences containing intervening transactions to achieve adjacency.

L1: If we have [A; TF(x)], where A is either an application transaction that ran after TF's upgrade was installed, or A is a transform function from a later upgrade, this has the same effect as running TF(x) before A.

Proof: Since A ran before TF(x), we know from S2 that A does not access x or any object x (transitively) owns. We also know from S1 that TF(x) only accesses x and objects x (transitively) owns. Therefore the read/write sets of A and TF(x) have no object in common and thus the effect is the same as if TF(x) ran before A.

L2: If TF(x) and TF(y) are from the same upgrade and TF(x) (transitively) uses y and we have [TF(y);TF(x)], this is equivalent to running the transform functions in the opposite order.

Proof: Since x (transitively) uses y, we know from S1 that x (transitively) owns y. Therefore, we know from S3 that TF(x) runs before TF(y). Thus the property holds trivially because the order [TF(y); TF(x)] will not occur.

L3: If TF(x) and TF(y) are from the same upgrade and TF(x) does not (transitively) use y and TF(y) does not (transitively) use x, then [TF(x); TF(y)] is equivalent to [TF(y); TF(x)].

Proof: TF(x) and TF(y) can commute unless there is some object z that is read by one TF and modified by the other. If such an object exists, we know from S1 that it must be owned (transitively) by both x and y. Moreover we know that the ownership relation forms a tree (see Figure 1). Therefore the existence of z implies that either x owns y and y owns z, or y owns x and x owns z. But the ownership system guarantees that an owning object is accessed before any object it owns. Therefore whichever object owns the other, the TF for that object must use the other before using z, which violates the assumption that neither TF uses the other object.

When S1 holds we also get another benefit. Recall from Section 2 that our system will abort an interrupted transaction if it previously read an object modified by the transform function. However, when S1 holds, such an abort will never happen. This is because the interrupted transaction cannot use any object that a pending transform function will use without first causing that pending transform function to run.

4.2 Triggers and Versions

There are two situations where a transform function may violate condition S1. The first occurs when the depends relation is not captured by ownership, perhaps because the depends relation does not form a tree. For example, in Figure 6 both TStack and TStackEnum depend on the linked list, but only one of them can own the list (the TStack in our code). The second case occurs when TF(x) reads objects that x does not depend on. Depends (and thus ownership) is intentionally limited to not include immutable properties of subobjects, since correctness of a class does not require encapsulation of the subobjects in this case. However, a transform function may read such subobjects, perhaps because they are no longer immutable after an upgrade.

When S1 is violated there are two possible approaches. The first is to explicitly order the transform functions so that the transform function that violates S1 runs before any access to the objects it uses (but does not own) takes place. The second solution is to use versions. Since the decision about which approach to use requires an understanding of program behavior, we bring violations of S1 to the attention of programmers so that they can decide what to do.

Explicit ordering of transform functions is possible when the object whose transform function violates S1 is owned by a containing object that also owns all objects used by the transform function. For example, in Figure 6 the TStackClient object owns both the TStack object and the TStackEnum object. In this case, we can force the TStackEnum to be transformed before the TStack is used by attaching a *trigger* to the TStackClient class. As mentioned earlier, when an inner object (like an iterator) is used outside its outer object, both inner and outer object will have the same owner. Therefore triggers can be used to order upgrades for iterators (and other inner class objects) unless the owner is world.

A trigger is a function that takes an object as an argument and returns a list of objects needing to be upgraded. Triggers are defined as part of an upgrade in addition to the class-upgrades; such a definition identifies the class being triggered and provides the code for the trigger. The system runs the trigger when an object of the class is first used (after the upgrade is installed); then it processes the list (in list order) and runs any pending transform functions on the objects in the list. A trigger is constrained to only read owned objects; thus it cannot affect the system state. Triggers provide L1-L3. E.g., triggers provide L1 and L2 because $[A;\ \mathrm{TF}(x)]$ and $[\mathrm{TF}(y);\ \mathrm{TF}(x)]$ cannot occur.

When there is no containing object, we have to fall back on versions. In this case, we keep old versions for any unowned object used by the offending transform function $\mathrm{TF}(x)$; for each such object z, we also keep versions for all objects it owns. We restrict the transform function to only read the old versions but not modify them. Versions provide L1-L3 because the immutable versions preserve the old interfaces and object states.

We use effects clauses to enforce constraints on triggers, and also constraints on transform functions with respect to old versions: in both cases we allow reads but not writes. Thus in both these cases, we take advantages of having separate reads and writes clauses.

As mentioned earlier, the problem with versions is that they are expensive. And sometimes they are not needed. For example, the correctness of a transform function for a TStackEnum in Figure 6 will not be affected by a mutation of the corresponding TStack (to push or pop an element), provided the TStack or the underlying list are not transformed incompatibly. We allow users to avoid versions in such cases; more details can be found in [38].

5 Related Work

Section 5.1 discusses related work in the area of upgrade systems. Section 5.2 discusses related type systems.

5.1 Upgrades in an OODB

The two main approaches to upgrading the classes in an OODB are class (or schema) evolution and class versioning. In the evolution approach the database has one logical schema to which modifications of class definitions are applied. Object instances are converted (eagerly or lazily, but once and forever) to conform to the latest schema. In the schema or class versioning approach (e.g. [19, 47, 14]) multiple versions of a schema or class can co-exist. Instances can be represented as if they belong to a specific version of their class, but how this is done (e.g., by creating a separate instance or by keeping one version-specific copy and dynamically converting it as needed) depends on the concrete system.

Our discussion focuses on the schema evolution approach because it is most relevant to our work; the efficiency and correctness issues in the schema versioning approach deal with supporting application access to the same object via multiple potentially incompatible interfaces and are very different from the efficiency and correctness issues in the evolution-based upgrade systems. The schema evolution approach is used in Orion [6], OTGEN [36], O2 [24, 53], GemStone [12, 45], Objectivity/DB [44], Versant [49], and PJama [5, 4] systems, and is the only approach available in any commercial RDBMS.

None of the existing schema evolution systems provides both expressive and efficient upgrades. Furthermore, none bases the correctness of the upgrade system on the property of encapsulation or ownership types. Work in O2 uses static techniques to reason about upgrade completeness [21] but does not consider static techniques for correctness of lazy upgrades or version avoidance.

To insure good semantics, existing schema evolution systems either restrict the expressive power of transforms, or sacrifice efficiency by adopting a stop-the-world (*eager*) approach that is costly in time and space (because of the use of versions). Very few systems support general transforms and lazy conversion. E.g., GemStone and Orion do not support user-defined transform functions that read subobjects (these are called *complex transforms*), ObjectStore supports a limited form of eager conversion and no lazy conversion, Versant supports lazy conversion for default transforms but complex transforms require eager conversion, PJama supports eager conversion for user-defined transforms but has no support for lazy conversion. Dmitriev [25] provides an extensive survey of the existing schema evolution systems and analyzes their limitations.

The work on O2 explores lazy conversion and complex transforms. This work introduces an upgrade system correctness condition [53] based on the equivalence of lazy and eager conversion, and is the first to identify problem posed by deferred complex transforms and incompatible upgrades. The O2 system ensures type safety for deferred complex transforms using a "screening" approach similar to versioning. Unlike our approach, however, analysis in O2 does not take encapsulation into account. Whenever an incompatible upgrade occurs after a complex transform is installed, it either activates an eager conversion, or avoids transform interference by keeping versions for all objects. This approach is unnecessarily conservative (it switches to eager execution even when S1 holds). Also, O2 does not solve the problem of applications modifying objects that are then used by transforms from earlier upgrades; this is unsafe because it violates condition L1.

```
class Main<o> {
  void some_method() {
    final Foo<world> x = ...;

    Object<x> i_should_not_have_got_this_pointer = x.get();
    /* owner is instantiated with a local variable !!! */

  }
}
class Foo<p> {
  Object<this> this_should_not_escape;

  Object<this> get() {
    return this_should_not_escape; // Violates Encapsulation
  }
}
```

Figure 8: Violation of Encapsulation in [15]

5.2 Related Type Systems

Euclid [33] was one of the first languages that considered the problem of aliasing. [30] stressed the need for better treatment of aliasing in object-oriented programs. Early work on Islands [29] and Balloons [3] focused on *fully encapsulated* objects where all subobjects an object could access were not accessible outside the object. 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 [34], but ESC/Java was unable to always enforce encapsulation.

Ownership types provide a statically enforceable way of specifying object encapsulation. [17] is one of the first systems that introduces ownership types. [16] presents a formalization of the type system. In these systems, a subtype must have the same owners as a super type. A subtype cannot have more owners. So $\mathsf{TStack}\langle\mathsf{stackOwner},\mathsf{TOwner}\rangle$ cannot be a subtype of $\mathsf{Object}\langle\mathsf{stackOwner}\rangle$. Moreover, one cannot express paradigms like iterators in these systems.

[15] builds on previous work in [17, 16]. It is the first type system to introduce the rule that in every type with multiple owners, (first owner $\preceq$ other owners). This rule enables the type system to provide a satisfactory solution for subtyping. To support paradigms like iterators, this type system allows an application to have stack aliases that violate encapsulation. As a result, this type system only enforces a weaker property, that all the paths in the heap to an object x must pass through the owner of x. The type system does not enforce encapsulation. Figure 8 presents example code in their system that violates encapsulation.

[10] uses a variant of ownership types to statically prevent data races in multithreaded programs. [9] extends this type system to also prevent deadlocks. These systems allow stack aliases that violate encapsulation. They support subtyping and paradigms like iterators. They do not have the rule that (first owner $\preceq$ other owners). But they have effects clauses [39] that ensure that every thread holds the lock on the owner of an object before the thread accesses the object. This ultimately enables them to enforce a weak encapsulation property, that an application must be able to name the owner of x to be able to access x.

Linear types [51] and unique pointers [40] can also be used to control object aliasing. Linear types have been used to support safe explicit memory deallocation in low level languages [18, 48, 28]. Vault [22, 23] uses an extension of linear types to enforce low level protocols. Linear types and unique pointers are orthogonal to ownership types, but the two can be used in conjunction to provide more expressive type systems. PRFJ [10] is the first system to combine ownership types with unique pointers. This lets programmers express programming idioms that neither ownership types nor unique pointers alone can express. SCJ [9] and AliasJava [2] also combine ownership types with unique pointers. The type system we presented in this paper can be combined with unique pointers in an analogous way.

Our ownership type system is somewhat similar to the type systems in Capability Calculus [18], Alias Types [48], and Cyclone [28] for doing region-based memory management. For example, in Cyclone, classes and methods are parameterized with regions. In our system, classes and methods are parameterized with owners. In Cyclone, methods specify the regions that must be alive at method entry. In our system, methods specify the owners of objects that will be read or written. But these other systems including Cyclone do not deal with objects or subtyping.

Effects [39] are orthogonal to ownership and encapsulation, but having effects in a type system can be useful. In this paper, we use effects with ownership types to provide safe lazy upgrades. PRFJ [10] is the first system to combine effects with ownership types to statically prevent data races in multithreaded programs. SCJ [9] uses effects with ownership types to statically prevent data races and deadlocks. [15] also combines effects with ownership types for program understanding. Recent work on data groups [35] describes a way of grouping objects and naming a group of objects in an effects clause. Ownership types provide another way of grouping objects—the name of an owner can be used to name all the objects owned by the owner. Ownership types thus provide an alternative to data groups for specifying groups of objects in an effects clause.

Systems such as TVLA [46], PALE [41], and Roles [32] specify the shape of a local object graph in more detail than ours. TVLA can verify properties such as when the input to the program is a list (or tree), the output is also a list (or tree). PALE can verify all the data structures that can be expressed as graph types [31]. Roles support compositional interprocedural analysis and verify similar properties. In contrast to these systems that take exponential time for verification, ownership types provide a lightweight and practical way to constrain aliasing.

Our type system is similar to proposals for parametric types for Java [42, 11, 1, 50]. The difference is that our parameters are values and not other types. Moreover, our type system fits naturally in a language with parameterized types.

6 Conclusions

Object-oriented databases provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. But providing a satisfactory way of *upgrading* objects in an OODB has been a long-standing challenge. Our first contribution addresses this challenge:

- We present a novel mechanism for upgrading objects in an OODB; unlike earlier mechanisms, ours is both expressive and efficient.

The mechanism is expressive because it allows transform functions to make method calls. It is efficient because it avoids making copies of the database, and does not stop application access to the database to run an upgrade.

An important issue with any upgrade system is what semantics it should provide. Defining this semantics is our next contribution:

- We define conditions (conditions L1-L3) that an upgrade system should satisfy in order to make it easy for programmers to reason about the correctness of upgrades. These conditions apply to both lazy and eager systems.

Conditions L1-L3 ensure that each transform function encounters invariants and interfaces that existed when its upgrade was installed. This enables programmers to reason about the correctness of their transform functions because they do not have to be concerned with the invariants and interfaces that existed in the past or will exist in the future.

Our system supports L1-L3 when objects are encapsulated. Ownership types were introduced with the goal of statically enforcing object encapsulation. Encapsulation is important because it enables programmers to reason locally about the correctness of their programs. However, previous ownership type systems only provided a weak encapsulation property; enforcing object encapsulation, while providing a satisfactory solution for subtyping and for paradigms like iterators, has been an open problem. Our next contribution is to provide a solution to this problem:

- We define a new ownership type system that statically enforces object encapsulation while supporting subtyping and paradigms like iterators.

Our ownership type system is useful in upgrade systems as well as in regular programs. Our approach makes use of inner classes. Objects of inner classes have privileged access to the representations of the corresponding objects of the outer classes. This principled violation of encapsulation allows programmers to express paradigms like iterators, yet reason locally about program correctness. In addition, we extend previous work on ownership to increase expressiveness:

- Our ownership type system supports generic methods (through the use of method parameters) and constraints on owners.

We also combine ownership with effects.

Ownership types and effects allow us to define a constraint on transform functions that ensures upgrades satisfy L1-L3: this is our next contribution:

- Our system guarantees that when transform functions are constrained to use only owned objects, L1-L3 hold.

However, not all transform functions will satisfy the constraint. We are able to recognize such transform functions statically and bring the problem to the attention of the user. The user can always use versions to solve the problem but versions are expensive. Therefore we provide an alternative, a novel approach for ordering transform function execution:

- We define a new technique (triggers) that allows the order of transform function execution to be controlled. Also, we show that triggers together with versions ensure conditions L1-L3.

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. 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] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] S. M. Clamen. Type evolution and instance adaptation. Technical Report CMU-CS-92-133, Carnegie Mellon University, June 1992.

[15] 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.

[16] 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.

[17] 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.

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

[19] V. M. Crestana-Jensen, A. J. Lee, and E. A. Rundensteiner. Consistent schema version removal: An optimization technique for object-oriented views. In *IEEE Transactions on Knowledge and Data Engineering (TKDE) 12(2)*, March 2000.

[20] 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.

[21] C. Delcourt and R. Zicari. The design of an integrity consistency checker (ICC) for an object-oriented database system. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1991.

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

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

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

[25] M. Dmitriev. Safe class and data evolution in large and long-lived Java applications. Technical Report TR-2001-98, Sun Microsystems, August 2001.

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

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

[28] 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.

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

[30] 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.

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

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

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

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

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

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

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

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

[44] Objectivity Inc. *Objectivity Technical Overview, Version 6.0*, 2001.

[45] 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.

[46] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Programming Languages (POPL)*, January 1999.

[47] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1986.

[48] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, March 2000.

[49] Versant Object Technology. *Versant User Manual*, 1992.

[50] 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.

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

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

[53] R. Zicari. A framework for schema updates in an object-oriented database systems. In *International Conference on Data Engineering (ICDE)*, April 1991.

Appendix

A TStack Example

```
// stackOwner owns the TStack object
// TOwner owns the T objects in the stack.

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

  void push(T<TOwner> value) writes(this) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>;
    newNode.init(value, head);
    head = newNode;
  }
  T<TOwner> pop() writes(this) {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
  TEnumeration<enumOwner, TOwner> elements<enumOwner>()
    writes(enumOwner) where (enumOwner <= stackOwner) {
    return new TStackEnum<enumOwner, stackOwner, TOwner>;
  }
  class TStackEnum<enumOwner, stackOwner, TOwner>
    implements TEnumeration<enumOwner, TOwner> {

    TNode<TStack.this, TOwner> current;

    TStackEnum() writes(enumOwner){
      current = TStack.this.head;
    }
    T<TOwner> getNext() writes(this) reads(TStack.this) {
      if (current == null) return null;
      T<TOwner> t = current.value();
      current = current.next();
      return t;
    }
    boolean hasMoreElements() reads(this) {
      return (current != null);
    }
  }
}

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

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

class T<TOwner> { }

class TStackClient<clientOwner> {
  void test() writes(this) reads(world) {
    T<this>               t = new T<this>;
    TStack<this, this>    s = new TStack<this, this>;
    s.push(t);
    TEnumerator<this,this> e = s.elements();
    t = e.getNext();
  }
}

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

B Ownership 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 [27] known as Classic Java [26]. 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.

B.1 Type Checking

Figure 9 presents our grammar. We present a number of predicates below that we use in the type system. These predicates are based on similar predicates from [26].

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 the same name
FieldsOnce(P)	No class contains two fields with the same name, either declared or inherited
MethodsOnce(P)	No class contains two methods with the 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

The core of our type system is a set of rules for reasoning about the typing judgment: $P; E; R; W \vdash e : t$. 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 $E ::= \emptyset \mid E, t\ x \mid E, \text{owner } f \mid E, (o_1 \preceq o_2)$. We define effects as $R, W ::= o_{1..n}$. We define the type system using the following judgments. We present the typing rules for these judgments in Appendix C.

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P; E \vdash \mathit{defn}$	defn is a well-formed class definition
$P; E \vdash \mathit{wf}$	typing environment E is well-formed
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P; E \vdash \mathit{field} \in c$	class c declares/inherits field
$P; E \vdash \mathit{meth} \in c$	class c declares/inherits meth
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P; E \vdash_{\text{owner}} o$	o is an owner
$P; E \vdash o_1 \preceq o_2$	o_1 is the same as or descendant of o_2
$P; E \vdash X \preceq Y$	effect X is subsumed by effect Y
$P; E \vdash e : t$	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

B.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 [52] of type soundness can be constructed by defining an operational semantics (by extending the operational semantics of Classic Java [26]) 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.

P	$::=$	$\mathit{defn}^*\ e$
defn	$::=$	class $cn\langle \mathit{formal}+\rangle$ extends c constr body
body	$::=$	$\{\mathit{iclass}^*\ \mathit{field}^*\ \mathit{meth}^*\}$
c	$::=$	$cn\langle \mathit{owner}+\rangle$ | Object$\langle \mathit{owner}+\rangle$
owner	$::=$	formal | world | cn.this
constr	$::=$	where $(\mathit{owner} <= \mathit{owner})^*$
iclass	$::=$	defn
meth	$::=$	$t\ mn\langle \mathit{formal}^*\rangle(\mathit{arg}^*)\ \mathit{effects}\ \mathit{constr}\ (\mathit{owner}^*)\ \{e\}$
$\mathit{effects}$	$::=$	reads (owner^*) writes (owner^*)
field	$::=$	$t\ \mathit{fd}$
arg	$::=$	$t\ x$
t	$::=$	c | int
formal	$::=$	f
e	$::=$	new c | let $(\mathit{arg}{=}e)$ in $\{e\}$ | x | $x = e$ | $e;e$ | $x.\mathit{fd}$ | $x.\mathit{fd} = y$ | $x.mn\langle \mathit{owner}^*\rangle(y^*)$
cn	$\in$	class names
fd	$\in$	field names
mn	$\in$	method names
x, y	$\in$	variable names
f	$\in$	owner names

Figure 9: Grammar

B.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. [10, 9] describes an intraprocedural type inference algorithm and some default types; we can use a similar approach. 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.

B.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 type parameters and 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 [42, 50]. [8] 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.

C Rules for Type Checking

$\boxed{\vdash P : t}$

[PROG]

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

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

[CLASS]

$$\frac{\begin{array}{c} E = E_1,\ \mathsf{owner}\ g,\ E_2 \Longrightarrow \exists_i\ g = f_i \\ E' = E,\ \mathsf{owner}\ f_{1..n},\ (o \preceq o')*,\ cn\langle f_{1..n}\rangle\ cn.\mathsf{this} \quad P;\ E' \vdash wf \\ P;\ E' \vdash c \quad P;\ E' \vdash \mathit{iclass}_i \quad P;\ E' \vdash \mathit{field}_i \quad P;\ E' \vdash \mathit{meth}_i \end{array}}{P;\ E \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ (o \preceq o')*\ \{\mathit{iclass}*\ \mathit{field}*\ \mathit{meth}*\}}$$

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

[ENV $\emptyset$]

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

[ENV X]

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

[ENV OWNER]

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

[ENV $\preceq$]

$$\frac{\begin{array}{c} P;\ E \vdash_{\mathrm{owner}} o \\ P;\ E \vdash_{\mathrm{owner}} o' \end{array}}{P;\ E,\ o \preceq o' \vdash wf}$$

[TYPE C]

$$\frac{\begin{array}{c} P;\ E \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle ...\ \mathsf{where}\ (g \preceq g')* ... \\ P;\ E \vdash_{\mathrm{owner}} f_i \Longrightarrow o_i = f_i \\ P;\ E \vdash_{\mathrm{owner}} o_i \quad P;\ E \vdash o_1 \preceq o_i \\ P;\ E \vdash (g_i[o_1/f_1]..[o_n/f_n] \preceq g'_i[o_1/f_1]..[o_n/f_n]) \end{array}}{P;\ E \vdash cn\langle o_{1..n}\rangle}$$

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

[TYPE INT]

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

[TYPE OBJECT]

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

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

[SUBTYPE REFL]

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

[SUBTYPE TRANS]

$$\frac{P;\ E \vdash t_1 <: t_2 \quad P;\ E \vdash t_2 <: t_3}{P;\ E \vdash t_1 <: t_3}$$

[SUBTYPE CLASS]

$$\frac{\begin{array}{c} P;\ E \vdash cn_1\langle o_{1..n}\rangle \\ P;\ E \vdash \mathsf{class}\ cn_1\langle f_{1..n}\rangle\ \mathsf{extends}\ cn_2\langle f_1\ o*\rangle\ ... \end{array}}{P;\ E \vdash cn_1\langle o_{1..n}\rangle <: cn_2\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

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

[METHOD DECLARED]

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

[METHOD INHERITED]

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

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

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ \mathsf{owner}\ f_{1..n},\ (o \preceq o')*,\ arg_{1..a} \\ P;\ E' \vdash wf \quad P;\ E';\ r_{1..r}, w_{1..w};\ w_{1..w} \vdash e : t \end{array}}{\begin{array}{c} P;\ E \vdash t\ mn\langle f_{1..n}\rangle(arg_{1..a})\ \mathsf{reads}(r_{1..r}) \\ \mathsf{writes}(w_{1..w})\ \mathsf{where}(o \preceq o')*\ \{e\} \end{array}}$$

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

[FIELD DECLARED]

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

[FIELD INHERITED]

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

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

[$X \preceq Y$]

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

$\boxed{P;\ E \vdash o \preceq o'}$

[$\preceq$ WORLD]

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

[$\preceq$ ENV]

$$\frac{E = E_1,\ (o \preceq o'),\ E_2}{P;\ E \vdash (o \preceq o')}$$

[$\preceq$ 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]

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

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

[OWNER WORLD]

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

[OWNER FORMAL]

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

[OWNER THIS]

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

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

[EXP TYPE]

$$\frac{\exists_{R,W}\ P;\ E;\ R;\ W \vdash e : t}{P;\ E \vdash e : t}$$

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

[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 VAR]

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

[EXP VAR ASSIGN]

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

[EXP NEW]

$$\frac{P;\ E \vdash c}{P;\ E;\ R;\ W \vdash \mathsf{new}\ c : c}$$

[EXP LET]

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

[EXP REF]

$$\frac{\begin{array}{c} P;\ E;\ R;\ W \vdash x : cn\langle o_{1..n}\rangle \quad P;\ E \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ R = R_1,\ r,\ R_2 \quad x \preceq r \end{array}}{P;\ E;\ R;\ W \vdash x.fd : t[o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{\begin{array}{c} P;\ E;\ R;\ W \vdash x : cn\langle o_{1..n}\rangle \quad P;\ E \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ W = W_1,\ w,\ W_2 \quad x \preceq w \quad P;\ E;\ R;\ W \vdash y : t[o_1/f_1]..[o_n/f_n] \end{array}}{P;\ E;\ R;\ W \vdash x.fd = y : t[o_1/f_1]..[o_n/f_n]}$$

[EXP SEQ]

$$\frac{\begin{array}{c} P;\ E;\ R;\ W \vdash e_1 : t_1 \\ P;\ E;\ R;\ W \vdash e_2 : t_2 \end{array}}{P;\ E;\ R;\ W \vdash e_1;\ e_2 : t_2}$$

[EXP INVOKE]

$$\frac{\begin{array}{ll} \multicolumn{2}{l}{P;\ E \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ {}^{j \in 1..k})\ \mathsf{reads}(r_{1..r})\ \mathsf{writes}(w_{1..w})\ \mathsf{where}(g \preceq g')*\ \{e\}) \in cn\langle f_{1..n}\rangle} \\ P;\ E;\ R;\ W \vdash x : cn\langle o_{1..n}\rangle & P;\ E;\ R;\ W \vdash x_j : t_j[o_1/f_1]..[o_m/f_m] \\ P;\ E \vdash_{\mathrm{owner}} o_i \quad P;\ E \vdash o_1 \preceq o_i & P;\ E \vdash r_{1..r}[o_1/f_1]..[o_m/f_m] \preceq R \\ P;\ E \vdash (g_i[o_1/f_1]..[o_m/f_m] \preceq g'_i[o_1/f_1]..[o_m/f_m]) & P;\ E \vdash w_{1..w}[o_1/f_1]..[o_m/f_m] \preceq W \end{array}}{P;\ E;\ R;\ W \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t[o_1/f_1]..[o_m/f_m]}$$