Ownership Type Systems and Dependent Classes

Werner Dietl
ETH Zurich
Werner.Dietl@inf.ethz.ch

Peter Müller
Microsoft Research
mueller@microsoft.com

Abstract

Ownership type systems structure the heap and enforce restrictions on the behavior of a program. Benefits of ownership type systems include simplified program verification, absence of race conditions and deadlocks, and enforcement of architectural styles.

Dependent classes are a generalization of virtual class systems that allows one class to depend on multiple objects. Dependency is expressed by explicit declaration of the depended-upon objects as class parameters. This allows one to declare dependencies independently of the nesting of classes, which increases the expressive power and reduces the coupling between classes.

In this paper, we describe how ownership type systems can be expressed on top of dependent classes. The ownership type systems are split into two parts: (1) ensuring the topological structure of the heap and (2) enforcing restrictions on the behavior of the program. We present an encoding of the topological part in dependent classes and describe how to enforce the restrictions of ownership type systems directly on the dependent classes program. Finally, we present the encoding of some examples for the MVC interpreter and discuss future work.

Keywords Ownership type systems, Universes, ownership domains, dependent classes, virtual classes

1. Introduction

Ownership type systems structure the heap and enforce restrictions on the behavior of a program. Benefits of ownership type systems include simplified program verification [31, 29, 32], improved specification of effects [10, 12], absence of race conditions and deadlocks [6, 7, 5], easier memory management [8, 3], and enforcement of architectural styles [1] and representation independence [4] of a program.

The problems of aliasing in object-oriented programs have long been recognized [24]. Ownership types [35, 15, 37] and their descendants (e.g., [10, 5]) allow classes to be parameterized by ownership information. Each class declares one owned domain (also called "context") and in addition can be parameterized by enclosing domains. Ownership types enforce the owner-as-dominator discipline, that is, every reference chain from an un-owned object to an object *o* includes *o*'s owner. Ownership domains [2] generalize ownership types by separating the aliasing policy from the mechanism to enforce structure.

FOOL '08 13 January, San Francisco, California, USA.
ACM .

Universe types [31, 18, 17] are a lightweight, non-parametric ownership system that is used for program verification [31, 29]. A type checker and runtime support for Universe types is implemented in the Java Modeling Language (JML) [28].

Virtual classes [30, 19, 20] express dependencies between objects. Similar to virtual methods, a class `A` can declare a dependent class `B` by nesting class `B` within the definition of `A`. Virtual classes can be overridden in subclasses, and the runtime type of an object determines the concrete definition of a virtual class. Recent work [36, 21, 13, 25, 38] formalized and extended virtual classes. All of these systems have in common that dependency is expressed by nesting of classes.

Dependent classes [23] are a generalization of virtual class systems that allows one class to depend on multiple objects. Dependency is expressed by explicit declaration of the depended-upon objects as class parameters. This allows one to declare dependencies independently of the nesting of classes, which increases the expressive power and reduces the coupling between classes.

In our recent work to extend the expressive power of Universe types [40], we became interested in virtual class systems. Expressing dependencies only by nesting of classes is too restrictive to express ownership relations, as the class of the owned objects in general cannot be nested within the owning class. In this paper, we explore the possibility of expressing ownership properties on top of dependent classes [23].

The remainder of this paper is structured as follows. Section 2 gives a brief overview of ownership domains and Universe types, and Section 3 explains the basics of dependent classes. Then, in Section 4, we discuss how the ownership topology can be encoded on top of dependent classes. Sections 5 and 6 give details about how Universe types and ownership domains can be encoded. Section 7 presents examples using the MVC interpreter for dependent classes. Section 8 discusses related and future work, and Section 9 concludes. Finally, Appendix A gives the complete source code for the examples.

2. Overview of Ownership Type Systems

In this section, we give a brief overview of ownership domains and Universe types. Only the main ideas of the two type systems are presented and discussed by examples.

We use uniform terminology for both type systems. A class declares one or more *domains* and can be parameterized by domain parameters. Reference types consist of one owning domain and in addition can be parameterized by further domain arguments. At runtime, an object is *owned* by a *domain* and a domain is *owned* by at most one object. If a class only declares a single domain, then we say that objects in that single domain are owned by the instance of the declaring class and usually do not mention the domain.

2.1 Ownership Domains

Ownership domains [1, 2] are an extension to ownership types [15] in three regards: (1) they allow the declaration of an arbitrary number of domains per class, (2) domains can be public or private, and (3) the encapsulation strategy can be declared by links between domains and assumptions on the domain parameters.

Ownership domains include a uniqueness type system in addition to the ownership system. For the rest of this paper, we will focus only on the ownership part.

We explain the basic concepts of ownership domains (and thereby also of ownership types) by an example: a map from keys to values. We discuss the interesting parts of the example here and provide the complete source code in Appendix A.3.

Class declarations can be parameterized by domains:

```
1 class Map< keyD, valueD > {
2   domain nodes;
3   public domain iters;
4 
5   link iters -> nodes;
```

In line 1 above, class `Map` is parameterized by two domains: the domain `keyD` for the domain of the key objects and the domain `valueD` for the domain of the value objects. Every class implicitly has access to the `domain owner`, the owning domain. By default, every class declares a domain `owned` for the representation of the class. The implicit domain `shared` can be used for globally shared objects.

Lines 2 and 3 declare a private domain `nodes` and a public domain `iters`. Objects in a public domain can be accessed by all objects that can access the owner of that domain.

By default, ownership domains enforce the owner-as-dominator discipline. Additional links between domains can be declared using `links`. Line 5 above allows objects in the `iters` domain to access objects in the `nodes` domain. Note that this property is not transitive. The ownership domains rules ensure that the declaration of links still maintains the encapsulation of the objects.

Types consist of an owning domain name, a class name, and a possibly empty list of domain arguments. For example,

```
7   nodes Node<keyD, valueD> first;
```

declares that the field `first` references an object in the `nodes` domain and passes the domains `keyD` and `valueD` as domain arguments to class `Node`.

Finally, let us look at a use of class `Map`:

```
64 class Client {
65   domain mydata;
66 
67   void main(shared Data value) {
68     final mydata Map<mydata, shared> map;
69     map = new Map<mydata, shared>();
70     map.put(new ID(), value);
71     map.iters Iter<mydata, shared> iter;
72     iter = map.iterator();
73     mydata ID id = (ID) iter.getKey();
74   }
75 }
```

Class `Client` declares a domain `mydata`. Method `main` takes a shared parameter `value`. In line 68, the final local variable `map` is declared. The object referenced by `map` is in the `mydata` domain. The keys of the map are also in the `mydata` domain, whereas the values of the map are shared.

Line 71 declares local variable `iter`. Note that the declared owner `map.iters` expresses that the iterator is in the `iters` domain of the `map` object. This path-type is expressible, because the local variable `map` is final.

2.2 Universe Types

The Universe type system [18, 31] is an ownership type system that enforces the owner-as-modifier discipline, which ensures that the owner of an object controls all modifications of an owned object. Only references to objects in the same domain and to owned objects can be used for modifications. This discipline enables the modular verification of invariants [32].

Statically, the Universe type system uses three different *ownership modifiers* to build this ownership structure. The modifier `peer` expresses that the current object `this` is in the same domain as the referenced object, the modifier `rep` expresses that the current object is the owner of the referenced object, and the modifier `any` does not give any static information about the relationship of the two objects. References with an `any` modifier convey less information as references with a `peer` or `rep` modifier with the same class and are therefore supertypes of the two more specific types.

The owner-as-modifier discipline is enforced by forbidding field updates and non-pure method calls through `any` references. An `any` reference can still be used for field accesses and to call pure methods. The method modifier `pure` is used to mark methods that leave objects in the pre-state of a method call unchanged.

Generic Universe Types [17] extend the Universe type system to type genericity as found in Java 5. We again use the map example to illustrate Generic Universe Types. The complete example can be found in Appendix A.5.

```
class Map< K, V > {
  rep Node<K, V> first;
```

Class `Map` has two type variables `K` and `V`. Type variables parameterize over the concrete class and over ownership at the same time. Field `first` is annotated as `rep`, that is, it references objects that form a part of the representation of the current object.

```
class Client {
  void main(any Data value) {
    rep Map<rep ID, any Data> map =
        new rep Map<rep ID, any Data>();
    map.put(new rep ID(), value);
    rep Iter<rep ID, any Data> iter;
    iter = map.iterator();
    rep ID id = iter.getKey();
  }
}
```

The type of local variable `map` expresses that the `Map` object is part of the representation of the `Client` object, that the keys are `rep ID` objects, and that the values are in an arbitrary domain. In this example, the iterators and the map are in the same domain and therefore local variable `iter` is also annotated as `rep`.

One can think of Universe types as declaring one domain `rep` that corresponds to the `owned` domain of ownership domains. The `peer` modifier corresponds to the `owner` domain of ownership domains, as both express that the referenced object is in the same domain as the current object. Ownership domains as described in [2] cannot express references to arbitrary domains (`any` references). An extension is possible [27], but not considered here.

3. Overview of Dependent Classes

Dependent classes [23] generalize virtual classes. Instead of expressing dependencies between classes by nesting the classes within each other, the dependent class is parameterized by depended-upon objects. In contrast to nesting, parameterization by more than one object is possible. The parametric style decouples the involved classes and can express more dependencies.

Class declarations are parameterized by immutable field declarations and the class is relative to the parameters. Types then take arguments for these fields; the arguments can be paths or classes and determine how restrictive the type is. Let us discuss these concepts by an example taken from [23].

```
abstract class Space {
  abstract Point(s: this) getOrigin();
}

abstract class Point(Space s) {
  abstract Point(s: this.s)
    add( Vector(s: this.s) v );
}
```

Class `Space` is the abstract base class for different spaces, e.g. two- or three-dimensional spaces. It provides a method that returns the point of origin for the space. The return-type of the method is relative. The space `s` of the returned point is the current `this` object. This expresses that the point of origin of the space is in the same space.

Class `Point` is parameterized by a `Space s`. Method `add` takes a vector that is parameterized by the same space as the current point instance. The type `Vector(s: this.s)` expresses that the `s` parameter of class `Vector` has `this.s` as its argument. Similarly, the returned value is a point that is in the same space. These declarations ensure that points and vectors from different spaces do not get mixed.

Dependent classes also allow classes to be specialized depending on the parameters. For example, specialized implementations of class `Point` can provide different implementations depending on the type of space that is provided.

```
abstract class Point(2DSpace s) { ... }
abstract class Point(3DSpace s) { ... }
```

A class can also depend on the multiple parameters that are provided, for example, a `Union` class can depend on the two `Shape` objects that are its components. In class declarations, the fields are implicitly matched by name with corresponding field declarations in superclasses.

For the remainder of this paper, we assume that dependent classes can be extended by the usual Java language constructs, especially the null value, type genericity, static fields, and in addition to the final fields on which the class depends on, we allow other fields that are not part of the type.

4. Encoding Topology

The basic idea for building ownership type systems on top of dependent classes is to separate ownership type systems into two parts: (1) the *topological system* that enforces a certain heap structure and (2) the *encapsulation system* that enforces a certain program behavior. Existing ownership type systems usually combine these two parts. Only ownership domains [2] separate the aliasing policy from the mechanism to enforce structure. But the structural part still enforces certain aliasing policies, for example, that references to arbitrary objects (similar to any-references in Universe types) are always forbidden. In the following, we describe how dependent classes can be used as a very flexible topological system.

The ownership type systems that we discuss in this paper all enforce the same basic topology: every object is owned by at most one other object, and ownership builds a tree that is rooted in one root domain.

Ownership domains allow distinct sets of objects that share the same owner, by adding the additional level of domains. Each object is owned by one domain and each domain is owned by at most one object. Only the root domain does not have an owning object.

The encoding in dependent classes makes the relation to the owning object explicit, by adding a field `owner` on which reference types depend. This field is used to enforce the tree structure of the ownership graph. There are two questions to answer:

1. In which class should the `owner` field be declared?
2. What type should the `owner` field have?

In which class should the **`owner`** ***field be declared?*** All objects in the system should support ownership, therefore the field `owner` should be added to the root class of the class hierarchy, that is, `java.lang.Object` or its equivalent.

An alternative is to add a new class, for instance, `OwnedObject`, which acts as the root class for all objects that can be owned. The system then has to ensure that all classes that should be ownable are subclasses of `OwnedObject`.

What type should the **`owner`** ***field have?*** The important issue is whether we create a recursion between the declaring class and the type of the `owner` field. There are two possibilities: (1) the `owner` field has the same type as the declaring class or (2) we use a different class as type for the `owner` field.

- The `owner` field has the same type as the declaring class. If the `owner` field is declared in the root class `Object`, then the type for `owner` again has to be `Object` or a subclass thereof. If the `owner` field is declared in a subclass `OwnedObject`, we can choose to use `OwnedObject` as type for `owner`.

  This nicely expresses that each object depends on at most one other object as its owner and that the owning object can be owned again. However, this introduces a recursion into our class declaration. For example, let us assume

  ```
  class OwnedObject( owner: OwnedObject ) {}
  ```

  The type of owner is again `OwnedObject`, but that type is incomplete, as no argument for its `owner` field is given. The dependent classes type system would need to build an infinite type and therefore forbids such a recursion.

  This also forbids us from adding the `owner` field directly to the root of the class hierarchy. As this would be a more natural model of ownership, it would be interesting to extend dependent classes to allow this recursion in the future.

- We use a different class as type for the `owner` field. For example, we could declare

  ```
  class OwnedObject( owner: Object ) {}
  ```

  There is no recursion between the class that is being declared and the depended-upon object. This would require that the programmer checks whether the owner of an object is again an owned object. This ensures that the programmer never goes outside of the ownership hierarchy. On the other hand this could also produce several different ownership trees, rooted at different objects.

After answering these two questions, we present an encoding of Universe types and ownership domains in the next two sections.

5. Encoding Universe Types

In recent work [26, 16], we separated the Universe type system into a topological system and an encapsulation system. We are currently working on a version of Generic Universe Types [17], which also separates the topological system and the encapsulation system. Such a separation simplifies the type system and the soundness proof. In this section, we show how the topology of Generic Universe Types can be encoded in dependent classes and discuss how the encapsulation properties can be checked.

5.1 Topology for Ownership Trees

To express the ownership tree structure that is found in ownership types and Universe types, we define:

```
class Object {...}
class OwnedObject( owner: Object ) {
  // introduce a global name for the root domain
  static final Object root = null;
  ...
}
```

The declaration of field `root` allows us to use a symbolic name for objects in the root domain.

We can express the topology of the following simple program that uses Universe types:

```
class C {...}
class D {
  rep C f = new rep C();
}
```

The field `f` may reference only `C` objects that have the current `D` object as owner. We can express this in the dependent classes encoding as:

```
class C( Object owner ) extends OwnedObject {...}
class D( Object owner ) extends OwnedObject {
  C(owner: this) f = new C(owner=this);
}
```

In the above program, we make explicit that the owner of the referenced `C` object is the current `this` object.

5.2 Translation

To translate a program that uses Universe types into a program that uses dependent classes, we perform the following transformation:

- Each class declaration adds the dependency on the `owner` field and extends class `OwnedObject` if no superclass is specified.
- A `peer D` type is encoded as `D(owner: this.owner)` which expresses that the owner of the referenced object is the same as the owner of the current object.
- A `rep D` type is encoded as `D(owner: this)` which expresses that the owner of the referenced object is the current object.
- An `any D` type is encoded as `D(owner: Object)` which does not give a specific owner object and can therefore reference objects with arbitrary owners. In the informal syntax of dependent classes, we can just omit the `owner` argument and write simply `D` for an object with arbitrary owner.
- The purity of methods is preserved.

In the resulting dependent classes program, we can enforce the owner-as-modifier discipline that is enforced by Universe types by checking that for field updates and non-pure method calls:

- the receiver is either `this` or
- the owner of the receiver type is `this` or `this.owner`.

A separate task is to check whether the purity annotations are used correctly. A conservative checker forbids all object creations, field updates and non-pure method calls within a pure method [31]. More relaxed purity checks, that allow object creations [18] and modifications of newly created objects [39], are possible.

There is a close relation between the path substitution found in dependent classes and the viewpoint adaptation of Universe types. Both adapt a relative type to a new environment and decide whether the resulting type lost information.

Consider this simple example program that uses Universe types:

```
class C {
  peer C f;
  rep C g;
  any C h;

  void m() {
    this.f = new peer C();
    this.g = new rep C();
    g.f = new rep C();
    g.g = new rep C(); // error
    h.h = new rep C(); // error
  }
}
```

This code is translated into the following program that uses dependent classes:

```
1  class OwnedObject( Object owner ) { ... }
2  class C( Object owner ) extends OwnedObject {
3    C( owner: this.owner ) f;
4    C( owner: this ) g;
5    C( owner: Object ) h;
6
7    void m() {
8      this.f = new C( owner = this.owner );
9      this.g = new C( owner = this );
10     g.f = new C( owner = this );
11     g.g = new C( owner = this ); // error
12     h.h = new C( owner = this ); // error
13   }
14 }
```

The type of `g.f` is `C(owner: this)` and therefore the assignment on line 10 is valid. On line 11 the type of `g.g` is `C(owner: Object)`, but this type may not be used for an update, as information about the concrete type was lost. Finally, on line 12, the type of `h.h` is `C(owner: Object)` and no information was lost, so dependent classes would allow this update. But the type of the receiver `h`, `C(owner: Object)`, has neither `this` nor `this.owner` as owner. Therefore the additional checks for the owner-as-modifier discipline prevent this update.

5.3 Map Example

The map example is encoded as follows. Note that we implicitly use `OwnedObject` as superclass of all classes.

```
class Map< K, V >( Object owner ) {
  Node<K, V>(owner: this) first;

  void put(K key, V value) {
    Node<K, V>(owner: this) newfirst;
    newfirst = new Node<K, V>(owner = this);
    newfirst.init(key, value, first);
    first = newfirst;
  }

  V get(K key) {
    Iter<K, V>(owner: this.owner) i = iterator();
    while (i.hasNext()) {...}
    return null;
  }

  Iter<K, V>(owner: this.owner) iterator() {
    IterImpl<K, V>(owner: this.owner,
                   map: this) res;
    res = new IterImpl<K, V>(owner=this.owner,
                             map=this);
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V >( Object owner ) {
  K key; V value;
  Node<K, V>(owner: this.owner) next;
  void init(K k, V v,
           Node<K, V>(owner: this.owner) n) {
    key = k; value = v; next = n;
  }
}

interface Iter< K, V >( Object owner ) {
    K getKey();
    V getValue();
    boolean hasNext();
    void next();
}

class IterImpl< K, V >( Object owner,
                           Map<K, V>(owner: Object) map )
implements Iter< K, V > {
  Node<K, V>(owner: this.map) current;

  void setCurrent(Node<K, V>(owner: this.map) c)
    { current = c; }
  K getKey() { return current.key; }
  V getValue() { return current.value; }
  boolean hasNext() { return current != null; }
  next() { current = current.next; }
}

class ID( Object owner ) { ... }
class Data( Object owner ) { ... }

class Client( Object owner ) {
  void main(Data(owner: root) value) {
    Map<ID(owner: this), Data(owner: root)>
       (owner: this) map;
    map = new Map<ID(owner: this),
                  Data(owner: root)>(owner=this);
    map.put(new ID(owner = this), value);
    Iter<ID(owner: this),
         Data(owner: root)>(owner: this) iter;
    iter = map.iterator();
    ID(owner: this) id = iter.getKey();
  }
}
```

6. Encoding Ownership Domains

In this section, we present an encoding of the topology of ownership domains and show how to translate an ownership domains program into a dependent classes program.

6.1 Topology for Ownership Domains

The topology for ownership domains is encoded similarly to Universe types, but we add an additional level: objects are owned by domains and domains are in turn owned by objects. To prevent recursion between the classes, we add appropriate superclasses that abstract away the owner information. Consider the following definitions:

```
class Object {...}
class OObject extends Object {...}
class Domain extends Object {...}
class PublicDomain(OObject owner) extends Domain {...}
class PrivateDomain(OObject owner) extends Domain {...}

class OwnedObject(Domain owner) extends OObject {
  final PrivateDomain(owner: this) owned =
    new PrivateDomain(owner=this);

  static final PublicDomain(owner: null) shared =
    new PublicDomain(owner=null);
}
```

The classes `OObject` and `Domain` do not specify an owner in order to break the recursion. Two subclasses of `Domain` are used to introduce the dependency on an owning object and to distinguish between public and private domains.

Class `OwnedObject` depends on an arbitrary domain as its owner and declares two additional fields. Field `owned` is a default private domain in ownership domains that is used to encapsulate the representation of an object. Each object also has access to a public domain `shared`. As we use the references to `Domains` in dependent types, these fields have to be declared `final`. The field `shared` is declared `static`; therefore, only one public domain object is created and used by all `OwnedObjects`.

We added the distinction between `Object` and `OObject` in order to ensure that in the ownership tree we always alternate between domains and owned objects. As class `Domain` is not a subtype of class `OObject`, the type system can directly prevent that a domain owns another domain.

Consider the following simple ownership domains program:

```
class D {}
class C<data> {
  owned D f = new D();
  void m( data D p ) { /*...*/ }
}
class Demo {
  domain store;
  store D d;
  void main() {
    owned C<store> s;
    s.m(d);
  }
}
```

which can be encoded as the following dependent classes program:

```
class C( Domain owner, Domain data )
extends OwnedObject {
  D(owner: this.owned) f = new D(owner=this.owned);
  void m( D(owner: this.data) p ) {...}
}
class Demo( Domain owner )
extends OwnedObject {
  final PrivateDomain(owner: this) store =
    new PrivateDomain(owner=this);
  D(owner: this.store) d;
  void main() {
    C(owner: this.owned, data: this.store) s;
    s.m(d);
  }
}
```

Note that this encoding of ownership domains can also be used to express Universe types. The `owned` domain corresponds to the representation domain and the `owner` domain corresponds to the `peer` domain. Dependent classes also allow one to express arbitrary owners by using `OObject` as type for the `owner`.

6.2 Translation

Translating a program that uses ownership domains into a program that uses dependent classes is achieved by the following program transformation:

- Each class declaration adds a dependency on the domain `owner`. Formal domain parameters are turned into additional dependent fields. If no superclass is specified explicitly, class `OwnedObject` is used. For instance, the declaration

  ```
  class D<domain dom> {...}
  ```

  is translated into

  ```
  class D( Domain owner, Domain dom )
  extends OwnedObject {...}
  ```

- Each domain declaration is translated into a normal field of the corresponding `Domain` subtype. The program fragment

  ```
  domain dom;
  public domain pubdom;
  ```

  is translated to the new fields

  ```
  final PrivateDomain(owner: this) dom =
    new PrivateDomain(owner=this);
  final PublicDomain(owner: this) pubdom =
    new PublicDomain(owner=this);
  ```

- Uses of domains as owners of types are translated to corresponding dependencies on the `owner` field. The type

  ```
  dom C
  ```

  becomes

  ```
  C(owner: this.dom)
  ```

- Uses of domains as actual arguments are changed to arguments for the corresponding field. The type

  ```
  owned D<data>
  ```

  becomes

  ```
  D(owner: this.owned, dom: this.data)
  ```

- The link and assume declarations from ownership domains are preserved in the dependent classes program. In the following examples, we continue to use the ownership domains syntax, but one can easily devise a conversion to stylized comments or annotations.

Checking whether such a translated program conforms to the encapsulation system that is specified by the link and assume declaration works like in ownership domains. The link and assume declarations are kept in the translated program and can be used to check the link soundness property. Also, the restrictions on link declarations can be checked as in ownership domains. For a detailed discussion of these rules see Sections 2.5 and 3.3 of [2].

6.3 Map Example

Below we show parts of an encoding into dependent classes of the ownership domains example from Section 2.1. The complete ownership domains code can be found in Appendix A.3 and the corresponding encoding in Appendix A.4. Note that we again implicitly use `OwnedObject` as superclass of all classes.

The class and domain declarations from ownership domains:

```
class Map< keyD, valueD > {
  domain nodes;
  public domain iters;
```

are translated to a corresponding dependency of the class and to fields that represent the domains:

```
class Map< K, V >( Domain owner ) {
  final PrivateDomain(owner: this) nodes =
    new PrivateDomain(owner=this);
  final PublicDomain(owner: this) iters =
    new PublicDomain(owner=this);
```

The declaration of field `first` in ownership domains:

```
  nodes Node<keyD, valueD> first;
```

corresponds to:

```
  Node<K, V>(owner: nodes) first;
```

where the ownership is expressed as dependency for the `owner` field. The following ownership domains program uses class `Map`:

```
class Client {
  domain mydata;

  void main(shared Data value) {
    final mydata Map<mydata, shared> map;
    map = new Map<mydata, shared>();
    map.put(new ID(), value);
    map.iters Iter<mydata, shared> iter;
    iter = map.iterator();
    mydata ID id = (ID) iter.getKey();
  }
}
```

This is translated into the following dependent classes program:

```
class Client( Domain owner ) {
  final PrivateDomain(owner: this) mydata =
    new PrivateDomain(owner=this);

  void main(Data(owner: shared) value) {
    final Map<ID(owner: mydata), Data(owner: shared)>
       (owner: mydata) map = new Map<ID(owner: mydata),
                   Data(owner: shared)>(owner=mydata);
    map.put(new ID(owner = mydata), value);
    Iter<ID(owner: mydata), Data(owner: shared)>
       (owner: map.iters) iter = map.iterator();
    ID(owner: mydata) id = iter.getKey();
  }
}
```

Note how the type of local variable `iter` can directly express the path-dependent owner.

7. Examples using MVC

The authors of [23] provide an interpreter called MVC for the vc^n calculus [22]. In this section, we present several examples using MVC — to ensure that our understanding of the calculus is correct and to highlight some interesting cases.

The input language of the MVC interpreter follows the formal calculus vc^n quite closely. It only supports a restrictive language that consists of dependent classes and only supports three kinds of expressions: the `this` reference, reading a field of an expression, and creating new objects. Methods are also encoded as classes and method calls are encoded as object creation. MVC also does not support the omission of fields in type declarations. The input language omits the `class` and `new` keywords. Unfortunately, the supported language subset is too restricted for many interesting examples.

In the following we only perform experiments with encoding the ownership topology in dependent classes. Implementing the translation and checking the encapsulation properties is future work.

We encode the ownership hierarchy as:

```
Object
OObject( Object owner ) extends Object
Root extends Object
```

We always use the following two classes:

```
Data( Object owner ) extends OObject
Main( Object owner ) extends OObject

do( Main(owner: Object) target, Object arg )
: Object extends Object {
  ...
}
main() : Object extends Object {
  do( Main(Root), Data(Root) )
}
```

which corresponds to:

```
class Main {
  void do( any Object arg ) {
    ...
  }

  public static void main(String[] args) {
    new peer Main().do( new peer Data() );
  }
}
```

In the experiments below, this basic framework will be extended by additional classes and an implementation of method do.

Peer Experiments As a first example, we model the class

```
class TestPeer {
  void foo( peer Object p ) {}
  void bar( peer Object p, peer Object q ) {
    foo( q )
  }
}
```

in dependent classes:

```
TestPeer( Object owner ) extends OObject

foo( TestPeer( owner: Object ) target,
     OObject( owner: this.target.owner ) p )
: Object extends Object {
  this
}

bar( TestPeer( owner: Object ) target,
     OObject( owner: this.target.owner ) p,
     OObject( owner: this.target.owner ) q )
: Object extends Object {
  foo( this.target, this.q )
}
```

Note how the types of p and q express that the referenced object has the same owner as the target object, exactly reflecting the meaning of a peer annotation in Universe types. Now we declare:

```
class Main {
  void do( any Object arg )
    new peer TestPeer.foo( new peer Data() );
  }
  ...
}
```

which corresponds to the following dependent class program, which is accepted by the MVC interpreter:

```
do( Main(owner: Object) target, Object arg )
: Object extends Object {
  // OK
  foo( TestPeer(this.target.owner),
       Data(this.target.owner) )
}
```

Other valid implementations for method do are:

```
foo( TestPeer(this.target),
     Data(this.target) ) // and
bar( TestPeer(this.target),
     Data(this.target), Data(this.target) )
```

which correspond to:

```
new rep TestPeer().foo( new rep Data() ); // and
new rep TestPeer().bar( new rep Data(),
                        new rep Data() );
```

In contrast, if we use:

```
foo( TestPeer(this.target),
     Data(this.target.owner) )
```

which corresponds to:

```
new rep TestPeer().foo( new peer Data() );
```

as body of class do, the MVC interpreter reports a type error.

This correctly reflects that the method is called on a TestPeer object that is owned by object target, and that as argument we get a Data object that is owned by the owner of target. This does not match the declaration of class foo, which requires that the argument has the same owner as the target.

Other invalid implementations of method do are:

```
bar( TestPeer(this.target.owner),
     Data(this.target), Data(this.target) ) // and
bar( TestPeer(this.target),
     Data(this.target.owner), Data(this.target) )
```

which correspond to

```
new peer TestPeer().bar( new rep Data(),
                         new rep Data() ); // and
new rep TestPeer().bar( new peer Data(),
                        new rep Data() );
```

All of these examples illustrate violations of Universe type rules, which are caught by the dependent class type checker.

Rep Experiments Next, we investigate the following program that uses the rep modifier

```
class TestRep {
  void bar( rep Object p ) {}
}
```

Note that in Universe types this method is only callable on the receiver this. Any other receiver could not be allowed, as the correct type for the argument is not expressible in Universe types. Class TestRep can be translated into dependent classes as:

```
TestRep( Object owner ) extends OObject

bar( TestRep( owner: Object ) target,
     OObject( owner: this.target ) p )
: Object extends Object {
  this
}
```

The type of parameter p expresses that the argument must be owned by the receiver of the method call. In dependent classes the type for the argument is expressible and therefore the method is callable on receivers other than this. Now we declare class Main as follows:

```
do( Main(owner: Object) target,
    TestRep(owner: Object) arg )
: Object extends Object {
  bar( this.arg, Data(this.arg) )
}

main() : Object extends Object {
  do( Main(Root), TestRep(Root) )
}
```

which corresponds to the following class that uses path-dependent Universe types [40]:

```
class Main {
  void do( any TestRep arg )
    arg.bar( new arg.rep Data() );
  }

  public static void main(String[] args) {
    new peer Main().do( new peer TestRep() );
  }
}
```

Note how a new data object is created that is owned by the receiver of the method call. The above dependent classes code passes through the MVC interpreter. This demonstrates the greater flexibility that is achieved by using dependent classes. If the body of do is changed to the following dependent classes code

```
bar( this.arg, Data(this.owner) )
```

we get an error message from the interpreter, because the argument is not owned by the receiver of the call.

Any Experiments Finally, we experiment with the any modifier. The following Java program with JML annotations

```
class TestAny {
  void anyarg( any Object a ) {}
  void ach( any Object p, any Object q ) {}

  void tung( any Object p, any Object q )
  /*@ requires p.owner == q.owner; @*/
  {}
}
```

declares methods ach and tung. Method ach accepts two references to objects with arbitrary owners.

The declared types of the parameters of method tung are the same and the type system also allows two arbitrary objects. But a JML precondition ensures that the two arguments have the same owner, that is, that they are in the same domain. Universe types can not express this constraint in the type system and the programmer needs to verify that the precondition always holds. However, in the dependent classes translation of the above program, we can express the constraint on method tung:

```
TestAny( Object owner) extends OObject

testpeer( OObject(owner: Object) a,
          OObject(owner: this.a.owner) b )
: Object extends Object

anyarg( TestAny(owner: Object) target, Object a )
: Object extends Object

ach( TestAny(owner: Object) target,
     OObject(owner: Object) p,
     OObject(owner: Object) q )
: Object extends Object {
  anyarg( this.target, this.p )
}

tung( TestAny( owner: Object ) target,
      OObject( owner: Object ) p,
      OObject( owner: this.p.owner ) q )
: Object extends Object {
  testpeer(this.q, this.p )
}
```

The following dependent classes program passes the MVC type checks:

```
do( Main(owner: Object) target, Object arg )
: Object extends Object {
  tung( TestAny(this.target),
        Data(this.arg), Data(this.arg) )
}
```

It corresponds to the following program with path-dependent Universe types:

```
class Main {
  void do( any Object arg ) {
    new rep TestAny().tung( new arg.rep Data(),
                            new arg.rep Data() );
  }
}
```

Both arguments of the call to tung are owned by object arg and the precondition is fulfilled. In contrast, the following implementation of method do creates an error message from MVC because the owners of the two arguments do not match, and dependent classes can detect this error:

```
do( Main(owner: Object) target,
    OObject(owner: Object) arg )
: Object extends Object {
  tung( TestAny(this.target),
        Data(this.arg.owner), Data(this.arg) )
}
```

Summary It was possible to express some interesting examples in the MVC input language and we received the expected results. However, the restrictive language prevented us from doing more involved examples.

8. Related and Future Work

8.1 Related Work

This paper presents first ideas for how ownership type systems can be combined with a dependent classes system. The only previous mention of ownership and virtual classes that we know of is one paragraph in [13] and the abstract of [11]. Unfortunately, there is no paper that corresponds to [11]. For dependent classes, we know of no previous work on expressing ownership.

The dependent classes paper [23] provides additional examples, discusses the relation to virtual classes and multiple dispatch, formalizes the semantics, proves soundness and completeness, and discusses the relation of the single-dispatch subset of dependent classes to several virtual class calculi. The authors also provide an Isabelle formalization of the type soundness proof and the interpreter MVC for the calculus written in Haskell [22].

The dynamic ownership encoding used in Spec# [29] is similar to the approach we took here. However, Spec# uses a ghost field to store the reference to the owner object and relies on theorem proving to check ownership, whereas we use an ordinary field and the type checker of dependent classes. Spec# supports ownership transfer, which is not permitted by dependent classes.

Like our translation to dependent classes, Ownership Generic Java [37] hijacks another type system to express and check ownership, in their case Java's generics. We believe that our encoding is more direct because dependent classes are already parameterized by objects.

8.2 Extending the Translation

An interesting question is the possible runtime overhead of using dependent classes. The dependencies in dependent classes are reflected at runtime to allow dynamic dispatch. In contrast, ownership domains do not need to preserve the reference to the owning domain at runtime. Only domain arguments need to be stored at runtime. Maybe dependent classes could be extended to allow dependent fields that are only used for static checking, but which are not preserved at runtime. For an encoding of Universe types into dependent classes the ownership information needs to be preserved at runtime, because references with arbitrary owners can be cast to references with a specific owner. Such a downcast needs to be checked at runtime and needs the ownership information.

The Isabelle proofs of dependent classes [23] could be combined with our proofs of the Universe type system [26] to show the well-formedness and equivalence of the encoding. Independently, the dependent classes proofs could be extended to prove general ownership properties, for instance, the tree structure or that a specific encapsulation system is enforced.

Once the system is formalized, an implementation of the translation of the ownership type systems and of the encapsulation checkers will allow case studies to be performed.

8.3 Extending Dependent Classes

The recently proposed multiple ownership system [9] generalizes ownership from tree structures to directed acyclic graphs. These DAGs cannot be directly encoded in dependent classes, as each dependent field only gets exactly one argument. One could envision to solve this problem using arrays as owners.

Some ownership type systems allow ownership transfer [14, 33]. In dependent classes all dependent fields are final and cannot be changed. It will be interesting to see whether the ideas of ownership transfer can be generalized to transfer in dependent classes.

8.4 Extending Ownership Type Systems

We are investigating how the increased expressiveness of dependent classes can be used for ownership type systems.

External Domain Declarations Should one only allow dependent classes programs that were generated from an ownership domains or Universe types program or should more flexible programs be allowed? For example, programs that are generated from our transformation only contain domains that have the current object as owner. If we allow arbitrary programs, we might write

```
class C {
  final Object x;
  final PublicDomain(owner: x) f =
   new PublicDomain(owner=x);
}
```

This would add an additional public domain to object x, something that cannot be expressed in ownership domains. This might be useful to associate some extra data with an object, but the implications of such flexibility have to be studied carefully. For example, ownership domains forbid to create an object in a public domain of an other object. Objects in a public domain can gain access to private domains by corresponding link declarations. If anyone could create objects in a public domain the security of the system could be subverted.

Static Domains The only domain that is globally accessible in ownership domains is `shared`, the root domain of the system. In our translation, we modeled this with a static field in class `OwnedObject`:

```
static final PublicDomain(owner: null) shared =
  new PublicDomain(owner=null);
```

We are investigating the use of other static domains to model the sharing between objects, that are in different locations in the ownership tree.

```
class Data( Domain owner ) {
  int do() { ... }
}
class Demo( Domain owner ) {
  static final PrivateDomain(owner: null) cache =
   new PrivateDomain(owner=null);

  static Data(owner: Demo.cache) data = null;

  public int do() {
    if( data == null ) {
      data = new Data(owner=Demo.cache);
    }
    return data.do();
  }
}
```

Objects of class `Data` provide some functionality, in the above example for simplicity the computation of some integer. Class `Demo` declares a static private domain `cache` and uses this domain as owner for the static field `data`. The instance method `do` checks whether the `Data` object was already created, creates one if necessary, and uses it for the computation. Different instances of class `Demo` have different owners, but all of them share the same `cache` domain. This allows us to share objects just between the objects of some class and to control the access to these objects. If we used the global `shared` domain instead, we would lose all knowledge about the shared objects.

Encapsulation Systems The flexibility to express more topological structures makes it necessary to also investigate corresponding encapsulation systems. The link declarations of ownership domains allow one to express many disciplines, but the resulting structures can become complex [34]. The owner-as-dominator and owner-as-modifier disciplines are simpler to apply, but forbid the use of some ownership topologies. Finding the right balance between flexibility, ease of use, and guaranteeing certain properties remains an open issue.

9. Conclusions

In this paper we presented first results on how different ownership type systems can be expressed on top of dependent classes. Our contributions are three-fold. First, we analyze how the topological structure of ownership type systems can be encoded in dependent classes. Second, we describe how programs that use ownership domains and Universe types can be translated into corresponding dependent classes programs. Third, we discuss how the encapsulation properties can be checked directly on the dependent classes program.

Is the research in ownership type systems over, once we have a mapping into dependent classes? Definitely not. Ownership type systems provide a concise and expressive language to express desired encapsulation properties. Writing them directly in dependent classes might be possible, but would be very tedious for the programmer or specifier.

Also, one might take advantage of the additional information and constraints available in an ownership type system. For example, in Generic Universe Types [17], we use the knowledge that the owner-as-modifier discipline is enforced to allow more flexible subtyping. This information would be lost in the translation to dependent classes.

On the other hand, basing future ownership type systems on dependent classes might simplify the soundness proofs, give more expressiveness, and allow the experimentation with different encapsulation systems.

Acknowledgments

We are grateful to Joseph N. Ruskiewicz and the FOOL '08 reviewers for their helpful comments, and to Vaidas Gasiunas for improvements to the MVC interpreter and helpful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, 2003.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.

[3] C. Andrea, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time systems. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2006.

[4] A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM, 2002.

[5] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[6] 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)*, pages 211–230. ACM, 2002.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM, 2003.

[8] C. Boyapati, A. Salcianu, Jr. W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Programming language design and implementation (PLDI)*, pages 324–337. ACM, 2003.

[9] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM, 2007.

[10] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[11] D. Clarke. Nested classes, nested objects and ownership. Invited talk at FOOL/WOOD '06, `www.research.att.com/~kfisher/FOOL/FOOLWOOD06/program.html#clarke`, 2006.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM, 2002.

[13] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A simple virtual class calculus. In *Aspect-Oriented Software Development (AOSD)*, 2007.

[14] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[15] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10). ACM, 1998.

[16] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, and P. Müller. UT: Type Soundness for Universe Types. To appear.

[17] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 28–53. Springer-Verlag, 2007.

[18] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8), 2005.

[19] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[20] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 303–326. Springer-Verlag, 2001.

[21] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Principles of programming languages (POPL)*, pages 270–282. ACM, 2006.

[22] V. Gasiunas, M. Mezini, and K. Ostermann. vcn - a calculus for multi-dimensional virtual classes. `www.st.informatik.tu-darmstadt.de/static/pages/projects/mvc/index.html`.

[23] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–152. ACM, 2007.

[24] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Ho lt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger, Report on ECOOP'91 workshop W3*, 3(2):11–16, 1992.

[25] A. Igarashi and M. Viroli. Variant path types for scalable extensibility. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 113–132. ACM, 2007.

[26] M. Klebermaß. An Isabelle formalization of the Universe type system. Master's thesis, ETH Zurich, 2007. `sct.inf.ethz.ch/projects/student_docs/Martin_Klebermass/`.

[27] N. Krishnaswami and J. Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *Programming language design and implementation (PLDI)*, pages 96–106. ACM, 2005.

[28] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `www.jmlspecs.org`, 2006.

[29] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[30] O. L. Madsen and B. Møller-Pedersen. Virtual classes - a powerful mechanism in object-oriented programming. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 1989.

[31] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[32] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[33] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.

[34] S. Nägeli. Ownership in design patterns. Master's thesis, ETH Zurich, 2006. `sct.inf.ethz.ch/projects/student_docs/Stefan_Naegeli/`.

[35] J. Noble, J. Vitek, and J. M. Potter. Flexible alias protection. In E. Jul, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*. Springer-Verlag, 1998.

[36] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, 2003.

[37] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 311–324. ACM, 2006.

[38] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 2007.

[39] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.

[40] D. Schregenberger. Universe type system for Scala. Master's thesis, ETH Zurich, 2007. `sct.inf.ethz.ch/projects/student_docs/Daniel_Schregenberger/`.

A. More Example Material

A.1 Un-annotated Map Example

For reference, here is the un-annotated source code for the map example.

```
class Map< K, V > {
  Node<K, V> first;

  void put(K key, V value) {
    Node<K, V> newfirst = new Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  V get(K key) {
    Iter<K, V> i = iterator();
    while (i.hasNext()) {
      if (i.getKey().equals(key))
          return i.getValue();
      i.next();
    }
    return null;
  }

  Iter<K, V> iterator() {
    IterImpl<K, V> res = new IterImpl<K, V>();
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V > {
  K key; V value; Node<K, V> next;
  void init(K k, V v, Node<K, V> n)
    { key = k; value = v; next = n; }
}

interface Iter< K, V > {
  K getKey();
  V getValue();
  boolean hasNext();
  void next();
}

class IterImpl< K, V > implements Iter< K, V > {
  Node<K, V> current;

  void setCurrent(Node<K, V> c) { current = c; }
  K getKey() { return current.key; }
  V getValue() { return current.value; }
  boolean hasNext() { return current != null; }
  void next() { current = current.next; }
}

class ID { /*...*/ }
class Data { /*...*/ }

class Client {
  void main(Data value) {
    Map<ID, Data> map = new Map<ID, Data>();
    map.put(new ID(), value);
    Iter<ID, Data> iter = map.iterator();
    ID id = iter.getKey();
  }
}
```

A.2 Ownership Types

The first type system that was expressive enough for common design idioms was ownership types [35, 15].

Ownership Generic Java (OGJ) [37] uses Java 5 style type parametrization to encode the ownership information. This allows the authors to reuse parts of the Featherweight Generic Java formalization and makes the formalization of OGJ elegant.

Ownership types can be encoded in ownership domains. The default links and assumptions in ownership domains enforce the owner-as-dominator property that is also enforced by ownership types.

The map example can be written in OGJ as

```
class Map< K extends Object<KOwner>,
           V extends Object<VOwner>,
           Owner extends World > {
  Node<K, V, This> first;

  void put(K key, V value) {
    Node<K, V, This> newfirst;
    newfirst = new Node<K, V, This>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  V get(K key) {
    Iter<K, V, Owner> i = iterator();
    while (i.hasNext()) {...}
    return null;
  }

  Iter<K, V, Owner> iterator() {
    IterImpl<K, V, Owner> res;
    res = new IterImpl<K, V, Owner>();
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V, Owner extends World > {
  K key; V value; Node<K, V, Owner> next;
  void init(K k, V v, Node<K, V, Owner> n) {
    key = k; value = v; next = n;
  }
}

interface Iter< K, V > {
    K getKey();
    V getValue();
    boolean hasNext();
    void next();
}

class Client< Owner extends World > {
  void main(Data<World> value) {
    Map<ID<This>, Data<World>, This> map;
    map = new Map<This>();
    map.put(new ID<This>(), value);
    Iter<ID<This>, Data<World>, This> iter;
    iter = map.iterator();
    ID<This> id = iter.getKey();
  }
}
```

A.3 Ownership Domains

The ownership domains code in this paper was tested with ArchJava 1.3.2. ArchJava only supports parameterization by domains, not by types.

The complete non-type-generic map example is

```
class Map< keyD, valueD > {
  domain nodes;
  public domain iters;

  link iters -> nodes;
  nodes Node<keyD, valueD> first;

  void put(keyD Object key, valueD Object value) {
    nodes Node<keyD, valueD> newfirst;
    newfirst = new Node<keyD, valueD>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  valueD Object get(keyD Object key) {
    iters Iter<keyD, valueD> i = iterator();
    while (i.hasNext()) {
      if (i.getKey().equals(key))
          return i.getValue();
      i.next();
    }
    return null;
  }

  iters Iter<keyD, valueD> iterator() {
    iters IterImpl<keyD, valueD, nodes> res =
        new IterImpl<keyD, valueD, nodes>();
    res.setCurrent(first);
    return res;
  }
}

class Node< keyD, valueD > {
  keyD Object key; valueD Object value;
  owner Node<keyD, valueD> next;
  void init(keyD Object k, valueD Object v,
           owner Node<keyD, valueD> n)
    { key = k; value = v; next = n; }
}

interface Iter< keyD, valueD > {
    keyD Object getKey();
    valueD Object getValue();
    boolean hasNext();
    void next();
}

class IterImpl< keyD, valueD, nodeD>
implements Iter<keyD, valueD> {
  nodeD Node<keyD, valueD> current;

  void setCurrent(nodeD Node<keyD, valueD> c)
    { current = c; }
  keyD Object getKey() { return current.key; }
  valueD Object getValue() { return current.value; }
  boolean hasNext() { return current != null; }
  void next() { current = current.next; }
}

class ID { /*...*/ }
class Data { /*...*/ }

class Client {
  domain mydata;

  void main(shared Data value) {
    final mydata Map<mydata, shared> map;
    map = new Map<mydata, shared>();
    map.put(new ID(), value);
    map.iters Iter<mydata, shared> iter;
    iter = map.iterator();
    mydata ID id = (ID) iter.getKey();
  }
}
```

The following version follows the syntax of [2], but could not be checked with a compiler.

```
class Map< K, V > {
  domain nodes;
  public domain iters;
  link iters -> nodes;

  nodes Node<K, V> first;

  void put(K key, V value) {
    nodes Node<K, V> newfirst;
    newfirst = new nodes Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  V get(K key) {
    iters Iter<K, V> i = iterator();
    while (i.hasNext()) {...}
    return null;
  }

  iters Iter<K, V> iterator() {
    iters IterImpl<K, V, nodes> res =
        new iters IterImpl<K, V, nodes>();
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V > {
  K key; V value; owner Node<K, V> next;
  void init(K k, V v, owner Node<K, V> n)
    { key = k; value = v; next = n; }
}

interface Iter< K, V > {
    K getKey();
    V getValue();
    boolean hasNext();
    void next();
}

class IterImpl< K, V, domain nodes>
extends Iter<K, V> {
  nodes Node<K, V> current;

  void setCurrent(nodes Node<K, V> c)
    { current = c; }
  K getKey() { return current.key; }
  V getValue() { return current.value; }
  boolean hasNext() { return current != null; }
  void next() { current = current.next; }
}

class Client {
  domain mydata;

  void main(shared Data value) {
    final mydata Map<mydata ID, shared Data> map;
    map = new mydata Map<mydata ID, shared Data>();
    map.put(new mydata ID(), value);
    map.iters Iter<mydata ID, shared Data> iter;
    iter = map.iterator();
    mydata ID id = iter.getKey();
  }
}
```

A.4 Ownership Domains Encoded in Dependent Classes

The following code is the encoding of the map example using dependent classes.

```
class Map< K, V >( Domain owner ) {
  final PrivateDomain(owner: this) nodes =
    new PrivateDomain(owner=this);
  final PublicDomain(owner: this) iters =
    new PublicDomain(owner=this);

  link iters -> nodes;

  Node<K, V>(owner: nodes) first;

  void put(K key, V value) {
    Node<K, V>(owner: nodes) newfirst;
    newfirst = new Node<K, V>(owner = nodes);
    newfirst.init(key, value, first);
    first = newfirst;
  }

  V get(K key) {
    Iter<K, V>(owner: iters) i = iterator();
    while (i.hasNext()) {...}
    return null;
  }

  Iter<K, V>(owner: iters) iterator() {
    IterImpl<K, V>(owner: iters,
                   nodes: this.nodes) res;
    res = new IterImpl<K, V>(owner=iters,
                             nodes=this.nodes);
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V >( Domain owner ) {
  K key; V value;
  Node<K, V>(owner: this.owner) next;
  void init(K k, V v,
            Node<K, V>(owner: this.owner) n) {
    key = k; value = v; next = n;
  }
}

interface Iter< K, V >( Domain owner ) {
    K getKey();
    V getValue();
    boolean hasNext();
    void next();
}

class IterImpl< K, V >( Domain owner,
                        Domain nodes )
implements Iter< K, V > {
  Node<K, V>(owner: this.nodes) current;

  void setCurrent(Node<K, V>(owner: this.nodes) c)
    { current = c; }
  K getKey() { return current.key; }
  V getValue() { return current.value; }
  boolean hasNext() { return current != null; }
  next() { current = current.next; }
}

class ID( Domain owner ) { ... }
class Data( Domain owner ) { ... }

class Client( Domain owner ) {
  final PrivateDomain(owner: this) mydata =
    new PrivateDomain(owner=this);

  void main(Data(owner: shared) value) {
    final Map<ID(owner: mydata), Data(owner: shared)>
       (owner: mydata) map = new Map<ID(owner: mydata),
                  Data(owner: shared)>(owner=mydata);
    map.put(new ID(owner = mydata), value);
    Iter<ID(owner: mydata), Data(owner: shared)>
       (owner: map.iters) iter = map.iterator();
    ID(owner: mydata) id = iter.getKey();
  }
}
```

A.5 Universe Types

The description of the Universe type system in [18] discussed how the ownership structure can be enforced by an encoding as object invariants. For this purpose, a ghost field (a specification-only field) `owner` is declared in class `java.lang.Object` and for a `peer` field `pf` the invariant

```
invariant pf == null || pf.owner == this.owner;
```

is added to the object invariant of the declaring class. Similarly, for a `rep` field `rf` the invariant

```
invariant rf == null || rf.owner == this;
```

is added. Fields with the `any` modifier do not give rise to an additional invariant.

Instead of expressing ownership as invariant on a ghost field, we make the dependency explicit in dependent classes.

Generic Universe Types [17] are implemented in an extension to the MultiJava compiler and JML checker and runtime system [28]. The complete source code for the map example is

```
class Map< K, V > {
  rep Node<K, V> first;

  void put(K key, V value) {
    rep Node<K, V> newfirst;
    newfirst = new rep Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  pure V get(K key) {
    peer Iter<K, V> i = iterator();
    while (i.hasNext()) {
      if (i.getKey().equals(key))
          return i.getValue();
      i.next();
    }
    return null;
  }

  pure peer Iter<K, V> iterator() {
    peer IterImpl<K, V> res;
    res = new peer IterImpl<K, V>();
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V > {
  K key; V value; peer Node<K, V> next;
  void init(K k, V v, peer Node<K, V> n)
    { key = k; value = v; next = n; }
}

interface Iter< K, V > {
    pure K getKey();
    pure V getValue();
    pure boolean hasNext();
    void next();
}

class IterImpl< K, V > implements Iter< K, V > {
  any Node<K, V> current;
  void setCurrent(any Node<K, V> c)
    { current = c; }
  pure any K getKey() { return current.key; }
  pure any V getValue()
    { return current.value; }
  pure boolean hasNext()
    { return current != null; }
  void next() { current = current.next; }
}

class ID { /*...*/ }
class Data { /*...*/ }

class Client {
  void main(any Data value) {
    rep Map<rep ID, any Data> map =
        new rep Map<rep ID, any Data>();
    map.put(new rep ID(), value);
    rep Iter<rep ID, any Data> iter;
    iter = map.iterator();
    rep ID id = iter.getKey();
  }
}
```

A.6 Universe Types Encoded in Dependent Classes

The following code is the encoding of the map example using dependent classes.

```
class Map< K, V >( Object owner ) {
  Node<K, V>(owner: this) first;

  void put(K key, V value) {
    Node<K, V>(owner: this) newfirst;
    newfirst = new Node<K, V>(owner = this);
    newfirst.init(key, value, first);
    first = newfirst;
  }

  V get(K key) {
    Iter<K, V>(owner: this.owner) i = iterator();
    while (i.hasNext()) {...}
    return null;
  }

  Iter<K, V>(owner: this.owner) iterator() {
    IterImpl<K, V>(owner: this.owner,
                map: this) res;
    res = new IterImpl<K, V>(owner=this.owner,
                             map=this);
    res.setCurrent(first);
    return res;
  }
}

class Node< K, V >( Object owner ) {
  K key; V value;
  Node<K, V>(owner: this.owner) next;
  void init(K k, V v,
           Node<K, V>(owner: this.owner) n) {
    key = k; value = v; next = n;
  }
}

interface Iter< K, V >( Object owner ) {
    K getKey();
    V getValue();
    boolean hasNext();
    void next();
}

class IterImpl< K, V >( Object owner,
                         Map<K, V>(owner: Object) map )
implements Iter< K, V > {
  Node<K, V>(owner: this.map) current;

  void setCurrent(Node<K, V>(owner: this.map) c)
    { current = c; }
  K getKey() { return current.key; }
  V getValue() { return current.value; }
  boolean hasNext() { return current != null; }
  next() { current = current.next; }
}

class ID( Object owner ) { ... }
class Data( Object owner ) { ... }

class Client( Object owner ) {
  void main(Data(owner: root) value) {
    Map<ID(owner: this), Data(owner: root)>
       (owner: this) map;
    map = new Map<ID(owner: this),
                  Data(owner: root)>(owner=this);
    map.put(new ID(owner = this), value);
    Iter<ID(owner: this),
         Data(owner: root)>(owner: this) iter;
    iter = map.iterator();
    ID(owner: this) id = iter.getKey();
  }
}
```