VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

Generic Ownership

Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle
School of Mathematical and Computing Sciences,
Victoria University of Wellington
New Zealand
{alex, kjx, robert}@mcs.vuw.ac.nz
[1] Institute of Information and Computing Sciences,
Utrecht University
The Netherlands
dave@cs.uu.nl

Technical Report CS-TR-03/16
December 2003

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Generic Ownership

Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle
School of Mathematical and Computing Sciences,
Victoria University of Wellington
New Zealand
{alex, kjx, robert}@mcs.vuw.ac.nz
[1] Institute of Information and Computing Sciences,
Utrecht University
The Netherlands
dave@cs.uu.nl

Technical Report CS-TR-03/16
December 2003

Abstract

Modern programming languages provide little support for object encapsulation and ownership. Escaped aliases to private objects can compromise both security and reliability of code in reference-abundant languages such as Java. Object ownership is a widely accepted approach to controlling aliasing in programming languages. Proposals for adding ownership to programming languages do not directly support type genericity. We present Generic Ownership — a unified approach to providing generics and ownership. By including support for default ownership, Generic Ownership imposes no more syntactic or runtime overheads than traditional generic types. We have implemented Generic Ownership in the context of the Ownership Generic Java (OGJ) programming language, an extension to Generic Java, and we ground the formal side of this work within the Featherweight Generic Java framework. We hope that our work will help bring full support for object encapsulation to the mainstream programming world.

Publishing Information

Submitted to OOPSLA2004.

Author Information

Alex Potanin is a PhD Student supervised by James Noble and Robert Biddle and Dave Clarke is based in CWI, Netherlands and completed his PhD under James Noble several years ago

Generic Ownership

Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle
School of Mathematical and Computing Sciences, Victoria University of Wellington
New Zealand
{alex, kjx, robert}@mcs.vuw.ac.nz
[1] Institute of Information and Computing Sciences, Utrecht University
The Netherlands
dave@cs.uu.nl

ABSTRACT

Modern programming languages provide little support for object encapsulation and ownership. Escaped aliases to private objects can compromise both security and reliability of code in reference-abundant languages such as Java. Object ownership is a widely accepted approach to controlling aliasing in programming languages. Proposals for adding ownership to programming languages do not directly support type genericity. We present Generic Ownership — a unified approach to providing generics and ownership. By including support for default ownership, Generic Ownership imposes no more syntactic or runtime overheads than traditional generic types. We have implemented Generic Ownership in the context of the Ownership Generic Java (OGJ) programming language, an extension to Generic Java, and we ground the formal side of this work within the Featherweight Generic Java framework. We hope that our work will help bring full support for object encapsulation to the mainstream programming world.

1. INTRODUCTION

Object instance encapsulation ensures that objects cannot be leaked beyond an object or collection of objects which *own* them. There are two main approaches to object encapsulation in the literature: enforcing coding conventions within an existing programming language, or significantly modifying a language to allow ownership support. The first approach is taken by Islands [24] and various kinds of Confined Types [9, 19]. Programs must be written to follow a set of specific conventions that can be checked to see if they provide containment guarantees [23]. The soundness of this approach has been been proven only recently [45]. Support for generics is added on top of such collections of restrictions for enforcing encapsulation [45].

The second approach is taken by languages such as Joe, Universes, AliasJava, and SafeConcurrentJava [17, 33, 4, 13]. Ownership parameterisation is added to the syntax and expressed explicitly within the type system of these languages. All of these type systems are distinct: they all employ ownership parameterisation, but none has support for type genericity.

Why would we want ownership and generic types? Consider for example a *box* as a kind of object. In any object-oriented language we are allowed to say: "this is a box" (meaning any box of any things). In a language with generics, we are allowed to say: "this is a box of books" (denoting a box of books, but not containing birds). In a language with ownership parameterisation, we are allowed to say: "this is my box" or "these are library books". Combining ownership and generics naturally allows us to say: "this is my box of library books", not birds and not my personal books. This illustrates the main idea of this paper: combining ownership and genericity. Ownership works exceptionally well with genericity, both in theory, practice, and implementation, as we describe below.

This paper continues efforts to provide effective object encapsulation within practical programming languages. We present Generic Ownership that uses a single parameter space to carry *both* generic type and ownership type information. We show that ownership systems can be subsumed completely within parametric polymorphic type systems.

We demonstrate the practicability of Generic Ownership by designing an extension to Generic Java, which we name Ownership Generic Java (OGJ). OGJ uses pure Java syntax with no modifications and, via a defaulting mechanism, allows seamless use of Generic Java classes and OGJ classes with owner parameters. We formalise our approach within the context of Featherweight Generic Java (FGJ) [26] and find that a static confinement invariant falls out "almost for free" with a surprisingly straightforward proof.

The contributions of this paper include: *generic ownership* — a unified combination of genericity and ownership that allows a single parameter space to carry both type and ownership information; a simple language design for Ownership Generic Java (OGJ) that provides generics, ownership, and confinement, using this single unified approach; a formal model for OGJ, with which we have proved a confinement invariant for the static part of the language; and an implementation of OGJ that provides backwards compatibility with GJ, and is sufficiently mature to compile significant GJ and OGJ programs including generic collections and its own compiler.

Outline. Section 2 reviews the generics support in Java and overviews the area of object encapsulation using owner-

```
public class SimpleMap {
  private Vector mapNodes;

  void put(Comparable key, Object value) {
    mapNodes.add(new MapNode(key,value));
  }

  Object get(Comparable k) {
    Iterator i = mapNodes.iterator();
    while (i.hasNext()) {
      MapNode mn = (MapNode) i.next();
      if (((Comparable) mn.key).equals(k)) {
        return mn.value;
      }
    }
    return null;
  }
}

class MapNode {
  public Object key;
  public Object value;
  MapNode(Object key, Object value) {
    this.key = key; this.value = value;
  }
}
```

Figure 1: A Java implementation of a `SimpleMap` class

ship. Section 3 describes OGJ. Section 4 gives an overview of FGJ+c — our simple set of type rules describing a subset of FGJ programs within which we can prove a static confinement invariant. We then discuss how we plan to extend this formalism to incorporate other constructs that are included in the OGJ language, such as shallow ownership and defaulting. Section 5 compares our approach with related work and argues why Generic Ownership is a practical design that can provide ownership support for mainstream programming. Section 6 concludes the paper.

2. BACKGROUND

Genericity and Ownership are two language mechanisms that, in different ways, allow programmers to make the intentions behind their code more explicit. This can provide programmers with more support, typically by detecting errors statically, at compile time, that could otherwise only be detected (or worse, remain undetected) once the program is run.

In this section, we will illustrate the advantages and impacts of genericity and ownership with reference to a simple example, shown in figure 1. This example is a small part of an implementation of a `SimpleMap` class that uses neither genericity nor ownership. The map is implemented using a `Vector` containing a number of `MapNode`s, each of which stores a key-value pair. The main `SimpleMap` class provides methods to insert a new key-value pair into the map and to return the first value associated with a particular key.

2.1 Genericity

Genericity allows us to use type parameters to give a better description of the type of the variable we are dealing with. This allows more sensible collections (e.g. of `MapNode`s rather than *anything*), better compile-time error detection, and more readable and reusable code. For example, the code in figure 1 exhibits a number of well-known weaknesses [15, 28]. One prominent weakness is that it relies upon subtyping to store objects of various types within the map itself and within the `Vector` implementing the map. That is, both the map and the vector simply store `Object`s (or `Comparable`s). Considering the vector, the map may store `MapNode`s into the `Vector` object, because Java `Vector`s hold any kind of `Object` and a `MapNode` is a kind of `Object`: however, when getting a `MapNode` from the `Vector` (via an iterator in this case) the code requires a dynamic cast to ensure the object returned actually is a `MapNode`. Malicious or buggy code, for example, could insert some other kind of object into that `Vector`, causing this cast to fail at runtime and triggering an exception.

This is an old problem, and the solutions are equally old, dating back to the mid-1970s [32]: the class definitions must be made generic (or parametrically polymorphic) so that particular instances of the classes can be created for particular argument types. While being widely adopted and appreciated in functional languages, and in specialised object-oriented languages such as Eiffel and Modula-3, genericity is only now on the cusp of acceptance in more popular object-oriented languages such as Java and C#. There have been a range of proposals to add genericity to Java [2, 34, 39, 15]; one of which, GJ [15], has been adopted as the basis for future versions of Java [42]. C# is developing along similar lines [28]. Figure 2 presents a version of the `SimpleMap` class written using genericity, in a Generic Java syntax.

Comparing figures 1 and 2 illustrates both the advantages and disadvantages of generic types. Regarding the advantages, the types of objects stored in the vector or map can now be preserved when they are returned, so there is no need for a typecast when objects are removed from the `Vector`. Method declarations can also carry more information, using generic type parameters like "`Key`" or "`Value`" instead of "`Object`" or "`Comparable`". As a result, only objects of the right types can be stored into Maps and Vectors; attempts to store the wrong types of objects will be detected at compile time.

The main disadvantage is that figure 2 is more complex than figure 1: in particular, the class definitions in the generic version declare formal generic type parameters, and then class instantiations must provide actual values for those parameters, resulting in expressions like "`Vector<MapNodes <Key,Value>> mapNodes`" rather than "`Vector mapNodes`".

2.2 Ownership

An object is *aliased* whenever there is more than one pointer referring to that object [25]. Aliasing can cause a range of difficult problems within object-oriented programs, because one referring object can change the state of the aliased object, implicitly affecting all the other referring objects [36, 9]. Aliasing is endemic and unavoidable in object-oriented programming languages, as any assignment statement may cause an extra alias to be created. To deal with such problems, object encapsulation has been widely studied in literature.

Islands, confinement, and ownership are all essentially forms of object encapsulation [35]. All these schemes are attempts to establish an encapsulation *boundary* that pro-

```
public class SimpleMap<Key extends Comparable,
                       Value> {
  private Vector<MapNode<Key,Value>> mapNodes;

  void put(Key key, Value value) {
    mapNodes.add(
      new MapNode<Key, Value>(key,value));
  }

  Value get(Key k) {
    Iterator<MapNode<Key,Value>> i
      = mapNodes.iterator();
    while (i.hasNext()) {
      MapNode<Key, Value> mn = i.next();
      if (mn.key.equals(k)) {
        return mn.value;
      }
    }
    return null;
 }
}

class MapNode<Key extends Comparable, Value> {
  public Key key;
  public Value value;
  MapNode(Key key, Value value) {
    this.key = key; this.value = value;
  }
}
```

Figure 2: A Generic implementation of a `SimpleMap` class

tects some objects *inside* the boundary from direct access by other objects *outside* that boundary. Where these proposals differ from earlier programming language encapsulation and module systems is that they restrict access to objects at runtime: that is, they constrain values of pointers or references to objects in object-oriented systems, rather than merely accesses to field and method names.

These schemes enforce a *containment invariant* that simply states that objects outside a particular boundary may not access objects inside that boundary. For example, in Confined Types [9], the unit of confinement is a Java package: all the instances of public classes within that package form the encapsulation boundary; all instances of private classes (known as *confined classes*) are inside the boundary, and instances of classes in any other package are outside the boundary. This means that a class may access instances of a public class belonging to any package, but may only access instances of those confined classes belonging to its own package.

What confinement means in practice is that code written in one protection domain (say a Java package) should, when executed, never *directly* refer to an instance of a class inside the boundary of another protection domain. Static references, such as those stored in object fields, must be restricted: a field of a class cannot hold a reference to an object that belongs inside a different package. The execution of a class's methods must also be restricted. Methods cannot access confined classes of other packages. Note, however, that this prohibition refers only to direct accesses: *indirect* access via boundary objects is permitted — indeed, is encouraged. Public classes (or instances of public classes) thus provide an interface to the private instances in their package.

Ownership allows a more granular control of which objects are allowed to have references to which objects. This fact allowed ownership types to be used widely to group together objects used inside the program for both practical and theoretical applications of ownership types [8, 31, 10, 3]. Clarke [16] formulated a containment invariant for ownership, which determines when an object can refer to another, based on their relative nesting. An object cannot refer to objects nested within another, unless it too is nested within the object. Again this reduces to the fact that objects outside a boundary cannot refer to objects inside that boundary: with object ownership, boundaries correspond to objects that may be nested within one another.

Compared to genericity, ownership addresses different kinds of errors in programs. To return to our example, considering the basic `SimpleMap` implementation from figure 1, the `mapNodes` field containing the `Vector` in the map is declared as private, and so Java will ensure that the field can only be accessed from within the `SimpleMap` class. Presumably this is done because the `Vector` is an internal part of the implementation of the `SimpleMap` class, and should not be accessed outside, and inserting or removing elements from the vector, or perhaps acquiring (but not releasing) its internal lock would break the invariants of the `SimpleMap` class and again cause runtime errors.

Unfortunately, the name based protection used in Java and most other programming languages is not strong enough to keep the `Vector` truly private to the `SimpleMap`. A malicious or erroneous programmer could insert a public method that exposed the vector, e.g.,

```
public Vector exposeVector() {return mapNodes;}
```

with no objection from the compiler. Any resulting errors will be subtle, possibly appearing at runtime long after the execution of the `exposeVector` method, and thus be difficult to identify and resolve. These kinds of errors have been identified as occurring in many Java libraries [41] and have caused significant problems for language security mechanisms [9].

Ownership Types [18, 17, 33, 4, 13] allow programmers to protect access to objects, rather than just the names or variables used to store them. Figure 3 gives an example of the `SimpleMap` class using ownership types. The syntax used is proposed by Boyapati [10, figure 2.7] in a work that also shows a wide applicability of ownership types in practice.

Comparing figure 3 with figures 1 and 2 illustrates both the strengths and weaknesses of ownership types. The most obvious difference is the presence of a range of *ownership parameter* definitions such as "`<mapOwner, keyOwner, valueOwner>`" on the class declarations. This declares a parameter, `mapOwner`, for the owner of the instances of the classes being declared, with further ownership parameters "`keyOwner`" and "`valueOwner`" describing the ownership of the keys and values that will be stored in the map. These parameters are then instantiated as the types are used, as when a `MapNode` is created within the `put` method of `SimpleMap`.

Note also that ownership parameters can be instantiated via the keyword "`this`", which ensures that the current object (the object usually denoted "`this`" in Java) *owns* the object being declared [18]. The `Vector` object is marked in

```
public class SimpleMap<mapOwner, keyOwner, valueOwner> {
  private Vector<this, this> mapNodes;

  void put(Comparable<keyOwner> key, Object<valueOwner> value) {
    mapNodes.add(new MapNode<this, keyOwner, valueOwner>(key, value));
  }

  Object<valueOwner> get(Comparable<keyOwner> key) {
    Iterator<this, this> i = mapNodes.iterator();
    while (i.hasNext()) {
      MapNode<this, keyOwner, valueOwner> mn =
        (MapNode<this, keyOwner, valueOwner>) i.next();
      if (mn.key.equals(key)) {
        return mn.value;
      }
    }
    return null;
  }
}

class MapNode<mapNodeOwner, keyOwner, valueOwner>
  public Comparable<keyOwner> key;
  public Object<valOwner> value;
  MapNode(Comparable<keyOwner> key, Object<valueOwner> value) {
    this.key = key; this.value = value;
  }
}
```

Figure 3: An Ownership Types implementation of a `SimpleMap` class (after Boyapati [10])

this way as being owned by the `SimpleMap`, for example, so any attempt to access or pass the `Vector` object outside the `SimpleMap` object will be detected and prevented at compile time. Code such as the `exposeVector()` method will be unable to cause any damage by breaching encapsulation.

The ownership parameters carry ownership around the program, so that the ownership status of the keys and values can be maintained outside the `SimpleMap`. For example, the ownership of the keys and values may be specified by each instantiation of the `SimpleMap` class, but by using the `keyOwner` and `valueOwner` ownership parameters, the fields that will store keys and values inside the subsidiary `MapNode` objects will have the correct ownership for these fields.

The main disadvantage of ownership types is quite similar to that of generic types: added syntactic complexity, with ownership parameters. In fact, this code also has all the type-related problems as the "straight" Java code: the problems that are addressed by genericity. This code relies on subtyping to store different types of objects, and so needs type casts when objects are removed from the `Vector` (or subsequently from the `SimpleMap`). Although the code may look generic, all the type declarations are simple Java types, such as `Object` or `Comparable`, with all the problems that involves. While the `Vector` stored in the `mapNodes` field can no longer be exposed out of the `SimpleMap` instance that owns it, malicious or buggy programming within that class can once again break code by directly inserting incorrect types into the vector.

2.3 Combining Genericity and Ownership

The state of the art, then, is that there are two separate but similar techniques that constrain which objects may be accessed by which types, fields, or expressions. Genericity constrains these accesses by compile-time types, while ownership constrains the accesses by compile-time object structures. These two mechanisms appear to be orthogonal, raising the question: could they both be included within a single programming language? Figure 4 repeats the `SimpleMap` example using a hypothetical language separately supporting both genericity (parameters marked with `[` and `]`) and ownership (parameters marked with `<` and `>`). The syntax in this figure is based on Boyapati [10, page 29] and bears resemblance to the Flexible Alias Protection proposal [37].

Again we can compare figure 4 with the preceeding figures 1, 2, and 3. Basically, this code now has both type and owner parameters, each taken from their respective languages. As with the generic system, types can be instantiated for keys and values, removing the reliance on subtyping and the associated fragile type casts. As with the ownership system, objects can be tagged as owned by `this`, ownership can be recorded via owner parameters, and so any exposing method would be detected and prevented.

Unfortunately, the syntax required to implement both ownership and genericity separately means that this code is significantly more complex than any of the other examples — with both classes requiring *five* ownership and type parameters — so complex, arguably, that it would be unusable in practice.

3. GENERIC OWNERSHIP

Generic Ownership is a new linguistic mechanism that successfully combines genericity and ownership into a single simple language. As in the hypothetical example of figure

```
public class SimpleMap<mapOwner>[Key<keyOwner> extends Comparable<keyOwner>, Value<valueOwner>] {
  private Vector<this>[MapNodes<this>[Key<keyOwner>, Value<valueOwner>]] mapNodes;

  void put(Key<keyOwner> key, Value<valueOwner> value) {
    mapNodes.add(new MapNode<this>[Key<keyOwner>, Value<valueOwner>](key, value));
  }

  Value<valueOwner> get(Key<keyOwner> key) {
    Iterator<this>[MapNodes<this>[Key<keyOwner>, Value<valueOwner>]] i = mapNodes.iterator();
    while(i.hasNext()) {
      MapNode<this>[Key<keyOwner>, Value<valueOwner>] mn = i.next();
      if (mn.key.equals(k)) {
        return mn.value;
      }
    }
    return null;
  }
}

class MapNode<mapNodeOwner>[Key<keyOwner> extends Comparable<keyOwner>, Value<valueOwner>] {
  public Key<keyOwner> key;
  public Value<valueOwner> value;
  MapNode(Key<keyOwner> key, Value<valueOwner> value) {
    this.key = key; this.value = value;
  }
}
```

Figure 4: A combined generic and ownership types implementation of a `SimpleMap` class (after Boyapati [10])

4, Generic Ownership provides the benefits of both type and ownership parameterisation: catching all the errors and avoiding all the bugs that the generic and ownership languages do individually. Unlike that example, Generic Ownership treats ownership and genericity as one single aspect of language design, and so code using Generic Ownership is no more complex than code that is either type-parametric or ownership-parametric.

The key technical contribution of Generic Ownership is that it treats ownership as an additional kind of generic type information. This means that existing generic type systems can be extended to carry ownership information with only minimal changes. By exploiting a defaulting mechanism which permits ownership information to be omitted in common circumstances, Generic Ownership programs are almost indistinguishable from their ownership-free counterparts, while gaining the benefits of ownership.

In this section, we present Generic Ownership in terms of a language that we have implemented called Ownership Generic Java (OGJ), a seamless extension to the Java Generics (GJ) proposal [42]. In section 4, we will outline the formal basis of our work.

3.1 Ownership Generic Java

Figure 5 revisits our `SimpleMap` example for the last time. This time, it is written in Ownership Generic Java. Comparing figure 5 with figures 1–4 shows that it is slightly more complex than the individual type genericity or ownership examples, but much simpler than the straightforward combination in figure 4. Note that the code in figure 5 is type-generic: definitions of fields in `MapNode` and methods everywhere use generic types such as `Key` and `Value` rather than plain class types such as `Object` or `Comparable`. But this code also supports ownership: each class has an extra generic parameter called `Owner` which represents its owner. As with other ownership type systems, it is not possible for owned objects to be exposed.

As you can see from the figure, OGJ is a language that is completely syntactically compatible with GJ — any valid OGJ program is also a valid GJ program. OGJ provides ownership by requiring every single class declaration to have at least one type parameter that is declared as a subclass of `World`. This parameter should always come last and we call it the *owner parameter*. For example, instead of declaring:

```
public class Foo { ... }
```

OGJ requires the following declaration:

```
public class Foo<Owner extends World> { ... }
```

Owner parameters are used to record the ownership of individual objects of a particular class. OGJ provides four *ownership domains* that programmers can use to instantiate owner parameters: `World`, `Package`, `Class`, and `This`. These domains represent different ownership scopes to which objects can belong, and were chosen to parallel the existing static access structures in Java. Any instance of a class whose owner parameter is instantiated with `World` can be referred to by any other object. References to an object instantiated with a `Package` owner are confined within the package where the instance's type is instantiated, and an instance marked with `Class` is confined within that class. Finally, an object marked with `This` is encapsulated within the object where the type containing `This` is instantiated — that is, it is owned directly by that object. In this way,

```
public class SimpleMap<Key extends Comparable,
                       Value,
                       Owner extends World> {
  private Vector<MapNode<Key, Value, This>,
                 This> mapNodes;

  public void put(Key key, Value value) {
    mapNodes.add(new MapNode<Key, Value,
                             This>(key, value));
  }

  public Value get(Key key) {
    Iterator<MapNode<Key, Value, This>, This> i
      = mapNodes.iterator();
    while (i.hasNext()) {
      MapNode<Key, Value, This> mn = i.next();
        if (mn.key.equals(key)) {
          return mn.value;
        }
      }
    return null;
  }
}

class MapNode<Key extends Comparable, Value,
              Owner extends World> {
  public Key key;
  public Value value;
  MapNode(Key key, Value value) {
    this.key = key;
    this.value = value;
  }
}
```

Figure 5: Generic Ownership implementation of a `SimpleMap` class

OGJ provides both confined types (`Package` and `Class`) and shallow ownership (`This`).

For example, if a `Foo` instance is created with its ownership parameter bound to `Package`, as in:

```
public Foo<Package> f = new Foo<Package>();
```

then the new instance of `Foo` can only be accessed via the package containing the instantiation. Similarly, if it is created using `This` ownership:

```
public Foo<This> f = new Foo<This>();
```

then it can only be accessed via the current instance of the class creating the new `Foo` object.

Consider an example exposing a `mapNodes` vector private to the `SimpleMap` object considered earlier in section 2.2:

```
public Vector exposeVector() {return mapNodes;}
```

This code is not going to be valid in OGJ as is, since every type has to carry an owner parameter (and casting to raw types [27] is prohibited). The type of the field `mapNodes` in figure 5 has an owner parameter `This`. If we try to give a return type of `exposeVector` method an owner parameter `This`, e.g.:

```
public Vector<MapNode<Key, Value, This>, This> exposeVect
  return mapNodes;
}
```

then this code will be valid OGJ, but the method can only be called if the result can be assigned to something that is a supertype of `Vector<..., This>`. Since OGJ preserves owners over subtyping, (see section 4) any valid supertype of the return type will have to have an owner `This`, which will only typecheck if it is declared in *the same instance* of `SimpleMap` class. In other words, this `exposeVector` method *cannot* expose the vector.

If we attempt to declare `exposeVector` with a return type having any other owner parameter, then OGJ will not compile the method since the return type and the return value's (`mapNodes`) type will not be assignment compatible due to owner parameters being different.

Classes can also be declared so that their owner is bounded by a more specific domain, rather than `World`. For example, if a `ConfinedFoo` class is declared:

```
public class ConfinedFoo<Owner extends Package> {
  ...
}
```

then *all* instances of class `ConfinedFoo` must be created inside the `Package` scope (e.g. `new Foo<Package>`); creating them in a wider scope (e.g. `new Foo<World>`) is disallowed. This provides very similar encapsulation to Confined Types [9].

The OGJ domains form a hierarchy, with `World` at the top, and having the widest scope, down through `Package`, `Class`, and finally to `This`, so any class can be instantiated with an owner whose scope is inside its declaration's bound. While `ConfinedFoo` can be instantiated with `Package` ownership, it could also be instantiated with `Class` or `This` ownership, provided the class or instance instantiating `ConfinedFoo` is within the package within which it is confined.

3.2 Ownership Defaulting

To further reduce the syntactic overhead of Generic Ownership, owner parameters can be elided from class declarations and instantiations.

Essentially, if a class declaration does not declare an owner parameter, the OGJ language will provide a default `Owner` extending `World`. We choose `World` so that ordinary Java and GJ code can be used unchanged in OGJ. For example, the following class declaration:

```
public class Athlete<Event> { ... }
```

when compiled using OGJ will be treated as:

```
public class Athlete<Event, Owner extends World> {
  ...
}
```

Similarly, if a class has an owner parameter in its declaration (possibly added by defaulting), this parameter may be omitted and OGJ will instantiate the parameter to its bound by default. For example, code using `Athlete`, such as:

```
Athlete<Discus> a = new Athlete<Discus>();
```

will be taken to mean:

```
Athlete<Discus<World>, World> a =
  new Athlete<Discus<World>, World>();
```

The main effect of defaulting is that programmers are able to write Generic Ownership code with very little syntactic overhead, providing owner parameters only when absolutely required. Because the rules for defaulting are quite straightforward, they have no effect on the modularity of OGJ code — as with defaults in other generic type systems, but in contrast to more complex type inference schemes used to support other ownership types systems [4, 23].

We have implemented this defaulting scheme within our OGJ compiler and can successfully check GJ's *collect.jar*, and compile the compiler source itself.

3.3 Implementing OGJ

OGJ is implemented as an extension to the Generic Java compiler. Implementing OGJ required two main additions to the compiler — first, to support ownership domains, and second, to enforce the constraints of ownership types.

Within the OGJ compiler, ownership domains are reified to hidden, package-private classes we call *owner classes*. The OGJ compiler creates owner classes automatically, and disposes of them once type checking is complete, so programmers are never aware of their existence. The owner class corresponding to the `World` domain is at the top of the hierarchy, and every other owner class inherits from it. Every package, class, and every instance have a corresponding owner class denoted by `Package`, `Class`, and `This` respectively.

To implement confinement, the OGJ compiler replaces every occurrence of an ownership domain name (`World`, `Package`, `Class`, and `This`) with the appropriate owner class — `Package`, `Class`, and `This` are replaced by different classes depending on the context in which they occur. Because the world, package, and class domains are static this is essentially all that is required to implement ownership in those domains: GJ's type soundness ensures that classes that have different ownership (are parameterized by different owner classes) will not be assignment compatible.

Per-instance object ownership via the `This` domain requires slightly more effort. First, we replace the `This` domain with the appropriate owner class. Then, we ensure that only expressions referring to the current instance (either implicitly via direct calls to methods or fields, or explicitly via the "`this`" keyword) can access types whose owner is a `This` domain owner class.

Finally, we have to ensure that ownership information cannot be lost. For this reason, we only allow type casts that preserve ownership. Types with ownership parameters cannot be cast to types that do not have ownership parameters (such as `java.lang.Object`) or to types with different ownership parameterisation. We also prevent casts to raw types [27, 42, 15] when such casts would delete an ownership parameter.

3.4 Compatibility with (Generic) Java

OGJ code is completely interoperable with Java code: OGJ code can call Java code (generic or otherwise) and vice versa. For the sake of complete forwards compatibility, we provide four blank interfaces (`World`, `Package`, `Class`, and `This`) that allow a GJ compiler to compile OGJ code by effectively ignoring the owner parameters. Without defaulting, OGJ also provides full backwards compatibility, as classes without any type parameters, or with no owner parameters, are treated as Java and GJ classes respectively.

Ownership defaulting, unfortunately, makes the situation more complex: there are certain GJ classes that are not valid in OGJ. (This is not surprising — if every GJ class were valid in OGJ, then OGJ could not be enforcing object ownership.) Consider the following class:

```
class Foo {
    static Foo singleton;
    public Foo() {
        Foo.singleton = this;
    }
}
```

where ownership defaulting will add `<Owner extends World>` to the class declaration, and then default the type of the `singleton` static variable to `Foo<World>`. The code inside the constructor will then fail to type check, because `this` will have the type `Foo<Owner>`, which is not assignment compatible with `Foo<World>`. In general, GJ classes will not compile under OGJ in cases where their code is not ownership parametric: that is, where they break a confinement invariant — in the example, `this` is leaked out from its owner to the `singleton` static variable. Although expressed differently, this restriction is essentially the same as the anonymity conditions required by Confined Types [9, 45]; other ownership schemes enforce similar constraints relying on owner parameterized classes [16, 17, 4, 13].

If we rewrote the code above as follows:

```
class Foo extends Object<World> {
    static Foo singleton;
    public Foo() {
        Foo.singleton = this;
    }
}
```

then the code could compile using *manifest ownership* [16] — that is, ownership of all the instances is no longer parametric, but rather is specified by instantiating the superclass's ownership parameter explicitly. In this case, `Foo extends Object<World>` would mean that all instances of this class would have owner `World`. Classes using manifest ownership need not be subject to ownership defaulting because neither declarations nor instantiations require an ownership parameter.

4. FORMALISATION OF OGJ

In this section we present an overview of our initial formalisation of OGJ, called FGJ+c; a more complete account can be found elsewhere [40]. FGJ+c embodies a minimalist confinement scheme which leverages parametrically polymorphic types to enforce static confinement, that is, for a statically known collection of protection domains. Our formalism is based on an extension to Featherweight Generic Java (FGJ), which, with Featherweight Java (FJ), provides simple models of Java and Generic Java's type systems [26]. Our compiler also implements *shallow ownership* and *defaulting*, though we have not treated this in this formalism.

As we have explained, the key idea behind Generic Ownership is to use generic type parameters to carry ownership

Syntax:

```
T ::= X  |  N
N ::= C<T̄,O>
L ::= class C<X̄ ◁ N̄, Owner ◁ Domain> ◁ N {T̄ f̄; K M̄}
K ::= C(T̄ f̄) { super(f̄); this.f̄=f̄; }
M ::= <X̄ ◁ N̄> T m(T̄ x̄) { return e; }
e ::= x | e.f | e.m<T̄>(ē) | new N(ē) | (N̄)e
```

For presentation reasons, we reserve `O` to refer to owner-classes, `Domain` for concrete owner-classes, `Owner` for owner variables. (Think `O ::= Domain | Owner`.) From the FGJ standpoint, `O` is no different from `X` or `N` where the latter is restricted to the owner classes as shown in figure 8.

Figure 6: FGJ syntax, adapted from Igarashi et al

information as well as type information. Following the traditional approach of Ownership Types [18] we require every FGJ+c class to have at least one type parameter to carry this ownership information. We use the *last* type parameter to record an object's owner. All FGJ+c classes descend from a new class `CObject` (for confinable object) that has just one parameter called `Owner`; all its subclasses must invariantly preserve this parameter to represent their owner. We are careful to ensure that OGJ programs remain FGJ programs, though they must satisfy certain additional restrictions. The consequence is that we can leverage FGJ's type soundness result, and use it when establishing the desired confinement property. The syntax of FGJ from Igarashi et al. [26], adapted only slightly for presentation reasons, is given in figure 6. Note that we do not account for statics, which are not present in FGJ. We have thus omitted manifest ownership from our formalism.

Every OGJ program must satisfy the FGJ type rules [26], along with some additional constraints which are presented in figure 7. The rules deal with three concerns: firstly, they ensure that owners are preserved within types, as the owner contains information to determine the visibility of a type; secondly, they determine which types are visible in a given context (class); and finally, they propagate and check the desired constraints for fields and methods.

4.1 Owner Preservation

The rules for classes (FGJ+c-Class) and types (FGJ+c-Type) combine together to ensure that every type has an owner (or that its bound does, in the case of type variables) *and* that this owner is preserved through subtyping. The class rule, among other things, ensures that the owner of a class is the same as the owner of its superclass. In addition, we ensure that the only types used in programs are a subtype of `CObject<O>` for some owner-class `O` and thus also prevent owner-classes from being instantiated. We also get the restriction that only owner-classes can be used as owners.

Figure 8 shows the FGJ+c class hierarchy. FGJ classes descend from `Object` and FGJ+c classes (denoted using white boxes) descend from `CObject<O>` class just below `Object`. Owner classes descend from `World` and form a separate hierarchy of FGJ-valid but not FGJ+c-valid classes.

4.2 Visibility

The visibility rules form the foundation of FGJ+c additional rules, they determine which owners, types, and terms are visible, and thus usable, within a given class. There are three sets of rules: those that determine *owner* visibility, *type* visibility, and *term* visibility.

The *owner* visibility rules simply check that a given owner corresponds to the class's package (e.g. $\pi_{\mathtt{D}}$ inside class `D`) or is a public owner denoted `World`. The *type* visibility rules allow the use of types when their owner parameter is from the current package, is the current instance's owner, or is obtained via one of the type parameters — roughly following Zhao et.al. [45].

Term visibility rules are defined inductively on the structure of FGJ terms. For each subexpression, they determine whether the type of that subexpression is visible according to the type visibility rules.

Consider the following code example, where `p1c` refers to an owner class marking instances confined to package `p1`.

```
p1.List<p2.Foo<p3c>, p4c>
```

This describes a list declared in package `p1` storing items declared in package `p2` that are confined to package `p3`, while list itself is confined to package `p4`. This list is allowed access to classes confined to `p1` (since the code performing the access is already inside package `p1`), `p4` (since the list instance is confined in that package during the execution) and to instances confined to package `p3` because one of the list's type parameters is confined in that package (i.e. is owned by `p4c`). Any classes confined to `p2` or any other package cannot be accessed inside the list.

4.3 Propagation of constraints in classes and methods

Visibility constraints are propagated through classes to their fields and methods and eventually to expressions. These rules say that *any* type appearing in the program, even as a subexpression, *must* be visible in the present class. For classes (FGJ+c-Class), we check that all the field types are visible and that the bounds on the type variables are also visible to the present class. For methods (FGJ+c-Method), we check that the argument and return types are visible, that the method body satisfies the visibility constraint, that all subexpressions use visible types, and again that the type variables to the method have bounds which are visible.

Note that we have used visibility checks throughout the rules, rather than build the appropriate checks into the well-formed type rule, so that we can reuse all of the FGJ rules (and proofs).

These rules guarantee that FGJ+c programs don't break confinement, allowing us to prove a confinement invariant.

4.4 Confinement Invariant

The confinement invariant for well-typed FGJ+c programs shows that types that are not visible within the current package are not reachable. We assume that we only deal with FGJ+c programs: that is FGJ programs where all the classes are `FGJ+c` as given by FGJ+c-Type rule in figure 7. We prove that during execution, we cannot get to an instance of a class that is not *visible* within the current context. This result relies on the fact that the owner parameter is preserved in the class hierarchy. Let $owner_{\Delta}(\mathtt{T})$ be `O` for

FGJ+c Types:

$$\frac{\Delta \vdash \texttt{T} \texttt{<:} \texttt{CObject<O>} \qquad \Delta \vdash \texttt{O} \texttt{<:} \texttt{World}}{\Delta \vdash \texttt{T OK+c}} \quad \text{(FGJ+c-Type)}$$

Owner Visibility:

$$\frac{}{\Delta \vdash \mathit{visibleowner}(\pi_{\texttt{D}},\ \texttt{D})}\ ^{\dagger} \qquad \frac{}{\Delta \vdash \mathit{visibleowner}(\texttt{World},\ \texttt{D})} \quad \text{(V-Owner)}$$

$$\frac{\Delta \vdash \mathit{visibleowner}(\mathit{bound}_{\Delta}(\texttt{Owner}),\ \texttt{D})}{\Delta \vdash \mathit{visibleowner}(\texttt{Owner},\ \texttt{D})}$$

Type Visibility:

$$\frac{\Delta \vdash \texttt{T OK+c} \qquad \texttt{N} = \mathit{bound}_{\Delta}(\texttt{T}) \qquad \texttt{N} = \texttt{C<}\overline{\texttt{T}},\ \texttt{O>}}{\Delta \vdash \mathit{owner}(\texttt{T},\ \texttt{O})}$$

$$\frac{\Delta \vdash \mathit{owner}(\texttt{T},\ \texttt{O}) \qquad \Delta \vdash \mathit{visibleowner}(\texttt{O},\ \texttt{D})}{\Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D<}\overline{\texttt{T}_{\texttt{D}}},\ \texttt{O}_{\texttt{D}}\texttt{>})}$$

$$\frac{\exists \texttt{T}' \in \overline{\texttt{T}_{\texttt{D}}} : \mathit{owner}(\texttt{T},\ \texttt{O}) \wedge \mathit{owner}(\texttt{T}',\ \texttt{O})}{\Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D<}\overline{\texttt{T}_{\texttt{D}}},\ \texttt{O}_{\texttt{D}}\texttt{>})}$$

$$\frac{\mathit{owner}(\texttt{T},\ \texttt{O}_{\texttt{D}})}{\Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D<}\overline{\texttt{T}_{\texttt{D}}},\ \texttt{O}_{\texttt{D}}\texttt{>})} \quad \text{(V-Type)}$$

Term Visibility:

$$\frac{\Delta;\Gamma \vdash \texttt{x} : \texttt{T} \qquad \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D})}{\Delta;\Gamma \vdash \mathit{visible}(\texttt{x},\ \texttt{D})} \quad \text{(V-Var)}$$

$$\frac{\Delta;\Gamma \vdash \mathit{visible}(\texttt{e},\ \texttt{D}) \qquad \Delta;\Gamma \vdash \texttt{e.f}_{\texttt{i}} : \texttt{T} \qquad \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D})}{\Delta;\Gamma \vdash \mathit{visible}(\texttt{e.f}_{\texttt{i}},\ \texttt{D})} \quad \text{(V-Field)}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e.m}(\overline{\texttt{e}}) : \texttt{T} \qquad \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D}) \qquad \Delta;\Gamma \vdash \mathit{visible}(\texttt{e},\ \texttt{D}) \qquad \Delta;\Gamma \vdash \mathit{visible}(\overline{\texttt{e}},\ \texttt{D})}{\Delta;\Gamma \vdash \mathit{visible}(\texttt{e.m}(\overline{\texttt{e}}),\ \texttt{D})} \quad \text{(V-Invk)}$$

$$\frac{\Delta;\Gamma \vdash \mathit{visible}(\overline{\texttt{e}},\ \texttt{D}) \qquad \Delta \vdash \mathit{visible}(\texttt{N},\ \texttt{D})}{\Delta;\Gamma \vdash \mathit{visible}(\texttt{new N}(\overline{\texttt{e}}),\ \texttt{D})} \quad \text{(V-New)}$$

$$\frac{\Delta;\Gamma \vdash \mathit{visible}(\texttt{e},\ \texttt{D}) \qquad \Delta \vdash \mathit{visible}(\texttt{N},\ \texttt{D})}{\Delta;\Gamma \vdash \mathit{visible}((\texttt{N})\ \texttt{e},\ \texttt{D})} \quad \text{(V-Cast)}$$

FGJ+c Methods:

$$\frac{\begin{array}{c}\Delta \vdash \overline{\texttt{T}}, \texttt{T OK+c} \qquad \Delta \vdash \mathit{visible}(\overline{\texttt{T}},\ \texttt{C}) \\ \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{C}) \qquad \Delta \vdash \mathit{visible}(\mathit{bound}_{\Delta}(\overline{\texttt{P}}),\ \texttt{C}) \\ \Delta;\ \overline{\texttt{x}} : \overline{\texttt{T}},\ \texttt{this} : \texttt{C<}\overline{\texttt{X}},\ \texttt{Owner>} \vdash \mathit{visible}(\texttt{e}_0,\ \texttt{C})\end{array}}{\texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\ \texttt{return e}_0;\ \}\ \texttt{FGJ+c IN C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}},\ \texttt{Owner>}} \quad \text{(FGJ+c-Method)}$$

FGJ+c Classes:

$$\frac{\begin{array}{c}\overline{\texttt{X}} \texttt{<:} \overline{\texttt{N}} \vdash \texttt{N}, \overline{\texttt{T}}\ \texttt{OK+c} \qquad \Delta \vdash \mathit{visible}(\mathit{bound}_{\Delta}(\overline{\texttt{N}}),\ \texttt{C}) \\ \Delta \vdash \mathit{visibleowner}(\texttt{Domain},\ \texttt{C}) \qquad \overline{\texttt{M}}\ \texttt{FGJ+c IN C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}},\ \texttt{Owner} \lhd \texttt{Domain>} \\ \Delta \vdash \mathit{visible}(\overline{\texttt{T}},\ \texttt{C}) \qquad \texttt{N} = \texttt{C}'\texttt{<}\overline{\texttt{T}'},\ \texttt{Owner>}\end{array}}{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}},\ \texttt{Owner} \lhd \texttt{Domain>} \lhd \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{FGJ+c}} \quad \text{(FGJ+c-Class)}$$

† $\pi_{\texttt{D}}$ is the owner-class corresponding to the package to which D belongs.

Figure 7: FGJ+c Additional Rules

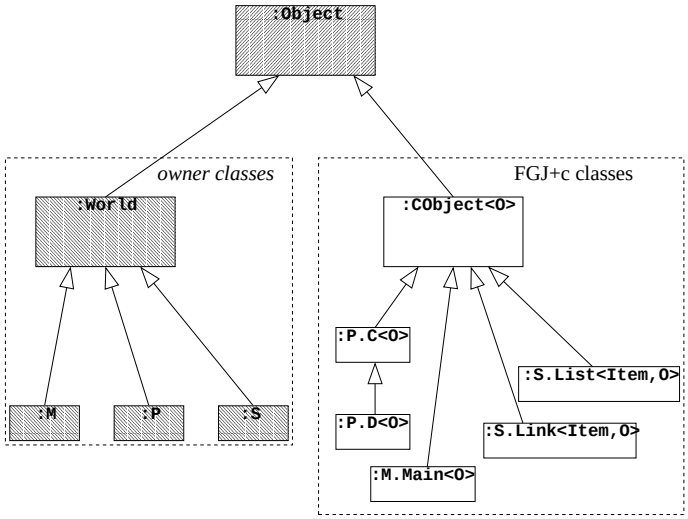

Figure 8: FGJ+c Class Hierarchy

types `C<`$\overline{\texttt{T}}$`,O>` and the owner of the bound in Δ for type variables. Then:

Lemma (Ownership Invariance over Subtyping). *If* $\Delta \vdash$ `S <: T` *and* $\Delta \vdash$ `T <: CObject<O>`*, then* $owner_\Delta$(`S`) = $owner_\Delta$(`T`) = `O`.

Proof. By induction on subtypes below `CObject`. `S` and `T` must eventually be bound to classes as these are the only ground types in FGJ.

Base case: they are bound to `CObject` so this is trivially true.

Inductive case: By FGJ subtyping they must be bound to some transitive subclass of `CObject`.

By FGJ+C-Class a FGJ+c class has the same owner parameter as its superclass. □

Theorem (Confinement Invariant). *Given any subexpression* `e` *of an expression* `d` *in a method* `m` *of class* `C` *which is* `FGJ+c`*: If* `e`$\rightarrow^*$`new D<`$\overline{\texttt{T}_\texttt{D}}$`, O>(`$\overline{\texttt{e}}$`)`*, then visible*(`D<`$\overline{\texttt{T}_\texttt{D}}$`, O>, C`).

Proof. By FGJ subject reduction: `e : T`$\rightarrow^*$`e'` : `T'` $\Rightarrow$ `T' <: T`.

Since *visible*(`d`, `C`) by FGJ+C-Method, by visibility rules we also get *visible*(`e`, `C`).

From the *owner* clause of V-Type rules `T` is `OK+c` and thus by FGJ+C-Type `T` = `B<`$\overline{\texttt{N}'}$`, Owner>` where `B <: CObject<Owner>`.

By the ownership invariance over subtyping lemma: `T'` = `A<`$\overline{\texttt{N}}$`, Owner>` where `Owner` is the same in all three.

But to get *visible*(`B<`$\overline{\texttt{N}'}$`, Owner>, C`) we must have *visible-owner*(`Owner`, `C`). By FGJ+C-Type, `A<`$\overline{\texttt{N}}$`, Owner>` will be `OK+c`.

Then by V-Type: *visible*(`A<`$\overline{\texttt{N}}$`, Owner>, C`). □

4.5 Generic Confinement

Generic confinement raises two issues that were also discussed by Zhao et.al. [45]. First, as with ownership type systems [16], instantiating a class with an actual owner parameter can be seen as giving the instances of that class permission to access other objects owned by the actual owner parameter — in our case, confined within the package corresponding to the actual owner parameter. This is made explicit in the 3rd type visibility rule, which explicitly checks these permissions, that is, which ensures a type is visible if its owner is the owner of any of the (actual) type parameters.

Second, if we consider the following FGJ+c expression evaluated using the FGJ rules:

```
class Foo {
 ...
 String<Class> m() {
  return (new SimpleMap<Integer<World>,
            String<Class>,
            Package>).get(new Integer(42));
 }
}
```

then we can see that although this code could be located *anywhere* (in this case, inside method `m` of an unrelated class `Foo` in an unrelated package), *inside* the evaluation of the `get` method objects private to the `SimpleMap`'s package can be accessed (as can objects owned by `World`, by the `Foo` class, or the `Foo` class's package) and can appear as intermediate results as the expression is evaluated. This breaks neither our confinement invariant nor that of Zhao et.al. [45] since the only objects that the `Foo` instance executing method `m` can access *directly* are the final results of evaluating whole subexpressions, such as the constructor call or the `get` invocation. The `SimpleMap` constructor or `get` method may well create or reference other objects that are private to the `SimpleMap`, but `Foo` itself will not have permission to access these other objects directly, and our system prevents such accesses.

4.6 Ownership

Object ownership works at a finer granularity than confinement: ownership allows objects to be encapsulated within a *dynamic* protection domain, typically another object, whereas

confinement is limited to *static* protection domains, such as Java packages. Recall that OGJ has the domain "`This`" to model objects owned by the current object (i.e., owned by "`this`"). The constraints on such objects are taken from the original ownership type system [18], stating that we only access such objects through "`this`". The rule follows, where *owners* finds the set of all ownership parameters or bounds in a type:

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{This} \in \mathit{owners}(\mathit{mtype}(\mathtt{m}, \mathit{bound}_{\Delta}(\mathtt{T}))) \Rightarrow \mathtt{e} \equiv \mathtt{this}}{\Delta;\Gamma \vdash \mathit{ownership}(\mathtt{e.m}(\bar{\mathtt{e}}),\ \mathtt{D})}$$

While we can state this rule within FGJ, the functional substrate of FGJ is insufficient to support a proof of an ownership invariant. We are currently working on a full proof. Existing results for imperative, though not generic, Java-like calculi [17] and for second-order polymorphic object calculi [16] will guide our way. Providing deep ownership will require the type system to keep track of the object containment relationships [17]. Doing so, however, will require a careful analysis of the interaction between the different varieties of confinement OGJ offers, and remains an open problem.

4.7 Defaulting

The formal treatment of defaulting requires two expansions that take FGJ code with some owner parameters lacking and add the required parameters. These expansions are applied to class declarations and instantiations. To expand a class declaration of the form:

```
class C<typeargs> extends T { ... }
```

we perform the following steps, if the last argument of `typeargs` is not an owner parameter:

- Expand superclass `T` and determine its owner parameter, recurse by defaulting its class declaration too if necessary. `CObject` (root of the FGJ+c class hierarchy) has an owner parameter.
- Once we obtain the bound of the superclass's owner parameter `Domain` (which can be `World`) we add an owner parameter to `typeargs` of the form `Owner extends Domain`.

To expand a type which doesn't instantiate its owner parameter (which could have been added to the class's declaration using the defaulting mechanism described above), we look up the class declaration and instantiate the owner to the bound used in the declaration. As discussed earlier, class:

```
class Foo<Owner extends Package> { ... }
```

when used without an owner parameter:

```
Foo f = new Foo();
```

will be defaulted to:

```
Foo<Package> f = new Foo<Package>();
```

where `Package` will correspond to the one used in its declaration.

The OGJ compiler supports defaulting: we are currently developing a full formal framework that will incorporate defaulting (and ownership, as described in the previous subsection).

5. DISCUSSION AND RELATED WORK

Object encapsulation has been recognised as a means for addressing aliasing, security, concurrency, and memory management problems, with the merit of smoothly aligning with the way many object-oriented programs are designed. Two complementary threads of research have evolved. On one hand are expressive but weighty *explicit* type systems based on ownership types [18]. On the other hand are lightweight but limited *implicit* systems based on confined types [9].

The explicit systems based on ownership types, including AliasJava, Universes, and the system of Boyapati et.al. [4, 3, 13, 33], differ essentially in only one characteristic, which Clarke and Wrigstad distinguish as *shallow* vs *deep* ownership [20]. A deep ownership type permits only a single object as entry point to the collection of objects it owns, whereas a shallow ownership type permits multiple entry points into the confined collection. Clarke and Drossopoulou [17] and Boyapati et.al. [13] describe how to exploit the useful properties of deep ownership. Generic Ownership falls within the category of explicit systems, as it uses type annotations to provide ownership: however, the key contribution of Generic Ownership is that it is the first system to combine type genericity and object ownership into a single system. Unlike our previous work [37, 16], Generic Ownership uses shallow ownership. There are several reasons for this: compatibility with Generic Java; better fit with other static ownership schemes; and finally the simplicity of the approach. We plan to extend OGJ to support a form of deep ownership. Combining deep ownership (e.g. Clarke [16]) and static (generic) confinement (e.g. Confined Types [45]) in a single language is currently an open problem.

Unlike some other ownership type schemes [12] OGJ does not currently support runtime downcasts, thus the rule forbidding casts that would lose ownership information. This is primarily for compatibility with existing Java and GJ programs: safe ownership downcasts require runtime information which existing compilers do not supply nor existing libraries expect. Also, we suspect that other Ownership Type systems require many of these downcasts because they are not type-generic: as with GJ, OGJ's genericity should remove the need for many of these downcasts.

The disadvantage of all these explicit systems is that ownership types require additional annotations to use them, raising issues about their role in programming. For this reason, AliasJava and Boyapati et.al. have described a range of type inference schemes to provide these annotations automatically [4, 14, 11, 12]. OGJ addresses the syntactic overhead via very straightforward ownership defaulting, rather than depending upon type inference algorithms. Again, the combination of genericity and ownership enables a simpler approach: many of the owner parameters that would have to be inferred by other schemes will already be present in type-generic code. We expect the defaulting rules to be considerably easier for mainstream programmers to deal with.

Also, one minor (but still important) language design feature makes defaulting in OGJ considerably easier than in the other systems: in OGJ, an object's primary ownership is carried by the *last* parameter, whereas in most other systems (following Clarke et al. [18]) it is carried by the *first*. Having ownership in the last position means that owners can be defaulted simply by leaving them off (e.g. "`Foo`") while Boyapati et.al., for example, require a placeholder (e.g. "`Foo<->`") which is referred to as an *anonymous owner*.

OGJ's defaulting is much weaker than the mechanisms provided by inference schemes [4, 10] since we only use *one* owner parameter and default it to `World`. Inferring only one owner parameter is less restrictive than it may seem, as all generic parameters can carry ownership information in OGJ. On the other hand, defaulting undeclared owners to `World` provides no encapsulation. However, type inference schemes can only infer types to describe the implicit encapsulation structures latent in the code. They do not enforce any particular encapsulation discipline: rather they will infer the equivalent of `World` ownership for any type which escapes the unit of analysis. This means that neither inference nor our defaulting can enforce encapsulation guarantees that the programmer has not specified, rather that inference will attempt to accurately identify latent encapsulation where defaulting currently will not.

We consider that the single defaulting rule is more appropriate for a general-purpose language such as OGJ — it is certainly simpler to explain and to understand. However, we plan to pursue ownership inference for OGJ programs as a programming tool outside the language, to support programmers adding explicit ownership declarations to programs to reflect their intentions. We also plan to use the metadata facilities available in the upcoming Java release [42] to investigate combining the last owner parameter with the existing access protection syntax (`public`, `protected`, `private`), allowing, for example, a programmer to mark a field as `@private` to mean both `private` and owner parameter `This` simultaneously.

Implicit confined type systems have achieved their more limited goals while keeping the amount of annotations low. Vitek and Bokowski's original system [9], which had security as its application, required certain classes to be annotated as confined, to indicate classes confined within the present package, and certain methods to be annotated as anonymous, to indicate that such methods do not reveal "`this`". Grothoff, Palsberg, and Vitek [23] show how type inference can be used to avoid the need for annotation, making a system that can provide per-package encapsulation in practical programs. Clarke, Richmond and Noble [19] apply these ideas in the context of Enterprise Java Beans, and by exploiting special architecture specific constraints, provide per-object encapsulation without annotations or inference.

More recent work by Zhao, Palsberg and Vitek [45] has formalised Vitek and Bokowski's approach to per-package confinement, with an operational semantics and a static type system called Confined Featherweight Java (CFJ) based on Featherweight Java, augmented by a number of specific rules that restrict programs. CFJ also proposes a notion of generic confined types, allowing, for example, a collection to be confined or not, depending upon the specifications of the contained elements, that is supported by further development of the static type system.

OGJ's approach is essentially the opposite. Rather than starting from a language without generic types, and then adding a special form of genericity to support confinement, we start from a language with generic types (GJ and its formal core FGJ) and then ensure confinement, and ultimately ownership, directly. Our approach has led to a simpler formal system, requiring fewer new concepts, and a distinctly simpler and shorter proof for class or package confinement [40]. We do not need to distinguish anonymous methods, because "`this`" is parameterized to record its ownership.

Banerjee and Naumann prove a per-object representation independence result for Java [6, 7]. They adopt a confinement discipline resembling ownership types, except that they apply the confinement only at the point they wish to reason about. They require that confined classes extend a special class called Rep, and that the boundary classes extend a special class called Own. Neither Rep nor Own can be forgotten from a type. Ultimately, their results say that confinement and ownership matter for deep reasoning about objects.

Clarke's thesis was the first account of a system with both parametric polymorphism and ownership [16]. This system was based on Abadi and Cardelli's object calculus [1], rather than a generic class-based language. Clarke, however, gives an encoding of a class-based language into his formalism. He further discusses how ownership can be combined with a generic class-based language (with inner classes) and a primitive form of defaulting. OGJ pushes this work to a practical level, exploiting the generic mechanism of Generic Java, rather than by requiring a separate syntactic category of owners, and employing a more advanced defaulting mechanism: all supported by a working compiler.

A bit further afield, we find that the implementation of the State Monad in Haskell [30] adopts mechanisms similar to Generic Ownership. In the State Monad, a type variable is assigned to the encapsulated state, and an appropriate hiding of the type (via rank-2 polymorphism) ensures that the state doesn't escape and thus behaves correctly. The Haskell design is more straightforward than OGJ, as they have the luxury of ignoring features used in mainstream object-oriented languages — imperative state, subtyping, classes, objects, and so on. Interestingly, this design resembles an encoding of existential types in terms of universal types, while Clarke's thesis formalises the confinement provided by ownership types as existential over owners [16].

Even more generally, types are a recognised tool for managing and reasoning about aliasing, being both useful for characterising when aliasing may be present, when it definitely is be present, when a particular piece of code does not introduce aliases, and so forth. Type-based alias analysis uses the class hierarchy to discover non-alias conditions [21], though the space of types considered is restricted to ordinary program types. Other systems use type inference à la Hindley-Milner to facilitate finer distinctions on potential aliasing [43, 44, 5, 38, 29]. These systems, along with ownership types, use an extra type dimension to express distinctions which are not possible with the original program types. Functional programmers use a similar technique called "phantom types" [22] to introduce more type distinctions to make safer programs or encode information which is not present in ordinary program values. Generally, this extra type dimension plays no role in computation, but it can be used to expose or enforce constraints on aliasing beyond what is possible without it. Generic Ownership uses generics to carry the extra type dimension which, when restricted to owner classes and subject to the corresponding visibility constraints, enables object encapsulation. Along the way, we gain the benefits of genericity, while paying a minimal syntactic cost.

6. CONCLUSION

In this paper, we introduce Generic Ownership, a single system that encompasses both generic types and object own-

ership. Our design and formalism show that ownership and generic type information can be expressed within the same system, and carried around the program as bindings to the same parameters. To reduce the syntactic impact of ownership types for programmers, we have developed ownership defaulting, allowing owner parameters to be omitted from type declarations and type instantiations. As a result, programs using Generic Ownership are slightly more complex than those using only generic types, yet they enjoy the full protection provided by ownership types. We have demonstrated the practicability of Generic Ownership by designing the language OGJ, a seamless, syntactically compatible extension to GJ, and we have proved the soundness of a significant subset of OGJ.

To summarise, our contributions are as follows: Generic Ownership, a seamless integration of genericity and ownership; Ownership Generic Java language implementation providing generics, ownership, and confinement; a formal model allowing the proof of static confinement invariant for OGJ; and owner defaulting, enabling the programmer to omit common ownership annotations.

Acknowledgments

This work is supported in part by the Royal Society of New Zealand Marsden Fund. The second author would like to thank the staff of Ward 18, Wellington Hospital. Finally, we are thankful to anonymous reviewers for their detailed comments on earlier versions of the paper.

7. REFERENCES

[1] Abadi, M., and Cardelli, L. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Agesen, O., Freund, S., and Mitchell, J. Adding type parameterization to java. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (1997).

[3] Aldrich, J., and Chambers, C. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)* (Oslo, Norway, 2004), Springer-Verlag.

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

[5] Baker, H. G. Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming* (Nice, France, June 1990), pp. 218–226.

[6] Banerjee, A., and Naumann, D. A. Ownership confinement ensures representation independence for object-oriented programs. Journal version of POPL 2002 paper, submitted., 2002.

[7] Banerjee, A., and Naumann, D. A. Representation independence, confinement and access control. In *POPL* (2002), pp. 166–177.

[8] Barnett, M., DeLine, R., Fahndrich, M., Leino, K. R. M., and Schulte, W. Verification of object-oriented programs with invariants. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming* (Darmstadt, Germany, July 2003), Springer-Verlag.

[9] Bokowski, B., and Vitek, J. Confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications* (1999), ACM Press.

[10] Boyapati, C. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2004.

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

[12] Boyapati, C., Lee, R., and Rinard, M. Safe runtime downcasts with ownership types. In *International Workshop on Aliasing, Confinement and Ownership (IWACO)*, D. Clarke, Ed. Utrecht University, July 2003, pp. 1 – 14.

[13] Boyapati, C., Liskov, B., and Shrira, L. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 2003).

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

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

[16] Clarke, D. *Object ownership and containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.

[17] Clarke, D., and Drossopoulou, S. Ownership, encapsulation, and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (2002).

[18] Clarke, D., Potter, J., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications* (1998), ACM Press.

[19] Clarke, D., Richmond, M., and Noble, J. Saving the world from bad beans: Deployment-time confinement checking. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (October 2003).

[20] Clarke, D., and Wrigstad, T. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)* (Darmstadt, Germany, July 2003), L. Cardelli, Ed., vol. 2473 of *Lecture Notes In Computer Science*, Springer-Verlag, pp. 176–200.

[21] Diwan, A., McKinley, K. S., and Moss, J. E. B. Type-based alias analysis. In *Proceedigns of the ACM*

SIGPLAN'98 Conference on Programming Language Design and Implementation (June 1998).

[22] Fluet, M., and Pucella, R. Phantom types and subtyping. In *Proceedings of the 2nd IFIP Intenational Conference on Theoretical Computer Science (TCS)* (Aug. 2002), pp. 448–460.

[23] Grothoff, C., Palsberg, J., and Vitek, J. Encapsulating objects with Confined Types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications* (2001), ACM Press.

[24] Hogg, J. Islands: Aliasing protection in object-oriented languages. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (New York, Nov. 1991), vol. 26, ACM Press, pp. 271–285.

[25] Hogg, J., Lea, D., Wills, A., de Champeaux, D., and Holt, R. The Geneva convention of the treatment of object aliasing. *OOPS Messenger 3*, 2 (April 1992), 11–16.

[26] Igarashi, A., Pierce, B., and Wadler, P. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (N. Y., 1999), L. Meissner, Ed., vol. 34(10), pp. 132–146.

[27] Igarashi, A., Pierce, B. C., and Wadler, P. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)* (2001).

[28] Kennedy, A., and Syme, D. The design and implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation* (2001).

[29] Lamport, L., and Schneider, F. B. Constraints: A uniform approach to aliasing and typing. In *Proceedings of the 12th Annual (ACM) Symposium on Principles of Programming Languages* (New Orleans, Louisiana, 1985), pp. 205–216.

[30] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (dec 1995), 293–341.

[31] Leino, K. R. M., and Muller, P. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)* (Oslo, Norway, 2004), Springer-Verlag.

[32] Milner, R. Theory of type polymorphism in programming. *Journal of Computer and System Sciences 17(3)* (1978), 348–375.

[33] Müller, P., and Poetzsch-Heffter, A. *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999, ch. Universes: a type system for controlling representation exposure.

[34] Myers, A. C., Bank, J. A., and Liskov, B. Parameterized Types for Java. In *POPL Proceedings* (1997).

[35] Noble, J., Biddle, R., Tempero, E., Potanin, A., and Clarke, D. Towards a model of encapsulation, 2003. Presented at the ECOOP 2003 IWACO Workshop on Aliasing, Confinement, and Ownership. Available at: http://www.mcs.vuw.ac.nz/comp/Publications.

[36] Noble, J., Vitek, J., and Potter, J. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)* (1998), Lecture Notes in Computer Science, Springer-Verlag.

[37] Noble, J., Vitek, J., and Potter, J. Flexible alias protection. In *ECOOP'98— Object-Oriented Programming* (Berlin, Heidelberg, New York, July 1998), E. Jul, Ed., vol. 1445 of *Lecture Notes In Computer Science*, Springer-Verlag, pp. 158–185.

[38] O'Callahan, R., and Jackson, D. Lackwit: a program understanding tool based on type inference. In *1997 International Conference on Software Engineering* (Boston, USA, May 1997).

[39] Odersky, M., and Wadler, P. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages* (January 1997).

[40] Potanin, A., Noble, J., Clarke, D., and Biddle, R. Featherweight generic confinement. In *Foundations of Object-Oriented Programming (FOOL11)* (Venice, Italy, January 2004).

[41] Pugh, B. Find Bugs — A Bug Pattern Detector for Java. http://www.cs.umd.edu/~pugh/jva/bugs/, 2003.

[42] Sun Microsystems. JSR14 prototype implementation. http://developer.java.sun.com/developer/earlyAccess/addi 2003.

[43] Talpin, J.-P., and Jouvelot, P. Polymorphic type, region, and effect inference. *Journal of Functional Programming 2*, 3 (July 1992), 245–271.

[44] Tofte, M., and Talpin, J.-P. Region-Based Memory Management. *Information and Computation 132*, 2 (1997), 109–176.

[45] Zhao, T., Palsberg, J., and Vitek, J. Lightweight confinement for Featherweight Java. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (October 2003).