Generic Ownership for Generic Java

Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle[2]
Victoria University of Wellington, New Zealand
{alex, kjx}@mcs.vuw.ac.nz
[1] CWI, The Netherlands
dave@cwi.nl
[2] Carleton University, Canada
robert_biddle@carleton.ca

ABSTRACT

Ownership types enforce encapsulation in object-oriented programs by ensuring that objects cannot be leaked beyond object(s) that *own* them. Existing ownership programming languages either do not support parametric polymorphism (type genericity) or attempt to add it on top of ownership restrictions. Generic Ownership provides per-object ownership on top of a sound generic imperative language. The resulting system not only provides ownership guarantees comparable to established systems, but also requires few additional language mechanisms due to full reuse of parametric polymorphism. We formalise the core of Generic Ownership, highlighting that only restriction of `this` calls and owner subtype preservation are required to achieve deep ownership. Finally we describe how Ownership Generic Java (OGJ) was implemented as a minimal extension to Generic Java in the hope of bringing ownership types into mainstream programming.

Categories and Subject Descriptors

D.3.2 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and Objects, Polymorphism*

General Terms

Design, Languages, Theory

Keywords

generics, ownership, type system, Java

1. INTRODUCTION

Object ownership (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 ownership 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 [52, 18]. Programs must be written to follow a set of specific conventions, conformance to which can be checked to see if they provide ownership guarantees [23]. The soundness of this first approach has been proven only recently [53]. Support for generics is added on top of such collections of restrictions that enforce encapsulation [53]. The second approach is taken by languages such as Joe, Universes, AliasJava, and SafeConcurrentJava [16, 38, 3, 9]. Ownership parameterisation is added to the syntax and expressed explicitly within the type systems of these languages. All of these different type systems employ ownership parameterisation, but none has support for type genericity.

Why would we want to combine 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 a box of any things). In a language with generics, we are allowed to say: "this is a box *of books*", denoting a box containing books, but not 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 a box of 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.

In this paper, we present Generic Ownership, a language design that uses a single parameter space to carry *both* generic type and ownership type information. We show that ownership systems can be subsumed completely by parametric polymorphic type systems, and formalise our approach within the context of an imperative extension to Featherweight Generic Java (FGJ) [26].

Contributions. The contributions of this paper are as follows:

- Generic Ownership — a practical way of integrating ownership and genericity by using a single parameter space to carry both type and ownership information;

- Featherweight Generic Ownership (FGO) — a formal model with soundness proofs demonstrating that (1) disallowing non-`this` calls on *owned* objects, (2) preserving owners as part of the type, and (3) preserving owner nesting is sufficient to provide ownership and confinement support, as long as the underlying sound type system supports genericity;

- Ownership Generic Java (OGJ) — a publicly available extension to Java 5 that supports Generic Ownership — which is the first language implementation that supports ownership, confinement, and generics at the same time, while preserving full syntactic compatibility with Java 5.

These contributions represent a significant advance over our previous work on Featherweight Generic Confinement [46, 44] which focused on confinement within static contexts (e.g. Java packages)

and was formalised without heap objects, pointers, or assignment. In this paper we present an imperative, object-oriented formalism including pointers and assignment that supports full, deep per-object ownership within dynamic contexts, and describe a language design and prototype implementation. We build upon a preliminary workshop paper [45] that provided only shallow ownership and did not discuss a language design.

The primary goal of this work is not to develop a higher level abstraction unifying ownership and genericity. Generic Ownership's contribution is a language design, that, by building upon an *existing* generic type system, can smoothly extend Java-like languages to provide ownership types. Our goal is also different to other recent ownership types work [41, 6, 35, 2, 30, 32] which investigates new kinds of topologies or restrictions or applications for ownership types. Rather, we hope that Generic Ownership will provide the means by which an ownership system can be incorporated into existing generically typed programming languages.

Outline. In Section 2 we present a background overview of ownership and genericity. Section 3 presents the fundamentals behind Generic Ownership and introduces our language design OGJ. Section 4 highlights the important aspects of the type system (available in full as a technical report[1]). Section 5 presents the dynamics of FGO, as well as the type soundness of the type system as a whole. Section 6 discusses the ownership guarantees provided by FGO. Finally, Section 7 discusses related work and Section 8 outlines future work and 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 simple `Map` class that uses neither genericity nor ownership. The map is implemented using a `Vector` containing a number of `Nodes`, each of which stores a key-value pair. The main `Map` 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 `Nodes` rather than *anything*), better compile-time error detection, and more readable and reusable code. The code in Figure 1 exhibits a number of well-known weaknesses [13, 29]. 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`.

This is an old problem, and the solutions are equally old, dating back to the mid-1970s [36]: the class definitions must be made generic (or parametrically polymorphic) so that particular instances of the classes can be created for particular argument types. Java now supports generic types [48, 13] and Figure 2 presents a version of the `Map` class written using genericity, in Java 5 syntax.

Comparing Figures 1 and 2 illustrates both the advantages and disadvantages of generic types. Regarding the advantages, the types

```
public class Map {
  private Vector nodes;

  void put(Comparable key, Object value) {
    nodes.add(new Node(key, value));
  }
  Object get(Comparable k) {
    Iterator i = nodes.iterator();
    while (i.hasNext()) {
      Node mn = (Node) i.next();
      if (((Comparable) mn.key).equals(k))
        return mn.value;
    }
    return null;
  }
}
class Node {
  public Object key; public Object value;
  Node(Object key, Object value) {
    this.key = key; this.value = value;
  }
}
```

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

```
public class Map<Key extends Comparable, Value> {
  private Vector<Node<Key, Value>> nodes;

  void put(Key key, Value value) {
    nodes.add(new Node<Key, Value>(key, value));
  }
  Value get(Key k) {
    Iterator<Node<Key, Value>> i = nodes.iterator();
    while (i.hasNext()) {
      Node<Key, Value> mn = i.next();
      if (mn.key.equals(k))
        return mn.value;
    }
    return null;
  }
}
class Node<Key extends Comparable, Value> {
  public Key key; public Value value;
  Node(Key key, Value value) {
    this.key = key; this.value = value;
  }
}
```

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

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 types like "`Vector<Node <Key, Value>> nodes`" rather than "`Vector nodes`".

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 [39, 52].

To return to our example, considering the basic `Map` implementation from Figure 1, the `nodes` 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 `Map` class. This is done because the `Vector` is an internal part of the implementation of the `Map` class and should not be accessed outside. Inserting or removing elements from the `Vector`, or perhaps acquiring but not releasing its internal lock would break the invariants of the `Map` class and 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 `Map`. An erroneous programmer could insert a public method that exposed the `Vector`, e.g.

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

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 [47] and have caused significant problems for language security mechanisms [52].

Ownership types [17, 16, 38, 3, 9, 37] protect against aliasing errors by allowing programmers to restrict access to objects at runtime, rather than just the names or variables used to store them. The key idea is that *representation* objects (like the `Map`'s `Vector`) are nested and encapsulated inside the objects to which they belong (the `Vector` belongs to the `Map`; the `Map` owns the `Vector`). Clarke [15] formulated an ownership invariant: that there can be no incoming references that bypass owners. This means that one object cannot refer to a second object directly, unless the first object is itself inside the second object's owner. Because this nesting (and thus the protection) is transitive, we call this *deep* ownership [19]: if an `Array` is part of a `Vector`'s representation, the array should be owned by the vector and is thus nested inside both the vector and the map. Enforcing encapsulation via deep ownership has many practical and theoretical applications [5, 32, 6, 2].

Figure 3 gives an example of the `Map` class using ownership types. The syntax used is proposed by Boyapati [6, Figure 2.7]. 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 type parameter* definitions such as "<mOwner, kOwner, vOwner>" on the class declarations. This declares a parameter, `mOwner`, to represent the ownership of the instances of the classes being declared, with further ownership parameters "kOwner" and "vOwner" 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 `Node` is created within the `put` method of `Map`.

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 [17]. The `Vector` object is marked in this way as being owned by the `Map`, for example, so any attempt to access or pass the `Vector` object outside the `Map` 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 information around the program, so that the ownership status of the keys and values can be maintained outside the `Map`. For example, the ownership of the keys and values may be specified by each instantiation of the `Map` class, but by using the `kOwner` and `vOwner` ownership parameters, the fields that will store keys and values inside the subsidiary `Node`

```
public class Map<mOwner, kOwner, vOwner> {
  private Vector<this, this> nodes;

  void put(Comparable<kOwner> key, Object<vOwner> value) {
    nodes.add(new Node<this, kOwner, vOwner>(key, value));
  }
  Object<vOwner> get(Comparable<kOwner> key) {
    Iterator<this, this> i = nodes.iterator();
    while (i.hasNext()) {
      Node<this, kOwner, vOwner> mn =
        (Node<this, kOwner, vOwner>) i.next();
      if (mn.key.equals(key))
        return mn.value;
    }
    return null;
  }
}
class Node<mapNodeOwner, kOwner, vOwner>
  public Comparable<kOwner> key; public Object<vOwner> value;
  Node(Comparable<kOwner> key, Object<vOwner> value) {
    this.key = key; this.value = value;
  }
}
```

Figure 3: An Ownership Types implementation of a `Map` class

```
public class Map<mOwner, kOwner, vOwner>
  [Key extends Comparable<kOwner>,
   Value extends Object<vOwner>] {

  private Vector<this, kOwner, vOwner>
    [Node<this, kOwner, vOwner>[Key, Value]] nodes;
  void put(Key key, Value value) {
    nodes.add(new Node<this, kOwner, vOwner>
      [Key, Value](key, value));
  }

  Value get(Key key) {
    Iterator<this>[Node<this, kOwner, vOwner>
      [Key, Value]] i = nodes.iterator();

    while(i.hasNext()) {
      Node<this, kOwner, vOwner>[Key, Value] mn = i.next();

      if (mn.key.equals(k)) {
        return mn.value;
    } }
    return null;
  }
}
class Node<mapNodeOwner, kOwner, vOwner>
  [Key extends Comparable<kOwner>,
   Value extends Object<vOwner>] {
  public Key key; public Value value;
  Node(Key key, Value value) {
    this.key = key; this.value = value;
  }
}
```

Figure 4: A combined generic and ownership types `Map` class

objects will have the correct ownership for these fields.

The main disadvantage of ownership types is quite similar to that of generic types: additional syntactic complexity as a result of ownership parameters. In fact, this code also has all the type-related problems of the "straight" Java code: the problems that are addressed by genericity. This code relies on subtyping to store different types of objects, and so requires type casts when objects are removed from the `Vector` (or subsequently from the `Map`). Although the code may look generic, all the declared types are simple Java types, such as `Object` or `Comparable`, with the problems that entail. While the `Vector` stored in the `nodes` field can no longer be exposed outside of the `Map` instance that owns it, malicious or buggy programming within that class can break this

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 `Map` example using a hypothetical language separately supporting both genericity with parameters marked with `[` and `]` and ownership with parameters marked with `<` and `>`. The syntax in this Figure is based on Boyapati [6, page 29] and resembles the original Flexible Alias Protection proposal [39].

Again we can compare Figure 4 with the preceding Figures 1, 2, and 3. This code now has both type and ownership 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 thus any exposing method would be detected and prevented.

Unfortunately, the syntax required to implement both ownership and genericity separately means that this code is annotated in a significantly more complex way than any of the other examples — with both classes requiring *five* ownership and type parameters, some of which are not directly utilised by the code. The code is, arguably, so complex that it would be unusable in practice.

3. GENERIC OWNERSHIP

Generic Ownership takes a novel approach of providing ownership and genericity in a programming language. As opposed to the hypothetical example of Figure 4 or the previous work on combining confinement and genericity [53] — Generic Ownership starts with a sound type polymorphic language and adds ownership as an extension to the existing generic type system. Providing genericity before ownership, surprisingly, results in both simpler formalism and a simpler language that 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. Generic Ownership treats ownership and genericity as one single aspect of language design, and so code using Generic Ownership is no more syntactically 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 the existing generic type systems can be extended to carry ownership information with only minimal changes [46].

Figure 5 revisits our `Map` example for the last time. This time, it is written in our new language design, Ownership Generic Java (OGJ). Note that the code in Figure 5 is type-generic: definitions of fields in `Node` and methods everywhere use generic types such as `Key` and `Value` rather than plain class types such as `Object` or `Comparable`. Note also that the code is ownership-generic. Every class has an extra type parameter that represents the object's owner: we place it *last*, typically name it `Owner` and call it the *owner parameter*. When creating an object, we can mark it as owned by the current object `this` by instantiating the new object's owner parameter with the owner constant `This`. All OGJ classes descend from a new parameterised root `Object<O>` that declares an owner parameter, and all subclasses must invariantly preserve their owner parameter. Owners descend from a separate root, `World`, which

```
public class Map<Key extends Comparable<KOwner>,
                 Value extends Object<VOwner>,
                 Owner extends World> {
  private Vector<Node<Key, Value, This>, This> nodes;
  public void put(Key key, Value value) {
    nodes.add((new Node<Key, Value, This>()).
             init(key, value));
  }
  public Value get(Key key) {
    Iterator<Node<Key, Value, This>, This> i =
      nodes.iterator();
    while (i.hasNext()) {
      Node<Key, Value, This> mn = i.next();
        if (mn.key.equals(key))
          return mn.value;
    }
    return null;
  }
}
class Node<Key extends Comparable<KOwner>,
           Value extends Object<VOwner>,
           Owner extends World> {
  public Key key; public Value value;
  public void init(Key key, Value value) {
    this.key = key; this.value = value;
  }
}
```

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

also acts as a second owner constant, meaning that access to an object is unrestricted.

Consider an example exposing a `nodes` vector private to the `Map` object considered earlier in Section 2:

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

This code is not valid in OGJ as is, since every type has an owner parameter and casting to raw types [27] is prohibited. The type of the field `nodes` 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<Node<Key, Value, This>, This>
    exposeVector() { return nodes; }
```

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, any valid supertype of the return type will have to have an owner `This`, which will only typecheck if it is called from *the same instance* of `Map`. This is enforced by the *this* function described in Section 4.3. In other words, this `exposeVector` method *cannot* expose the vector. If we attempt to declare this method with a return type having any other owner parameter, then the return type and the return value's (`nodes`) type will not be assignment compatible in OGJ, as their owner parameters will be different.

3.1 Expressiveness of Generic Ownership

Comparing Figure 5 with Figures 1–4 shows that it is slightly more complex than the individual type genericity or ownership examples, but rather simpler than the straightforward combination in Figure 4. In particular, Figure 5's declaration of the `Map` class has only *three* parameters (the same as in Figures 2 and 3). Furthermore, the type of field `nodes` in Figure 2 is more readable:

```
Vector<Node<Key, Value, This>, This> nodes;
```

rather than:

```
Vector<this, kOwner, vOwner>
  [Nodes<this, kOwner, vOwner>[Key, Value]] nodes;
```

presented in Figure 4.

Because every OGJ class has a distinguished owner parameter, the bounds of formal generic parameters (e.g. `Key extends Comparable<KOwner>`) must be declared with a *placeholder* owner parameter (`KOwner` in this case). Apart from this declaration, programmers are *not* required to supply placeholder parameters: rather they are bound implicitly, with very similar semantics to Java wildcards [51, 34]. Within the scope of the type parameter declaration (a generic type or method) the bound type variable (e.g. `Key`) ranges simultaneously over type and owner. Owners can also be passed explicitly (generally as arguments to formal parameters bound by `World`) — indeed, the distinguished owner parameter is just a special case of this.

This implicit binding of owner parameters — and, more importantly, the combination of ownership into types — reduces the number of formal arguments required by generic ownership. Programmers do not need to write code such as:

```
public class Map<KO extends World, VO extends World,
 Key extends Comparable<KO>, Value extends Object<VO>,
 Owner extends World> { ... }
```

because the owners are bound implicitly. The implicit binding is generally sufficient: we find that ownership and type parameters are hardly ever used independently.

For example, if a programmer wished to produce a specialised version of a map class (`MyMap`) that was not type generic — say it could store only names and addresses — but which needed to be ownership generic, additional type parameters (`NameO`, `AdrO`) bound by `World` can be declared explicitly to carry the ownership for names and addresses.

```
class MyMap<NameO extends World, AdrO extends World,
  Owner extends World> {
    void    put(Name<NameO> n, Address<AdrO> a);
    Address get(Name<NameO> n);
    List<Name<NameO>, World>
      getAllPeopleOnThisStreet(String s);
}
```

On the other hand, given that the `MyMap` class requires three parameters anyway, standard OGJ style is to declare two bound generic parameters, and always instantiate those parameters with the same types as their bounds (although, for ownership polymorphism, with different owners):

```
class MyMap<MyName extends Name<NameO>,
            MyAddress extends Address<AdrO>,
            Owner extends World> {
    void    put(MyName n, MyAddress a);
    Address get(MyName n);
    List<MyName, World>
      getAllPeopleOnThisStreet(String s);
}
```

Finally, to live up to the full potential of generics in Java 5, Generic Ownership allows owner parameters to be mixed with generic method parameters. Owners on methods allow more granular control of what access each method has to the other objects, and they can be utilised usefully for more granular alias control with little overhead for the programmer.

To summarise this subsection: these examples show how OGJ can provide independent ownership and type genericity — with five separate parameters — and ownership genericity without type genericity — with "naked" owner parameters, or by using parameters

instantiated at their type bounds. Programmers can provide type parametricity without ownership parametricity either by supplying ownership constants (e.g. `World`) at point of use, or by using manifest ownership (described below) to fix an owner for all instances of a class.

3.2 OGJ and Confinement

Building on our previous work, OGJ supports *both* ownership and static *confinement* — allowing an object to be owned by a static package, rather than by another dynamically allocated object [46]. This is expressed by using an owner constant named after the current package — so a package called m would have a matching owner constant `M`, that is only accessible within its package: as a convention, we will prefix class names by the name of their defining packages where necessary. This means that `mMain` below refers to class `Main` inside package m. Whereas objects owned by `this` can only be accessed from within their owner, objects owned by a package can be accessed by any object within that package.

The following declaration of a class called `Main` shows how an OGJ class is declared — with its owner parameter `Owner` bound to its superclass's owner parameter (`Object` in this case), and then how a stack class can be instantiated with different ownership.

```
class mMain<Owner extends World>
  extends Object<Owner>{
  OwnedStack<Object<World>,World> public() {
    return new OwnedStack<Object<World>,World>; }
  OwnedStack<mMain<World>,M> confined() {
    return new OwnedStack<mMain<World>,M>; }
  OwnedStack<mMain<M>,This> private() {
    return new OwnedStack<mMain<M>,This>; }
  OwnedStack<mMain<M>,Owner> shared() {
    return new OwnedStack<mMain<M>,Owner>; }
}
```

Within the `Main` class, four methods return different kinds of `OwnedStack` objects: one of these is public, another is confined to package m, the next one is owned by a particular instance of class `Main`, and the last one is owned by the owner of `mMain` class and can be shared with others owned by the same owner. The public stack stores `Object<World>` instances that are accessible from anywhere (because `Object`'s owner parameter is instantiated by `World`). The second stack stores `mMain` instances that are also globally accessible, however the stack itself has owner `M`, meaning that it is only accessible within package m. The private stack stores instances of `mMain` accessible inside package m only, while the actual stack is only accessible by objects owned by a particular instance of `mMain` that created it. The shared stack stores the same sort of instances of `mMain` as the private stack, except that it is accessible by any other instances who have the same owner as current instance of `mMain`. In each case, the stack's second parameter describes its owner. Again, these stacks illustrate how OGJ provides both *type polymorphism* (the stacks hold different item types) and *ownership polymorphism* (the stacks belong in different protection contexts).

3.3 Manifest Ownership

OGJ supports a form of *manifest ownership* [15] to allow classes without explicit owner type parameters. A manifest class does not have an explicit owner parameter, rather the class's owner is fixed, so all the objects of that class have the same owner. This is just the same way that in Generic Java, for example, a non-generic `IntegerList` class can be defined as extending `List<Integer>`, binding and fixing the list's type parameter. To demonstrate manifest ownership, consider the following alternative formulation of a public stack class:

```
import ogj.ownership.*;

class Point<Owner extends World> {
  Integer x; Integer y;
  Point(Integer x, Integer y) {
    this.x = x; this.y = y;
  }
}

class Rectangle<Owner extends World> {
  private Point<This> upperLeft;
  private Point<This> lowerRight;

  public Rectangle(Point<Owner> ul, Point<Owner> lr) {
    // Copy rather than assignment is enforced by Java
    // (and hence OGJ) since type parameter Owner != This.
    upperLeft = new Point<This>(ul.x, ul.y);
    lowerRight = new Point<This>(lr.x, lr.y);
  }

  public void doIt() {
    Point<This> p;
    p = this.upperLeft;
    p = this.exposeUpperLeft();
    Rectangle<Owner> ro = this;
    p = ro.upperLeft; // WRONG in OGJ, OK in Java
    p = ro.exposeUpperLeft(); // WRONG in OGJ, OK in Java
  }

  private Point<This> exposeUpperLeft() {
    return upperLeft;
  }

  public Point<Owner> getUpperLeft() {
    // return upperLeft; // WRONG in both Java and OGJ.
    return new Point<Owner>(upperLeft.x, upperLeft.y);
  }
}
```

Figure 6: `Rectangle` Class in OGJ

```
class PublicStack extends OwnedStack<World>{ }
```

In this example, the *owner* of class `PublicStack` is `World`, and thus all of its instances are owned by `World`. Because the owner is bound in the class declaration, uses of `PublicStack` require no owner type parameter. Manifest ownership allows us to fit existing Java classes into our class hierarchy by simply making Java's root class `Object` into a manifest class[2]:

```
class Object extends Object<World> { ... }
```

With this definition `Object` and every class inheriting from it has a default owner parameter `World` (thus making them publicly accessible). We can write the following familiar declaration of a public `Stack` object, which is indistinguishable from that of Java:

```
class Stack extends Object { ... }
```

The important difference is that with manifest ownership, every `Stack` instance has an owner originating from OGJ's root class.

3.4 OGJ Language Design

We conclude this section by bringing together the details of our language design. Ownership Generic Java is designed as a minimal extension to Java 5 [13]. The key difference is that OGJ allows classes to be declared with a distinguished (last) ownership type parameter (conventionally `Owner`) extending `World`. Classes without an ownership parameter ("plain Java" classes descending from

java.lang.Object) are treated as if they used manifest ownership.

OGJ supplies a number of ownership type constants: `World`, `Package`, and `This`. When bound to a class's ownership type parameter, these constants mark the instances as public, as confined within their package, or as owned by the current "`this`" object, respectively. To ensure deep ownership, OGJ restricts the types which can be formed so that the distinguished ownership type parameter is always *inside or equal to* (greater or equally encapsulated than) any other parameters' ownership type. In practice, this means that when an actual owner parameter is `World`, all the other type parameters must have an owner `World`; if the actual owner parameter is a package or an ownership type variable, then that package (or variable) and `World` are permissible; if the actual owner parameter is an enclosing class's `Owner` then `World`, `Owner`, and other (placeholder) formal ownership type parameters of that class are permissible; and if an ownership type parameter is bound to `This`, then the other parameters may be bound to anything.

This is sufficient to provide package confinement. To enforce per-object ownership OGJ ensures that types with an actual owner parameter of `This` can only be accessed via Java's `this` keyword, either explicitly or implicitly. Assignments such as `this.pvtField = other.pvtField` between two `Node` instances are illegal if `this.pvtField` is owned by this; similarly method or field accesses involving a `This` owner are only permitted on the current "`this`" object.

Finally, to ensure that ownership information cannot be lost, OGJ requires type casts to preserve ownership. This follows Java's existing rules for subtyping parameterised types, except that it prevents casts to raw types [27, 13] when such casts would delete an ownership parameter. OGJ must also restrict wildcards for ownership types to type variable bounds, and prevent reflection when it could breach ownership.

These are *the only* restrictions imposed by Generic Ownership on top of the vanilla type generic language. This achieves ownership *and* confinement guarantees comparable to the alternative systems [53, 3, 6] with a large burden carried by the underlying sound type polymorphic system. Hence, these restrictions highlight the concepts important to the mechanism of ownership, excluding the rest of the rules required by the alternative type systems as technical details.

Our prototype implementation [43] includes a compiler that can bootstrap itself (albeit with substantial use of standard (manifest) Java classes) and compile a growing test suite. There are a number of implementation-specific issues omitted in this paper, such as distinguishing `This` between an inner class and the outer class, dealing with generic arrays and static fields and methods. All of these issues are addressed in the thesis referenced in our accompanying technical report. Our future plans include the development of an ownership-aware version of the Java Collections library to support larger OGJ programs without resorting to manifest ownership.

To demonstrate the difference between OGJ and the other ownership languages, consider the examples in the Figures 6 and 7. Both of these simple examples were compiled with the latest version of AliasJava[3] and OGJ[4]. Both examples have two private fields protected from erroneous exposure by making them *owned* by the instance of `Rectangle` that created them. The constructor is forced to make a private copy of the supplied `Point` references. The method `doIt` fails to expose the private fields via `exposeUpperLeft()` unless the method receiver is explicitly

---

[2]To avoid class name conflict with `Object`, OGJ can choose a different name for its root, such as `OObject<World>`, meaning *owned* object.

[3]`http://www.archjava.org/`
[4]`http://www.mcs.vuw.ac.nz/~alex/ogj/`

```
class Point {
  int x; int y;
  Point(int x, int y) {
    this.x = x; this.y = y;
  }
}

class Rectangle {
  private owned Point upperLeft;
  private owned Point lowerRight;

  public Rectangle(Point ul, Point lr) {
    // ArchJava forces to copy the values, since the default
    // annotation (lent) doesn't match annotation owned.
    upperLeft = new Point(ul.x, ul.y);
    lowerRight = new Point(lr.x, lr.y);
  }

  public void doIt() {
    owned Point p;
    p = this.upperLeft;
    p = this.exposeUpperLeft();
    Rectangle r = this;
    p = r.upperLeft; // WRONG
    p = r.exposeUpperLeft(); // WRONG
  }

  private owned Point exposeUpperLeft() {
    return upperLeft;
  }

  // The copy is enforced using unique annotation.
  public unique Point getUpperLeft() {
    // return upperLeft; // WRONG.
    return new Point(upperLeft.x, upperLeft.y); // RIGHT
  }
}
```

Figure 7: `Rectangle` Class in AliasJava

this. It is interesting to observe that a lot of work to do with aliasing protection is performed by Java 5 on behalf of OGJ, as the comments in the figure point out. These have to do with type parameters (which in OGJ can also include owner classes) checked by Java's type system.

AliasJava adds support for ownership by introducing extra annotations such as owned added on top of Java syntax, while OGJ's syntax is completely Java compatible. OGJ also supports confinement, in addition to ownership, while still keeping Java syntax. This means that it inherits the problems of Generic Java, such as lack of proper generic array support. Most importantly, OGJ is the first working language implementation that supports ownership, confinement, and generic types at the same time.

4. FEATHERWEIGHT GENERIC OWNERSHIP

In this section, we present Featherweight Generic Ownership (FGO) and its type system. We started the development of FGO by taking Featherweight Generic Java (FGJ) [26] and adding imperative features [42] to be able to take into account assignments and field updates in the presence of per-object ownership. As described above, we then added a separate hierarchy of owner classes rooted in World that are used to carry ownership information for each FGO type. In fact, if we only aim to support static confinement, then we need add little other than owner classes to FGJ as we demonstrate elsewhere [46]. The key formal contribution of this work is that, by adding the imperative features (assignment, object store, etc) we demonstrate *object* ownership in dynamic contexts.

$$T ::= X \mid N$$
$$N ::= C < \overline{T} >$$
$$L ::= \text{class } C < \overline{X} \lhd \overline{N} > \lhd N\{\overline{T} \, \overline{f}; \overline{M}\}$$
$$M ::= < \overline{X} \lhd \overline{N} > T \, m(\overline{T} \, \overline{x}) \, \{\text{return } e; \}$$
$$e ::= x \mid e.f \mid e.m < \overline{T} >(\overline{e}) \mid \text{new } N() \mid (N) \, e \mid e.f = e \mid$$
$$\quad \text{let } x = e \text{ in } e \mid l \mid l > e \mid \text{null}$$
$$v ::= l \mid \text{null}$$

Environment and Permission:
$$\Delta = \{x : T\} \cup \{X \ll N\} \cup \{l : T\} \qquad P ::= C \mid l$$

Store (Heap):
$$l \in locations \qquad S ::= \{l \mapsto N(\overline{v})\}$$

Figure 8: FGO Syntax

To support dynamic deep ownership, in addition to adding owners to types, we prevent non-this access to types *owned* by particular instances, ensure owner invariance over subtyping, and provide for owner parameter nesting. This lets us prove the deep ownership property known as *owners as dominators* (objects are nested within their owners). The remainder of this section highlights the important aspects of the FGO type system.

4.1 Syntax, Functions, and Judgements

Figure 8 shows FGO's syntax. The syntax is derived from FGJ by adding expressions for locations (references), let (local variables), assignment (field update), and null. The figure contains definitions for syntactical terms corresponding to types (T), type variables (X), nonvariable types (N), class declarations (L), method declarations (M), and expressions (e). The environment Δ stores mappings from variables to their types, mappings from type variables to nonvariable types, and the types of the locations. There is no explicit constructor declaration: fields are initialised to null. This is important because the ownership types mean that objects owned by This cannot be constructed outside the object to which they belong.

FGO's additions to FGJ syntax include the locations (l), the store (S) and adding location types to the environment (Δ). The expression $l > e$ represents the expression e resulting from reducing a method call with the receiver object allocated at the location l in the store S — this allows us to type the occurrences of this expression. We use *CT* (class table) to denote a mapping from class names C to class declarations L, and *P* (permission) to denote a class or a location. Permissions are used to implement object ownership using *this* function described in Section 4.3. FGO also adopts the syntactical idiosyncrasies of FGJ, such as $\overline{T} \, \overline{x}$ denoting a list of pairs: $T_1 \, x_1, \ldots, T_n \, x_n$, rather than two lists of types and variables.

FGO ownership types are just types, but we assume that owners are syntactically distinguishable:

$$O ::= X^O \mid N^O$$

where O ranges over all owners, X^O ranges over owner variables, and N^O ranges over nonvariable owners such as World and This, as well as the owner classes corresponding to packages. We use capitals (P) for the owner class corresponding to a lower case package name (p). We use subscript ($This_l$) for the owner class corresponding to the owner of an object at location l. Pure FGO types and classes are written to include an owner class as their last type parameter or argument, which can be distinguished using the following syntax:

$$N_{pure} ::= C < \overline{T}, O >$$
$$L_{pure} ::= \text{class } C < \overline{X} \lhd \overline{N}, X^O \lhd N^O > \lhd N \{\overline{T} \, \overline{f}; \, \overline{M}\}.$$

$$\frac{\text{X} \in dom(\Delta)}{\Delta \vdash \text{X OK}} \quad \text{(WF-VAR)}$$

$$\frac{\Delta \vdash \text{O} \lessdot \text{World}}{\Delta \vdash \text{Object} < \text{O} > \text{OK}} \quad \text{(WF-OBJECT)}$$

(WF-TYPE):
$$\frac{\begin{array}{c} \text{class C} < \overline{\text{X}} \lhd \overline{\text{N}} > \lhd \text{N} \{...\} \quad \Delta \vdash \text{N} \lessdot \text{Object} < \text{O} > \\ \Delta \vdash \text{O} \lessdot \text{World} \quad \Delta \vdash \overline{\text{T}} \text{ OK} \quad \Delta \vdash \overline{\text{T}} \lessdot [\overline{\text{T}}/\overline{\text{X}}]\overline{\text{N}} \\ \boxed{\forall \text{T} \in \overline{\text{T}} : owner_\Delta(\text{C} < \overline{\text{T}} >) \lessdot owner_\Delta(\text{T})} \end{array}}{\Delta \vdash \text{C} < \overline{\text{T}} > \text{ OK}}$$

Figure 9: FGO Type Well-Formedness Rules. WF-TYPE **rule enforces ownership with the grey clause checking the nesting among different owners present in the same type.**

While the last type parameter is distinguished to make it easy to identify *the* owner of a particular instance, other type parameters can be "naked" owners as well as type parameters with no syntactic distinction made by the type system.

FGO makes use of a number of functions to simplify the presentation:

π_C	the package owner class corresponding to class C
$this_P(\texttt{e})$	validates the use of this. calls in e (Section 4.3)
$owner_\Delta(\text{T})$	the owner of type T
$visible_\Delta(\text{O}, \text{C})$	owner O is visible in class C
$visible_\Delta(\text{T}, \text{C})$	type T is visible in class C

π_C is assumed to be an implicit lookup function; *this*, *owner*, and *visible* are described in detail in the rest of this section. Finally, we use the following judgements:

$\Delta \vdash \text{T OK}$	Type T is OK.
$\Delta \vdash \text{T} \lessdot \text{U}$	Type T is a subtype of type U.
$\Delta; P \vdash \texttt{e} : \text{T}$	Expression e is well typed.
$\Delta; P \vdash visible(\texttt{e})$	Expression e is visible with respect to P.
$\Delta \vdash S$	Store (heap) is well-formed.
$\Delta \vdash < \overline{\text{Y}} \lhd \overline{P} > \text{ T m}(\overline{\text{T}}\, \overline{\text{x}})\{\texttt{return e}_0; \} \text{ FGO IN C}, \text{C}^0$	
	Method m definition is OK.
$\texttt{class C} < \overline{\text{X}} \lhd \overline{\text{N}} > \lhd \text{N} \{\overline{\text{T}}\, \overline{\text{f}};\, \overline{\text{M}}\} \text{ FGO}$	Class C definition is OK.

4.2 Owner Lookup

$$\begin{aligned} owner_\Delta(\text{N}^0) &= \text{N}^0 \\ owner_\Delta(\text{X}) &= owner_\Delta(\Delta(\text{X})) \\ owner_\Delta(\text{C} < \overline{\text{T}}, \text{O} >) &= \text{O} \\ owner_\Delta(\text{C} < \overline{\text{T}} >) &= owner_\Delta([\overline{\text{T}}/\overline{\text{X}}]\text{N}), \text{ where} \\ & \quad CT(\text{C}) = \texttt{class C} < \overline{\text{X}} \lhd \overline{\text{N}} > \lhd \text{N}\{\overline{\text{T}'}\, \overline{\text{f}};\, \overline{\text{M}}\} \end{aligned}$$

The *owner* function gives the owner of a type. Manifest class owner is found by traversing the class hierarchy.

4.3 *this* Function (FGO-THIS)

$$this_\text{C}(\texttt{this}) = \text{This} \quad this_l(l) = \text{This}_l \quad this_P(...) = \bot$$

The *this* function is used extensively during the typing of FGO expressions. It helps enforce ownership, as it ensures that types involving This can only be used within the current object, that is, as part of message sends or field accesses upon this. Basically,

Bound of Type:
$$bound_\Delta(\text{X}) = \Delta(\text{X}) \qquad bound_\Delta(\text{N}) = \text{N}$$

Subclassing:

$$\text{C} \unlhd \text{C} \qquad \frac{\text{C} \unlhd \text{D} \quad \text{D} \unlhd \text{E}}{\text{C} \unlhd \text{E}}$$

$$\frac{\texttt{class C} < \overline{\text{X}} \lhd \overline{\text{N}} > \lhd \text{D} < \overline{\text{T}} >\{...\}}{\text{C} \unlhd \text{D}}$$

Subtyping:

(S-REFL):
$$\frac{}{\Delta \vdash \text{T} \lessdot \text{T}}$$

(S-TRANS):
$$\frac{\Delta \vdash \text{S} \lessdot \text{T} \quad \Delta \vdash \text{T} \lessdot \text{U}}{\Delta \vdash \text{S} \lessdot \text{U}}$$

(S-VAR):
$$\frac{}{\Delta \vdash \text{X} \lessdot \Delta(\text{X})}$$

(S-CLASS):
$$\frac{\texttt{class C} < \overline{\text{X}} \lhd \overline{\text{N}} > \lhd \text{N} \{...\}}{\Delta \vdash \text{C} < \overline{\text{T}} > \lessdot [\overline{\text{T}}/\overline{\text{X}}]\text{N}}$$

(S-OWNER):
$$\frac{\texttt{class C} < \overline{\text{X}} \lhd \overline{\text{N}} > \lhd \text{N} \{...\}}{\Delta \vdash \pi_\text{C} \lessdot \text{World}} \qquad \frac{l \in dom(\Delta)}{\Delta \vdash \text{This}_l \lessdot owner_\Delta(\Delta(l))}$$

Valid Method Overriding:
$$\frac{\begin{array}{c} mtype(\text{m}, \text{N}) = < \overline{\text{Z}} \lhd \overline{\text{Q}} > \overline{\text{U}} \to \text{U}_0 \\ \Rightarrow \overline{P}, \overline{\text{T}} = [\overline{\text{Y}}/\overline{\text{Z}}](\overline{\text{Q}}, \overline{\text{U}}) \text{ and } \overline{\text{Y}} \lessdot \overline{P} \vdash \text{T}_0 \lessdot [\overline{\text{Y}}/\overline{\text{Z}}]\text{U}_0 \end{array}}{override(\text{m}, \text{N}, < \overline{\text{Y}} \lhd \overline{P} > \overline{\text{T}} \to \text{T}_0)}$$

Figure 10: FGO Subtyping Rules

every occurrence of This in the type of a method call or field access is substituted with the result of calling the *this* function; if the type involves This, the expression will typecheck only if the target of the call or field access is this. As such, the *this* function is one of the key extensions we make to FGJ.

In detail, there are two distinct places where *this* is used. They are distinguished by the permission P that is present on the left hand side of the expression type rules. The first place is during the validation of FGO class declarations (in the class and method typing rules in Figure 13, which rely on expression typing in Figure 11). Here, the permission P is set to the class C currently being validated. When typing a field access or method call inside C, the *this* function is called upon the expression $\texttt{e}_0$ that is the target of the field access ($\texttt{e}_0.\texttt{f}$) or method call ($\texttt{e}_0.\texttt{m}()$). Then, all occurrences of This in the types of the method or field are substituted by the result of the *this* function. If the target is this (e.g. this.f), the *this* function returns This, the substitution will replace This with itself, and so the expression typechecks, even if it involves This types. If the target is other than this, the *this* function returns an undefined ($\bot$) result, so This is substituted by $\bot$ (leaving other types unaffected), and any expressions with This types will fail to typecheck. In this way, FGO ensures that This types can only be used upon this.

The second place *this* appears is during the reduction of FGO expressions (e.g. R-METHOD in Figure 15). In this case, expression types include locations (l). Every occurrence of a This owner is replaced by a location specific This_l. The expression typing rules further ensure that every occurrence of This is made location specific. To achieve this, the T-CONTEXT rule in Figure 11 sets the permission P to the current location l. As the expression typing rules recurse into the structure of the expression e, every occurrence of This is replaced appropriately by $this_l$ to be This_l. This replacement ensures that FGO expression reduction distinguishes

between different instances of the same class, allowing us to prove our ownership system provides per-object ownership guarantees.

In either case, invalid use of an owner class causes the expression type to become undefined ($\bot$). FGO type soundness guarantees that any validated FGO class will not incur an invalid access to a location-specific instance (marked by $This_l$ owner) during reduction.

4.4 Well-formed Types and Subtyping

FGO's type well-formedness rules shown in Figure 9 are the same as those of FGJ, except that the root of the class hierarchy is parameterised. The grey clause in the type formation rule ensures that FGO supports deep ownership: WF-TYPE enforces the nesting of owner parameters essential to ensure *owners as dominators* object encapsulation. The owner nesting required of each type is that the last (distinguished) owner parameter of $C < \overline{T} >$ is *inside* of every other owner of the rest of the type parameters $\overline{T}$.

Figure 10 shows FGO's subtyping rules. They are generally taken verbatim from FGJ, except for the addition of subtyping for owner classes. World forms the top of the owner class hierarchy, which any package owner class extends directly. This also directly extends World, while location-specific version $This_l$ extends This. The latter is important for the subject reduction proof. The owner hierarchy for location-specific owners is built up in the FGO class rules (Figure 13) using owner variables.

4.5 Expressions and Visibility

Figure 11 shows the expression typing rules. These are the standard FGJ rules with added support for locations, assignment, null, and let expressions [42].

Another important addition that FGO makes to FGJ is a set of visibility rules similar to those used by Featherweight Generic Confinement [46]. While a subset of our visibility rules for terms is similar to those used by ConfinedFJ [53], the owner and type visibility rules are part of the foundation of FGO.

Figure 12 shows the owner visibility rule that checks if an owner O is visible inside class C. This is the case if the owner is World, belongs to the same package as C, or is an owner of one of the generic parameters used when instantiating C. Supplying an actual owner parameter to a class gives that class permission to access everything owned by that parameter. This, for example, can allow a type polymorphic class to have private access to more than one package.

Finally, we allow complete visibility of the This owner, and rely on the *this* function described earlier to stop illegal uses of This. Type visibility simply checks the owner of a given type for visibility. Term visibility (not shown) recursively checks all the types involved in the possible expressions of FGO to make sure that they are visible in a given class C. Since these checks are performed on class declarations, locations are not present in these expressions.

4.6 Class and Method

The class and method declarations are checked to make sure that they contain well-formed, visible types and expressions as shown in Figure 13. There are rules for pure and manifest FGO classes that are distinguished only by the fact that the owner is either explicit or must be looked up from a superclass. Both of the class rules check that (1) all the types involved (types of fields and type parameters) are *visible* within the context of the owner of the class being declared or its superclass; (2) all the types are *well formed* FGO types; and (3) all the methods declarations are valid. The grey clauses ensure that owner nesting (the distinguished owner is *inside* the owners of the other type parameters) is preserved for the purposes of the deep ownership invariant proven below in Section 6.

(T-FIELD):
$$\frac{\Delta; P \vdash \mathtt{e_0} : \mathtt{T_0} \quad \Delta \vdash \mathtt{T} \text{ OK}}{fields(bound_\Delta(\mathtt{T_0})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \mathtt{T} = [this_P(\mathtt{e_0})/\mathtt{This}]\mathtt{T}_i}{\Delta; P \vdash \mathtt{e_0.f}_i : \mathtt{T}}$$

(T-FIELD-SET):
$$\frac{\Delta; P \vdash \mathtt{e_0} : \mathtt{T_0} \quad \Delta; P \vdash \mathtt{e} : \mathtt{T} \quad \Delta \vdash \mathtt{T} \text{ OK}}{fields(bound_\Delta(\mathtt{T_0})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \mathtt{T} = [this_P(\mathtt{e_0})/\mathtt{This}]\mathtt{T}_i}{\Delta; P \vdash \mathtt{e_0.f}_i = \mathtt{e} : \mathtt{T}}$$

(T-METHOD):
$$\forall \mathtt{V}' \in \overline{\mathtt{V}} : (\Delta \vdash \mathtt{V}' \text{ OK} \lor \Delta \vdash \mathtt{V}' \lessdot \mathtt{World}) \land$$
$$\land\ (\ owner_\Delta(\mathtt{V}') \lessdot owner_\Delta(P)\ )$$
$$mtype(\mathtt{m}, bound_\Delta(\mathtt{T_0})) = <\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{U}} \to \mathtt{U}$$
$$\Delta; P \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \quad \Delta; P \vdash \mathtt{e_0} : \mathtt{T_0} \quad \Delta \vdash \mathtt{T} \text{ OK}$$
$$\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ this_P(\mathtt{e_0})/\mathtt{This}]\mathtt{U} \quad \Delta \vdash \overline{\mathtt{V}} \lessdot [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ this_P(\mathtt{e_0})/\mathtt{This}]\overline{\mathtt{P}}$$
$$\frac{\Delta \vdash \overline{\mathtt{S}} \lessdot [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ this_P(\mathtt{e_0})/\mathtt{This}]\overline{\mathtt{U}}}{\Delta; P \vdash \mathtt{e_0.m} < \overline{\mathtt{V}} > (\overline{\mathtt{e}}) : \mathtt{T}}$$

(T-CAST):
$$\frac{\Delta \vdash \mathtt{N} \text{ OK} \quad \Delta; P \vdash \mathtt{e_0} : \mathtt{T_0}}{\Delta; P \vdash (\mathtt{N})\mathtt{e_0} : \mathtt{N}}$$

(T-LET):
$$\frac{\Delta; P \vdash \mathtt{e_0} : \mathtt{T_0} \quad \Delta, \mathtt{x} : \mathtt{T_0}; P \vdash \mathtt{e} : \mathtt{T}}{\Delta; P \vdash \mathtt{let}\ \mathtt{x} = \mathtt{e_0}\ \mathtt{in}\ \mathtt{e} : \mathtt{T}}$$

(T-CONTEXT):
$$\frac{\Delta; l \vdash \mathtt{e} : \mathtt{T}}{\Delta; P \vdash l > \mathtt{e} : \mathtt{T}}$$

(T-NEW):
$$\frac{\Delta \vdash \mathtt{N} \text{ OK}}{\Delta; P \vdash \mathtt{new}\ \mathtt{N}() : \mathtt{N}}$$

(T-ERROR):
$$\frac{\Delta \vdash \mathtt{T} \text{ OK}}{\Delta; P \vdash \mathtt{error} : \mathtt{T}}$$

(T-NULL):
$$\frac{\Delta \vdash \mathtt{T} \text{ OK}}{\Delta; P \vdash \mathtt{null} : \mathtt{T}}$$

$$\frac{}{\Delta; P \vdash \mathtt{x} : \Delta(\mathtt{x})}\ \text{(T-VAR)} \qquad \frac{}{\Delta; P \vdash l : \Delta(l)}\ \text{(T-LOC)}$$

Figure 11: FGO Expression Typing

(V-TYPE):
$$visible_\Delta(\mathtt{T}, \mathtt{C}) = visible_\Delta(owner_\Delta(\mathtt{T}), \mathtt{C})$$
(V-OWNER):
$$visible_\Delta(\mathtt{O}, \mathtt{C}) = \mathtt{O} \in owners(\mathtt{C}) \cup \{\mathtt{This}, \pi_\mathtt{C}, \mathtt{World}\}$$

where $owners(\mathtt{C}) =$
$$\begin{cases} \{owner_\Delta(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}, \mathtt{N}\}, \\ \quad \text{if } CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\{\ldots\} \\ \{\mathtt{X}^0\} \cup \{owner_\Delta(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}\}, \\ \quad \text{if } CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}}, \mathtt{X}^0 \lhd \mathtt{N}^0 > \lhd \mathtt{N}\{\ldots\} \end{cases}$$

Figure 12: FGO Type and Owner Visibility

Additionally, FGO allows class declarations to use implicit place-holder owner parameters in formal type parameter bounds, e.g.:

```
class List<E extends Foo<FO>,
           Owner extends World> { ... }
```

Here, List's formal parameter E can only be bound by actual type parameters that are subclasses of Foo; the owner of that type parameter (FO) can be different from the owner of the list (Owner).

Method Typing (FGO-METHOD):

$$\Delta \vdash \forall \mathtt{P}' \in \overline{\mathtt{P}} : (\mathtt{P}' \ \mathtt{OK}) \vee (\mathtt{P}' \prec \mathtt{World})$$
$$\Delta = \Delta \cup \{\overline{\mathtt{Y}} \prec \overline{\mathtt{P}}\} \qquad \Delta = \Delta \cup placeholderowners_\Delta(\overline{\mathtt{N}})$$
$$\boxed{\forall \mathtt{P}' \in \overline{\mathtt{P}} : \Delta = \Delta \cup \{\mathtt{C}^0 \prec owner_\Delta(\mathtt{P}')\}}$$
$$\Delta \vdash \overline{\mathtt{T}}, \mathtt{T} \ \mathtt{OK} \qquad \mathtt{class} \ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \ \lhd \ \mathtt{N} \ \{\dots\}$$
$$visible_\Delta(\overline{\mathtt{T}}, \mathtt{C}) \qquad visible_\Delta(\mathtt{T}, \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{P}}, \mathtt{C})$$
$$\Delta, \ \overline{\mathtt{x}} : \overline{\mathtt{T}}, \ \mathtt{this} : \mathtt{C} < \overline{\mathtt{X}} > ; \mathtt{C} \vdash visible(\mathtt{e}_0)$$
$$\Delta, \ \overline{\mathtt{x}} : \overline{\mathtt{T}}, \ \mathtt{this} : \mathtt{C} < \overline{\mathtt{X}} > ; \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}$$
$$\frac{\Delta \vdash \mathtt{S} \prec \mathtt{T} \qquad override(\mathtt{m}, \mathtt{N}, \ < \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{T}} \to \mathtt{T})}{\Delta \vdash < \overline{\mathtt{Y}} \ \lhd \ \overline{\mathtt{P}} > \ \mathtt{T} \ \mathtt{m}(\overline{\mathtt{T}} \ \overline{\mathtt{x}}) \{ \ \mathtt{return} \ \mathtt{e}_0; \ \} \ \mathtt{FGO} \ \mathtt{IN} \ \mathtt{C}, \ \mathtt{C}^0}$$

Class Typing (FGO-CLASS-MANIFEST):

$$\Delta = \{\overline{\mathtt{X}} \prec \overline{\mathtt{N}}\} \qquad \Delta = \Delta \cup placeholderowners_\Delta(\overline{\mathtt{N}})$$
$$\boxed{\forall \mathtt{N}' \in \overline{\mathtt{N}} : \Delta = \Delta \cup \{owner_\Delta(\mathtt{N}) \prec owner_\Delta(\mathtt{N}')\}}$$
$$\Delta \vdash \overline{\mathtt{M}} \ \mathtt{FGO} \ \mathtt{IN} \ \mathtt{C}, \ owner_\Delta(\mathtt{N}) \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}, \overline{\mathtt{N}} \ \mathtt{OK}$$
$$\frac{visible_\Delta(\mathtt{N}, \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{T}}, \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{N}}, \mathtt{C})}{\mathtt{class} \ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N} \ \{\overline{\mathtt{T}} \ \overline{\mathtt{f}}; \ \overline{\mathtt{M}}\} \ \mathtt{FGO}}$$

Class Typing (FGO-CLASS-PURE):

$$\mathtt{N} = \mathtt{D} < \overline{\mathtt{T}'}, \mathtt{X}^0 > \qquad \forall \mathtt{N}' \in \overline{\mathtt{N}} : \Delta \vdash \mathtt{N}' \ \mathtt{OK} \vee \Delta \vdash \mathtt{N}' \prec \mathtt{World}$$
$$\Delta = \{\overline{\mathtt{X}} \prec \overline{\mathtt{N}}, \mathtt{X}^0 \prec \mathtt{N}^0\} \qquad \Delta = \Delta \cup placeholderowners_\Delta(\overline{\mathtt{N}})$$
$$\boxed{\forall \mathtt{N}' \in \overline{\mathtt{N}} : \Delta = \Delta \cup \{\mathtt{N}^0 \prec owner_\Delta(\mathtt{N}')\}}$$
$$\Delta \vdash \overline{\mathtt{M}} \ \mathtt{FGO} \ \mathtt{IN} \ \mathtt{C}, \ \mathtt{N}^0 \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}} \ \mathtt{OK}$$
$$\frac{visible_\Delta(\mathtt{N}^0, \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{T}}, \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{N}}, \mathtt{C})}{\mathtt{class} \ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}}, \ \mathtt{X}^0 \lhd \mathtt{N}^0 > \lhd \mathtt{N} \ \{\overline{\mathtt{T}} \ \overline{\mathtt{f}}; \ \overline{\mathtt{M}}\} \ \mathtt{FGO}}$$

Placeholder Owners Function:

$$placeholderowners_\Delta(\mathtt{C} < \overline{\mathtt{T}} >) \ = \ \{owner_\Delta(\mathtt{C} < \overline{\mathtt{T}} >) \prec \mathtt{World}\} \cup$$
$$\cup \ placeholderowners_\Delta(\overline{\mathtt{T}})$$
$$\text{if} \ owner_\Delta(\mathtt{C} < \overline{\mathtt{T}} >) \notin dom(\Delta)$$
$$placeholderowners_\Delta(\mathtt{C} < \overline{\mathtt{T}} >) \ = \ placeholderowners_\Delta(\overline{\mathtt{T}})$$
$$\text{otherwise}$$
$$placeholderowners_\Delta(\mathtt{X}) \ = \ \{\}$$

Figure 13: FGO Method and Class Typing with Placeholder Owners Function

Since FGO (like FGJ) requires every type variable to be bound, we need to make sure that our environment contains an appropriate mapping for implicit placeholder parameters like FO. A function *placeholderowners* in Figure 13 does exactly that, by making sure that any (placeholder) owner used in the type bounds is recorded in the FGO environment Δ as being a subtype of World. This ensures that every owner present in the class declaration is bound, whether implicitly or explicitly.

The method typing rule checks that all the types involved are *well formed* FGO types that are *visible* within the class that contains the method. It also recursively checks the method's expression to ensure that all the subexpressions are *visible* with respect to the current class. Finally, in exactly the same manner as FGJ, the validity of method overriding (if applicable) is verified.

5. FGO DYNAMIC SEMANTICS

This section addresses the dynamic aspects of FGO. The store typing rules shown in Figure 14 are standard [16, 2]. The mapping Δ contains the types for each location and the FGO-STORE-WF rule in Figure 14 ensures that every one of the types is well-formed. The mapping S contains the type instantiations with locations for each field. The main FGO-STORE rule ensures that not only the types are well-formed, but also that each field location is valid and is a correct

Store Well-Formedness (FGO-STORE-WF):

$$\frac{\forall l \in dom(\Delta) : \Delta \vdash \Delta(l) \ \mathtt{OK}}{\Delta \ \mathtt{OK}}$$

Store Typing (FGO-STORE):

$$\Delta \ \mathtt{OK} \qquad dom(\Delta) = dom(S) \qquad S[l] = \mathtt{N}(\overline{v}) \Longleftrightarrow \Delta(l) = \mathtt{N}$$
$$(S[l, i] = l') \wedge (fields(\Delta(l)) = \overline{\mathtt{T}} \ \mathtt{f}) \Longrightarrow$$
$$\Longrightarrow \Delta \vdash \Delta(l') \prec [\mathtt{This}_l/\mathtt{This}]\mathtt{T}_i$$
$$\frac{(S[l, \ i] = l') \Longrightarrow \Delta \vdash \Delta(l') \ \mathtt{OK}}{\Delta \vdash S}$$

Figure 14: FGO Store

subtype of the declared field type. It is interesting to note that our type system does not have any explicit ownership constraints in the store rule — the benefit of ownership information being part of the type is that subtyping ensures that none of the ownership constraints are broken.

Figure 15 shows the reduction rules. Again, these are standard given the expressions that FGO supports. The context reduction rules are *omitted* from this paper for brevity, but are present in the accompanying technical report[5].

5.1 Soundness

In this subsection I present manual proofs of the FGO type soundness. It is possible to create a machine-assisted proof of type soundness. The machine-assisted proofs catch any omissions that can happen when hand writing a tedious proof that goes through all the possible cases. It is part of the future work on Generic Ownership to convert these proofs to a machine-checkable format.

Theorem (Preservation). *If* $\Delta; P \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta \vdash S$ *and* $\mathtt{e}, S \to \mathtt{e}', S'$, *then* $\exists \Delta' \supseteq \Delta$ *and* $\exists \mathtt{T}' \prec \mathtt{T}$ *such that* $\Delta; P \vdash \mathtt{e}' : \mathtt{T}'$ *and* $\Delta' \vdash S'$.

Proof. Using structural induction on reduction rules in Figure 15. Present in complete detail in the accompanying technical report. ∎

Theorem (Progress). *Suppose* $\mathtt{e}$ *is a closed well-typed expression. Then either* $\mathtt{e}$ *is a value or there is a reduction rule that contains* $\mathtt{e}$ *on the left hand side.*

Proof. Based on all the possible expression types, either $\mathtt{e}$ is a value (see the last four rules in Figure 11) or one of the reduction rules applies. It can't be a variable because it is closed. We need to check that each of the reduction rules is satisfied. The only rules that require additional conditions are R-FIELD, R-FIELD-SET, and R-METHOD — in the case of R-BAD-CAST the program reduces to errorif the downcast is impossible.

In case of R-FIELD and R-FIELD-SET well-typedness of N ensures that $fields(\mathtt{N})$ is well defined and $\mathtt{f}_i$ appears in it. In case of R-METHOD, the fact the $mtype$ looks up the type for m, ensures that $mbody$ will succeed too and will have the same number of arguments (since MT-CLASS and MB-CLASS are defined in the same way).

In case of $l = \mathtt{null}$, one of R-FIELD-NULL, R-METHOD-NULL, and R-FIELD-SET-NULL will ensure that we reduce to error. ∎

Type soundness is then immediate from preservation and progress theorems. Note that this type soundness result proves only the absence of "ordinary" type errors. The next section discusses the ownership guarantees provided by FGJ.

[5]http://www.mcs.vuw.ac.nz/~alex/files/FGOTR.pdf

(R-NEW):
$$\frac{l \notin dom(S) \quad S' = S[l \mapsto \overline{\texttt{null}}] \quad |\overline{\texttt{null}}| = |fields(\texttt{N})|}{\texttt{new N}(), S \to l, S'}$$

(R-FIELD):
$$\frac{S[l] = \texttt{N}(\overline{v}) \quad fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{l.\texttt{f}_i, S \to v_i, S}$$

(R-FIELD-SET):
$$\frac{S[l] = \texttt{N}(\overline{v}) \quad fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ S' = S[l \mapsto [v/v_i]\texttt{N}(\overline{v})]}{l.\texttt{f}_i = v, S \to v, S'}$$

(R-METHOD):
$$\frac{S[l] = \texttt{N}(\overline{v_l}) \quad mbody(\texttt{m}<\overline{\texttt{V}}>, \texttt{N}) = \overline{\texttt{x}}.\texttt{e}_0}{l.\texttt{m}<\overline{\texttt{V}}>(\overline{v}), S \to l > [\overline{v}/\overline{\texttt{x}}, l/\texttt{this}, \texttt{This}_l/\texttt{This}]\texttt{e}_0, S}$$

$$\frac{S[l] = \texttt{N}(\overline{v}) \quad \texttt{N} \lessdot \texttt{P}}{(\texttt{P})l, S \to l, S} \text{ (R-CAST)}$$

$$\frac{S[l] = \texttt{N}(\overline{v}) \quad \texttt{N} \nleqslant: \texttt{P}}{(\texttt{P})l, S \to \texttt{error}, S} \text{ (R-BAD-CAST)}$$

$$\frac{}{l > v, S \to v, S} \text{ (R-CONTEXT)}$$

$$\frac{}{\texttt{let x } = v \texttt{ in } \texttt{e}_0, S \to [v/\texttt{x}]\texttt{e}_0, S} \text{ (R-LET)}$$

(R-*-NULL):

$$\frac{}{\texttt{null.m}<\overline{\texttt{V}}>(\overline{v}), S \to \texttt{error}, S}$$

$$\frac{}{\texttt{null.f}_i, S \to \texttt{error}, S} \qquad \frac{}{\texttt{null.f}_i = v, S \to \texttt{error}, S}$$

Figure 15: FGO Reduction Rules

6. OWNERSHIP GUARANTEES

Lemma (Ownership Invariance). *If* $\Delta \vdash S \lessdot T$ *and* $\Delta \vdash T \lessdot \texttt{Object}<\texttt{O}>$, *then* $owner_\Delta(S) = owner_\Delta(T) = \texttt{O}$.

Proof. By induction on the depth of the subtype hierarchy. By FGO class typing rules a FGO class has the same owner parameter as its superclass. ∎

6.1 Ownership Invariant

We define *inside* ($\prec$) relationship on owner classes for objects (e.g. $\texttt{This}_l$) in the same manner as classic ownership type papers [17, 9]. During the execution of any FGO program with deep ownership, if an object at l refers to object at l', then owner class $\texttt{This}_l$ corresponding to the object at l is *inside* ($\prec$) owner class corresponding to l' ($owner(\Delta(l'))$).

At class declaration validation time, the $\prec$ relationship for the owner classes is as follows: $\texttt{This} \prec \texttt{Owner} \prec \texttt{World}$ (note that our owner classes' subtyping relationship is along the same lines: $\texttt{This}_l <: \texttt{This} <: \texttt{World}$). During the reduction, both $\texttt{Owner}$ and $\texttt{This}$ will have appropriate location-specific owners (e.g. $\texttt{This}_l$) substituted for them. This allows us to prove a deep ownership invariant similar to that of Clarke (and as used by Boyapati):

Definition (refers to). Object at location l *refers to* object at location l' if and only if (1) $\Delta(l) = \texttt{N}(\overline{l})$ and $l' \in \overline{l}$; or (2) for some Δ, P we have $\Delta; P \vdash l > \texttt{e} : \texttt{T}$ and l' occurs as one of the subexpressions of $\texttt{e}$.

Definition (inside). Owner class $\texttt{T}$ is *inside* ($\prec$) owner class $\texttt{T}'$ if and only if $\Delta \vdash \texttt{T} \prec \texttt{T}' \prec \texttt{World}$.

Theorem (Ownership Invariant). *l refers to l' only if* $\texttt{This}_l \prec owner_\Delta(l')$ *or* $(owner_\Delta(l') = \pi_\texttt{C}$ *and* $visible_\Delta(\pi_\texttt{C}, \Delta(l)))$.

Proof. $\Delta(l') \lessdot [\texttt{This}/\texttt{This}_l]\ \texttt{T}_i$. If owner is $\texttt{World}$ or $\texttt{This}$, then we are OK by definition of $\prec$. If owner is anything else then since well-formedness preserves owner class nesting and $\texttt{This} \prec \texttt{Owner} \prec \texttt{O}$ (where $\texttt{O}$ is the set of owners of type parameters) holds, we have $\texttt{This}_l \prec \texttt{This}_{l'}$. The second part of the proof holds due to the confinement invariant (below). ∎

Theorem (Confinement Invariant). *Let $\texttt{e}$ be a subexpression appearing in the body of a method of a well-formed FGO class $\texttt{C}$ during program execution.*
Then: If $\texttt{e} \to^* \texttt{new D}<\overline{\texttt{T}_\texttt{D}}>(\overline{\texttt{e}})$, then $visible_\Delta(\texttt{D}<\overline{\texttt{T}_\texttt{D}}>, \texttt{C})$.

Proof. Because the class is a well-formed FGO class, its methods are well-formed FGO methods. This, plus the standard subformula property, implies that, for appropriate $\Delta; P$: both $\Delta; P \vdash \texttt{e} : \texttt{T}$ and $\Delta; \texttt{C} \vdash visible(\texttt{e})$ hold. From this we can derive $visible_\Delta(\texttt{T}, \texttt{C})$, and hence $visible_\Delta(owner_\Delta(\texttt{T}), \texttt{C})$. By FGO's subject reduction property, there is a $\texttt{T}'$ such that $\Delta; P \vdash \texttt{new D}<\overline{\texttt{T}_\texttt{D}}>(\overline{\texttt{e}}) : \texttt{T}'$, where $\Delta \vdash \texttt{T}' \lessdot \texttt{T}$. Furthermore, we have that $\Delta; P \vdash \texttt{new D}<\overline{\texttt{T}_\texttt{D}}>(\overline{\texttt{e}}) : \texttt{D}<\overline{\texttt{T}_\texttt{D}}>$, and hence clearly $\Delta \vdash \texttt{D}<\overline{\texttt{T}_\texttt{D}}> \lessdot \texttt{T}'$, and $\Delta \vdash \texttt{D}<\overline{\texttt{T}_\texttt{D}}> \lessdot \texttt{T}$. By the Ownership Invariance lemma, $owner_\Delta(\texttt{D}<\overline{\texttt{T}_\texttt{D}}>) = owner_\Delta(\texttt{T})$, from which we deduce $visible_\Delta(owner_\Delta(\texttt{D}<\overline{\texttt{T}_\texttt{D}}>), \texttt{C})$, and hence $visible_\Delta(\texttt{D}<\overline{\texttt{T}_\texttt{D}}>, \texttt{C})$. ∎

The major difference between our generic ownership proofs, and more traditional non-type-generic ownership invariant proofs lies in a much simpler formulation. The key benefit comes from integrating ownership into a parametric polymorphic type system, rather than building an ownership-parametric type system on top of a non-generic typed language.

7. DISCUSSION AND RELATED WORK

Object ownership can address aliasing, security, concurrency, and memory management problems, while smoothly aligning with typical object-oriented program designs. Systems using object ownership range from expressive but weighty *explicit* systems based on ownership types [17] to lightweight but limited *implicit* systems based on confined types [52].

Explicit systems such as AliasJava, Universes, and the systems of Boyapati et.al. [3, 2, 9, 38], differ primarily in one characteristic that Clarke and Wrigstad define as *shallow* v.s. *deep* ownership [19]. Deep ownership protects transitively nested objects, while shallow ownership does not. Clarke and Drossopoulou [16] and Boyapati et al. [9] describe how to exploit the strong protection provided by deep ownership. Generic Ownership is an explicit deep ownership system: however its key contribution is that it combines type genericity and object ownership into a single system.

The strong, transitive protection deep ownership provides is also a liability, because the deep protection prevents programmers from accessing objects' internal structures, and can require inefficient coding idioms to move data across objects' interfaces. Aldrich and Chambers' AliasJava and Boyapati et al.'s SafeJava [2, 9] show how ownership types can support more flexible object graph topologies: we do not address this issue in this paper. Inspired by OGJ, Alias-Java adopted Generic Ownership style parameters [2] to provide both type and ownership genericity, although without a type generic formal system: here we establish that this approach is sound. Krishnaswami and Aldrich have recently formalised an extension to Ownership Domains using System F and permission-based ownership [30], illustrating a range of flexible ownership topologies, and supporting strong encapsulation via ML-style abstract types and genericity.

Other recent research has extended the applicability of ownership-style schemes. Lu and Potter have shown how to ensure object references are acyclic [33], or to control field updates even when references are unconstrained [35]: they also employ ownership wildcards. They later extended their work to owner variance [34]. Boyland and Retert have described how various forms of uniqueness and ownership can be encoded using fractional permissions [11, 12]. None of these proposals are type-generic: however, we expect that, like AliasJava, languages designed with these systems could take advantage of generic ownership to provide both type and owner polymorphism.

Unlike some other ownership type schemes [8] FGO 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, FGO's genericity should remove the need for many of these downcasts.

All these explicit systems require additional annotations to use them, raising issues about their role in programming. For this reason, Aldrich et al. and Boyapati et al. have described a range of type inference schemes to provide these annotations automatically [3, 10, 7, 8]. Generic ownership enables a simpler approach, as many of the owner parameters that have to be inferred by other schemes are already present in type-generic code.

Implicit confined type systems have achieved their more limited goals while keeping the number of annotations low. Vitek and Bokowski's original system [52] required classes to be annotated as confined, while Clarke, Richmond and Noble [18] apply these ideas in the context of Enterprise Java Beans. More recent work by Zhao, Palsberg and Vitek [53] has formalised confined types based on Featherweight Java. Confined Featherweight Java also includes a notion of generic confined types, allowing e.g. a collection to be confined or not depending upon the specifications of the contained elements. Our 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 (Generic Java, and its formal core FGJ), and then ensure ownership and confinement directly. Our approach leads to a simpler formal system requiring fewer new concepts and a distinctly simpler and shorter proofs.

Clarke's thesis was the first account of a system with both parametric polymorphism and ownership [15]. This system was based on Abadi and Cardelli's object calculus [1], rather than a 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 class-based language (with inner classes), but does not provide a generic type system or language design.

Ownership types are similar to region types and region polymorphism [49, 50], but serve quite different purposes: ownership accounts for encapsulation while regions manage memory allocation. Regions are restricted to stack-based memory allocation while ownership supports long-lived objects and much less restrictive topologies. Just as region polymorphism permits functions to be applied to arguments in different regions, so generic ownership polymorphism allows a class to be instantiated to handle arguments with different ownership — the actual ownership type arguments act as permissions allowing the generic class access the arguments with those types.

Regions and other systems use Hindley-Milner style type inference to make fine distinctions on potential aliasing [4, 40, 31].

Type-based alias analysis uses the class hierarchy to discover non-alias conditions [20]. More generally, functional programmers use a related technique called "phantom types" [21] to include a wide range of information within types. Generic ownership is similar to these approaches in that it, too, uses additional type parameters to carry ownership information, while paying a minimal syntactic cost.

Finally, John Potter[6] has suggested that owners could be modeled as an orthogonal kind to class types: in such a scheme, types would be a pair comprising an owner context and a class type (e.g. T ::= C@O, where @ binds a type and an owner context). Cyclone, X10, and F_{own} [22, 14, 30] types have similar structures. This approach has the advantage of keeping owners and types conceptually separate, and, if type variables can range over such pairs, providing much of the polymorphism of generic ownership. Compared with our model these pairs need more language support — a separate kind of owners, and then constructors and accessors to retrieve owners from type pairs. This would be harder to incorporate into a practical programming language than generic ownership (X10 and Cyclone are new designs) and also harder to take advantage of new language features. In particular, our deep ownership type formation constraints are very similar to GADT type constraints [28] although we require subtyping rather than type equality. Based on our model of generic ownership, we expect that a language with GADT's and subtype constraints may be strong enough to express ownership directly in its type system.

8. FUTURE WORK AND CONCLUSION

In this paper we described and formalised Generic Ownership — a single system that encompasses both generic types and deep, per-object ownership. The goal of Generic Ownership is not to provide a novel abstraction combining ownership and genericity. Rather we are aiming to provide a practical way of integrating ownership into modern object-oriented languages (like Java or C#) that already employ genericity. Generic Ownership reuses such generic language constructs as much as possible to provide ownership. Our language design and formalism show that ownership and generic type information can be expressed within a single system, and carried around the program as bindings to the same parameters. As a result, programs using Generic Ownership are only slightly more complex than those using just generic types, yet enjoy the full protection provided by ownership types.

We have demonstrated the practicability of Generic Ownership with Ownership Generic Java, a seamless, syntactically compatible extension to generic-capable Java 5 and we have implemented an OGJ language as an extension to Java 5 [43].

In the future, we plan to utilise the formal foundations provided by FGO to work out proof principles of ownership guarantees, and develop a set of design patterns for OGJ for programmers wishing to make use of ownership in their programs.

To summarise, our contributions are as follows: Generic Ownership, a seamless integration of genericity and ownership; Featherweight Generic Ownership, a formal model allowing the proof of soundness, confinement, and deep ownership invariants; and Ownership Generic Java combining generic types, ownership, and confinement in a single, straightforward language design.

Acknowledgments

---

9. REFERENCES

[1] ABADI, M., AND CARDELLI, L. *A Theory of Objects.* Springer-Verlag, Berlin, Heidelberg, Germany, 1996.

[2] ALDRICH, J., AND CHAMBERS, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (Oslo, Norway, June 2004), vol. 3086, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 1–25.

[3] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias Annotations for Program Understanding. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Seattle, WA, USA, Nov. 2002), ACM Press, New York, NY, USA, pp. 311–330.

[4] 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.

[5] 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 (FTfJP)* (Darmstadt, Germany, July 2003), Springer-Verlag, Berlin, Heidelberg, Germany.

[6] BOYAPATI, C. *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, EECS, MIT, February 2004.

[7] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (November 2002).

[8] BOYAPATI, C., LEE, R., AND RINARD, M. Safe runtime downcasts with ownership types. In *Proceedings of International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, D. Clarke, Ed. Utrecht University, July 2003, pp. 1 – 14.

[9] BOYAPATI, C., LISKOV, B., AND SHRIRA, L. Ownership Types for Object Encapsulation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)* (New Orleans, LA, USA, Jan. 2003), ACM Press, New York, NY, USA, pp. 213–223. Invited talk by Barbara Liskov.

[10] BOYAPATI, C., AND RINARD, M. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Tampa Bay, FL, USA, 2001), ACM Press, New York, NY, USA, pp. 56–69.

[11] BOYLAND, J. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium* (2003), no. 2694 in Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, Germany, pp. 55–72.

[12] BOYLAND, J. T., AND RETER, W. Connecting effects and uniqueness with adoption. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)* (2005).

[13] BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. Making the future safe for the past: Adding Genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 1998).

[14] CHARLES, P., DONAWA, C., EBCIOGLU, K., GROTHOFF, C., KIELSTRA, A., SARKAR, V., AND PRAUN, C. V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005).

[15] CLARKE, D. *Object Ownership and Containment.* PhD thesis, School of CSE, UNSW, Australia, 2002.

[16] CLARKE, D., AND DROSSOPOULOU, S. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Seattle, WA, USA, Nov. 2002), ACM Press, New York, NY, USA, pp. 292–310.

[17] CLARKE, D., POTTER, J., AND NOBLE, J. Ownership Types for Flexible Alias Protection. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Vancouver, Canada, Oct. 1998), ACM Press, New York, NY, USA, pp. 48–64.

[18] CLARKE, D., RICHMOND, M., AND NOBLE, J. Saving the World from Bad Beans: Deployment-Time Confinement Checking. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Anaheim, CA, 2003), ACM Press, New York, NY, USA, pp. 374–387.

[19] CLARKE, D., AND WRIGSTAD, T. External Uniqueness is Unique Enough. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (Darmstadt, Germany, July 2003), vol. 2473 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 176–200.

[20] DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 1998).

[21] FLUET, M., AND PUCELLA, R. Phantom Types and Subtyping. In *International Conference on Theoretical Computer Science (TCS)* (Aug. 2002), pp. 448–460.

[22] GROSSMAN, D., MORRISETT, J. G., JIM, T., HICKS, M. W., WANG, Y., AND CHENEY, J. Region-based memory management in cyclone. In *PLDI* (2002), pp. 282–293.

[23] GROTHOFF, C., PALSBERG, J., AND VITEK, J. Encapsulating Objects with Confined Types. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Tampa Bay, FL, USA, 2001), ACM Press, New York, NY, USA, pp. 241–255.

[24] HOGG, J. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Phoenix, AZ, USA, Nov. 1991), vol. 26, ACM Press, New York, NY, USA, 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. C., AND WADLER, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23*, 3 (May 2001), 396–450.

[27] IGARASHI, A., PIERCE, B. C., AND WADLER, P. A recipe for raw types. In *Proceedings of Workshop on Foundations of*

Object-Oriented Languages (FOOL) (2001).

[28] KENNEDY, A., AND RUSSO, C. Generalized algebraic data types and object-oriented programming. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005).

[29] KENNEDY, A., AND SYME, D. The design and implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2001).

[30] KRISHNASWAMI, N., AND ALDRICH, J. Permission-based ownership: Encapsulating state in higher-order typed languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Chicago, UL, USA, 2005), ACM Press, New York, NY, USA, pp. 96–106.

[31] LAMPORT, L., AND SCHNEIDER, F. B. Constraints: A uniform approach to aliasing and typing. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)* (New Orleans, Louisiana, 1985), pp. 205–216.

[32] LEINO, K. R. M., AND MULLER, P. Object invariants in dynamic contexts. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2004), Springer-Verlag, Berlin, Heidelberg, Germany.

[33] LU, Y., AND POTTER, J. A type system for reachability and acyclicity. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2005), vol. 3586 of *Lecture Notes in Computer Science*, Springer, pp. 479–503.

[34] LU, Y., AND POTTER, J. Flexible ownership types with owner variance. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2006).

[35] LU, Y., AND POTTER, J. Protecting representation with effect encapsulation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)* (2006).

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

[37] MITCHELL, N. The runtime structure of object ownership. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (Nantes, France, July 2006), D. Thomas, Ed., vol. 4067 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 74–98.

[38] MÜLLER, P., AND POETZSCH-HEFFTER, A. *Programming Languages and Fundamentals of Programming.* Fernuniversität Hagen, 1999, ch. Universes: a Type System for Controlling Representation Exposure. Poetzsch-Heffter, A. and Meyer, J. (editors).

[39] NOBLE, J., VITEK, J., AND POTTER, J. Flexible Alias Protection. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (July 1998), E. Jul, Ed., vol. 1445 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 158–185.

[40] O'CALLAHAN, R., AND JACKSON, D. Lackwit: a program understanding tool based on type inference. In *Proceedings of the International Conference on Software Engineering (ICSE)* (Boston, USA, May 1997).

[41] PERMANDLA, P., AND BOYAPATI, C. A type system for preventing data races and deadlocks in the java virtual machine language. Tech. rep., University of Michigan, 2005.

[42] PIERCE, B. C. *Types and Programming Languages.* MIT Press, 2002.

[43] POTANIN, A. Ownership Generic Java Download. `http://www.mcs.vuw.ac.nz/˜alex/ogj/`, 2005.

[44] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Defaulting Generic Java to Ownership. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)* (Oslo, Norway, June 2004), Springer-Verlag, Berlin, Heidelberg, Germany.

[45] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Featherweight Generic Ownership. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)* (Glasgow, Scotland, July 2005), Springer-Verlag, Berlin, Heidelberg, Germany.

[46] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Featherweight Generic Confinement. *Journal of Functional Programming* (2006). Accepted for publication.

[47] PUGH, B. Find Bugs — A Bug Pattern Detector for Java. `www.cs.umd.edu/˜pugh/jva/bugs/`, 2003.

[48] SUN MICROSYSTEMS. Java Development Kit. Available at: `http://java.sun.com/j2se/`, 2005.

[49] TALPIN, J.-P., AND JOUVELOT, P. Polymorphic type, region, and effect inference. *Journal of Functional Programming 2*, 3 (July 1992), 245–271.

[50] TOFTE, M., AND TALPIN, J.-P. Region-Based Memory Management. *Information and Computation 132*, 2 (1997), 109–176.

[51] TORGERSON, M., ERNST, E., HANSEN, C. P., VON DER AHÉ, P., BRACHA, G., AND GAFTER, N. Adding wildcards to the Java programming language. *Journal of Object Technology 3*, 11 (Dec. 2004), 97–116. Special Issue: OOPS track at SAC 2004.

[52] VITEK, J., AND BOKOWSKI, B. Confined Types in Java. *Software Practice & Experience 31*, 6 (May 2001), 507–532.

[53] ZHAO, T., PALSBERG, J., AND VITEK, J. Type-Based Confinement. *Journal of Functional Programming 16*, 1 (2006), 83–128.