Ownership Meets Java

Christo Fogelberg
Victoria University of Wellington
cgf.vicmail@syntilect.com

Alex Potanin
Victoria University of Wellington
alex@mcs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@mcs.vuw.ac.nz

Abstract

Ownership Generic Java (OGJ) is a language with ownership types as an extension to Java. In this position paper we outline the state of OGJ. We hope that the other aliasing and ownership researchers would benefit from the discussion around how to add ownership into a modern generic and annotation-capable typed object-oriented language like Java.

1. Introduction

With the lively state of ownership research [1, 3, 4, 8] a question comes up: “What stops us from adding ownership to Java today?” This position paper claims that there is nothing substantial which stops us from starting to use ownership today. The only problem is how it can be presented to the programming community to promote its usefulness.

OGJ [7] is a language with deep, reference-based ownership support. Over the recent months we have been working on resolving the remaining issues which arise when ownership and Java meet in an actual language implementation. Section 2 outlines these by dealing with statics, exceptions [5], arrays, equals and clone methods, and wildcards. The problem of arrays is a consequence of Java language design. The other problems are deeper issues which face many ownership systems, and we discuss them in the context of OGJ in this paper. Section 3 wonders what else we need to do before we could propose an ownership extension to Java (ultimately as a JSR).

The solutions described in this paper are in no way designed to be definitive, rather we pose a question to the community as to what could be the best alternatives to solving these issues.

2. Ownership Meets Java

2.1 Statics

Because static members cannot be owned by any instances or instance-associated ownership types it means that the possible owners are limited to `World`, `Package`, and `Class`. We propose that public static members be implicitly owned by `World`, that package-private static members be implicitly owned by `Package` and that protected and private static members be implicitly owned by `Class`.

Each of these approaches is, superficially, fairly straightforward, however, in the case of inheritance complexities can arise. This is because a public or protected static member of some class `C` which is owned by <`Class`> is not visible to any subclass of `C`, according to the original definition of OGJ.

We propose to modify the visibility rules in OGJ to take into account the inheritance hierarchy, in exactly the same way as the object typing rules do. For example, if `D extends C`, then `C.foo<Class>` would be visible and could be referred to by any instance of type `D`.

```
class Super<SuperOwner extends World> {
  public static Integer pub = 1; // assumes World
  static Integer pac = 2; // assumes Package
  protected static Integer pro = 3; // assumes Class
  private static Integer pri = 4; // assumes Class

  // Cannot use the instance type parameters or This
  // static String<SuperOwner> s = "illegal!";
}

class Sub<SubOwner extends World>
  extends Super<SubOwner> {

  public static String<Class>
    weird1 = "legal,␣but␣strange!";
  private static String<World>
    weird2 = "legal,␣but␣strange!";

  void DoFoo() {
    // Legal, because we made Class visible
    // to subclasses
    Sub.pro = 99;
  }
}
```

Figure 1. Static members and ownership.

Additionally, the presence of owner generic methods in OGJ allows for static methods to have additional owners supplied via generic parameters.

Figure 1 shows an example of statics in OGJ. The static fields gain an implicit owner based on their visibility. An explicit owner can be used as long as it doesn’t contradict the name visibility of the static field.

2.2 Exceptions

Incorporating exceptions into an ownership type system raises two main problems: (1) the ownership of the exception itself, (2) and the possibility of leaking references. Ownership and exceptions were also addressed by Werner Dietl and Peter Müller [5].

An object throwing an exception could specify either that its owner was a global type like `World`, or that it was owned by one of the ownership types which were in scope. Specifying `World` would mean that non-`World` objects could not be assigned as fields to the reference. Similarly, using one of the other visible ownership types would mean that exceptions could not propagate very far. Thus exceptions would be either greatly weakened or method calls would be unduly restricted by static type checking. This would mean that many design idioms could not be expressed.

For OGJ we have chosen to introduce a special owner `Exception` (that is a subtype of `World`) to resolve this problem. All exceptions are owned by `Exception`. `Exception` is just like `World`, in that any object can refer to or create an object of type `Exception`. Using

```
class FooException <EOwner extends Exception>
  extends Throwable {

  public Foo<EOwner> causeOfException;

  public FooException(
    Foo<EOwner> causeOfException) {

    this.causeOfException = causeOfException;
  }
}

class A<Owner extends World>() {
  private Foo<This> f;

  public void SomeMethod() {
    throw new FooException<Exception>(f);
  }
}

class B<Owner extends World> {
  public static Foo<World> globalF;
  private Foo<This> myF;

  public void <AnotherOwner extends World>
    DoSomething(A<AnotherOwner> a) {

    Foo<This> localF;
    try {
      a.SomeMethod();
    } catch(FooException<Exception> e) {
      e.causeOfException.fix();

      // These lines cause compile-time errors:
      // B.globalF = e.causeOfException;
      // this.myF = e.causeOfException;
      // localF = e.causeOfException;
    }
  }
}
```

Figure 2. Exception handling and ownership.

a subtype of World marks off the relevant subset of global objects and allows them to be treated slightly differently.

In particular, we adopt the following conditions for Exception. Firstly, an object owned by Exception can only be created as part of a throw statement. Secondly we allow the ownership typing rules to be briefly broken during the exception throwing, so long as all other typing rules are not. This means that any object can be passed as a parameter to and used in the constructor of an Exception. In this situation the language implementation will only emit a warning saying that ownership will have been temporarily and locally broken.

The key advantages of this approach are as follows. Firstly, borrowing or uniqueness[6, 2] do not need to be introduced into OGJ just so that exceptions can be used. This minimises the learning curve for users of the language. Secondly, some exceptions in the API already expose references. An example of this is omg.org.CORBA.portable.ApplicationException. Rewriting exceptions like this to work in an ownership environment could also mean that the underlying architecture would need to be redesigned. Our approach maximises legacy interoperability and minimises code conversion costs.

Figure 2 shows an example of exception handling in OGJ. Inside method SomeMethod we can pass a field f owned by This to a publicly owned exception only because we are doing as part of the throw statement. Inside the catch block, the exception's fields can be referenced and modified via method calls, however copying the exception fields to variable not local to the catch block will generate a compile-time error.

```
class Foo<Owner> {}

class MyArrays<Owner extends World> {
  // What the specification claims is allowed:
  public Foo<Owner>[] myOtherArray = new Foo<?>[20];

  // What javac actually allows, they are equivalent:
  public Foo<Owner>[] myArray = new Foo[20];
}
```

Figure 3. Java arrays and ownership.

2.3 Arrays

In general, arrays are not a problem for ownership, however, some design decisions do need to be made. Although much of the discussion in this subsection is concerned with the details of implementing owned arrays with type erasure in Java, it generalises nicely to typed languages with generics.

We propose that arrays not have an owner [9]. Their one owner parameter refers to both the objects in the array and to the array itself. Although it is possible to give arrays their own owner, and while arrays have some minimal functionality added to them in Java there is very little that can be done with an array that does not directly involve its elements. This is not the case with collections. In our experience the added complexity of giving each array its own owner distinct from its elements' was not justified by the changes that would be required in Java. In the very rare cases of separate owner being useful, the use of a collection (e.g., ArrayList with appropriate owners) was more appropriate.

We consider compatibility with old Java virtual machines to be crucial. This means that, just like generics, ownership-checked programs must run on pre-ownership and pre-generic virtual machines. At the point of array creation, due to the restrictions imposed by Java's type erasure, there are really only two options: (1) allow ownership (but not generics) to be specified in the allocation statement and the type of the reference, or (2) only allow ownership in the type of the reference. Currently, if an approach like the second were used the language implementation would raise an "unchecked warning". We would recommend not raising a warning in the case of ownership types. This is because OGJ guarantees that the references will be properly type-checked.

These approaches work because of the way in which the Java type checks its programs. Aside from checking the type of the allocation statement at the time of construction against the type of the reference, Java otherwise type checks references against each other, not against the type of the underlying object. Because all references to array objects will include owner information as part of their type, ownership remains sound with arrays.

This means that if we can just get past the hurdle of object creation, owned arrays will be handled automatically by Java's type checking rules, just as existing generic arrays are handled. Even though these parameters will be erased by the language implementation, they are always type checked first.

Figure 3 shows an example of arrays in OGJ. These approaches work because although the object itself is unowned the only reference to the object is through its reference, which is properly type checked and which is forbidden by OGJ from having its owner cast away. In addition, the implementation still does the type substitution into the class.

```
class A<EO1 extends World, EO2 extends World,
  Owner extends World> {

  Foo<EO1, This> f1;

  boolean <O> equals(OObject<O> o) {
    if (o == this) return true;
    if (!(o instanceof List))
      return false; // compare to raw type

    A<OtherEO1, OtherEO2, O> oA =
     (A<OtherEO1, OtherEO2, O>) o;

    if(!this.f1.equals(oA.f1)) return false;
    else return true;
  }
}
```

Figure 4. Object comparison and ownership.

2.4 Equals

Equals and clone methods in OGJ suffer from the problem of not being able to have the same signature (and thus be overridden in the subclasses) due to a varying set of owners required by the method to perform equals or clone operation on various objects having access to many owners [8].

Since each object only has one "main" owner and the rest are simply those additional "outside" owners that it has access to, existential owners [8] shows why it is safe to sometimes lose track of the non-main owners and then gain them back by downcasting and introducing existential ownership types distinct from any other ownership types which are visible.

For example, this allows us to implement an equals method in OGJ as shown in Figure 4. Notice that downcasting introduces new owner type variables (e.g. `OtherEO1` and `OtherEO2`) following the existential ownership proposal [8]. The downcast uses owner `O` (coming via the method parameter `o`) which is not necessarily the same as the owner of the class `A` (`Owner`).

In Figure 4 class `A` has additional owners `EO1` and `EO2` in addition to the main owner `Owner`. Its equals method is a generic method that accepts the owner parameter `O` of the object being compared to (which maybe `This` for the other object). The equals method thus has access to two (potentially unrelated and private) objects in the object graph only for the duration of the method itself. Note that the equals method has the same signature for every class even though there may be multiple owners involved.

2.5 Clone

Figure 5 illustrates cloning in OGJ. For cloning, a method call to the assignee (which may have a new owner if one would like to change ownership of the clone) is required as shown in Figure 5. We call this *assisted cloning*. A C++-like copy constructor could be used in place of the special assistant method `assignClone`. For the sake of exposition we have omitted null and self-reference checks. The code as written would fail because of this but it illustrates the assistant method approach.

2.6 Wildcard bounds in OGJ

In this section we will discuss how wildcards and bounded wildcards are taken into account in OGJ. The Java type system makes this relatively easy.

An unbounded `?` can only be used as a wildcard in limited situations. This is because it is entirely anonymous. Although at compilation the static types of actual instances will be substituted in place of this wildcard and can be checked it cannot be referred to or used by the code. For example, a class specified with `?` as its owner could not use that owner in any of its method or field definitions.

```
class A<Owner extends World> {
  A<This> f; int i;

  <NewOwner> OObject<NewOwner> clone() {
    A<NewOwner> temp = new A<NewOwner>();
    temp.i = this.i;
    temp.f.<This>assignClone(this.f);
    return temp;
  }

  <NewOwner> void assignClone(A<NewOwner> a) {
    this = (A<Owner>) a.<Owner>clone();
  }
}
```

Figure 5. Object cloning and ownership, other version

Named bounded wildcards (e.g. `Owner super Package`) simply act as statically checkable restrictions on owner types which are valid for that method parameter, class or generic method. Note that an anonymous bounded wildcard (e.g. `? extends Package`) also faces the same restrictions as an unbounded anonymous wildcard and cannot be referred to in the code.

Because only `World`, `Package`, `Class` and `This` are ownership types known to exist, only a few bounds can be specified for a class which is not an inner class. For example, usefully. a class could have its owner specified as `Owner extends Package` or `Owner super Package`. In the first case it would mean that no instance of the class could be accessible outside the `Package`. In the latter it would mean that all instances of this class must be either `Package`-visible or `World`-visible. In both cases the valid owners are restricted to one of the globally defined types. This provides region-like capabilities.

In addition, because $\texttt{This}_l$ is a subtype of the owner of the object l, inner classes can be bound to have owners which are sub or super types of their enclosing classes owners. In relationships between unnested classes which are siblings in the class hierarchy it is impossible to express bounds like "the owner of `Foo` must be a subtype of the owner of `Bar`". This is because the owner variable of `Bar` is only visible to `Foo` if `Bar` is a supertype of `Foo` or if `Foo` is an inner class of `Bar`.

3. What Next?

In this position paper we presented our design choices for five corner cases in ownership language design: static ownership matching static visibility, global `Exception` owners for exceptions, ownership of array being the same as that of array's elements, existential owners for equals, and assisted clone implementation. We also discussed bounds on ownership types.

For ownership to be successful a large number of issues still remain to be resolved in a consistend and agreed upon manner in collaboration with other aliasing language researchers. These include formalising interaction between owned and *unowned* code, the choice between effective and reference-based ownership, using implicit or explicit owner parameters, adding additional features such as immutability or external uniqueness, developing a collection of language implementations, and writing fully ownership-aware collections and libraries.

An agreed upon compromise and implementation support would attract more users to the ownership-enabled languages and help resolve any issues stopping the ownership research proposal becoming a JSR reality.

Acknowledgments

This work is supported by the Victoria University of Wellington URF Grant Number 26214.

References

[1] Cameron, N., Drossopoulou, S., Noble, J., and Smith, M. Multiple ownership. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007).

[2] 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.

[3] Clifton, C., Leavens, G. T., and Noble, J. MAO: Ownership and effects for more effective reasoning about aspects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2007).

[4] Dietl, W., Drossopoulou, S., and Müller, P. Generic universe types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2007).

[5] Dietl, W., and Müller, P. Exceptions in ownership type systems. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)* (2004).

[6] 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).

[7] Potanin, A., Noble, J., Clarke, D., and Biddle, R. Generic ownership. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2006).

[8] Wrigstad, T., and Clarke, D. Existential owners for ownership types. *Journal of Object Technology* (May 2007). Accepted for publication.

[9] Zhao, T., Palsberg, J., and Vitek, J. Type-Based Confinement. *Journal of Functional Programming 16*, 1 (2006), 83–128.