Alias Control with Ownership Types

Aivar Annamaa
aivar.annamaa@gmail.com

December 21, 2010

Abstract

Mainstream object-oriented programming languages allow creating networks of objects, where possible references and interaction patterns between objects are not statically restricted. While this provides great flexibility, it also makes difficult to predict the behaviour of large programs, because state updates in one part of the program can affect behaviour in any other part of the program.

Parametric ownership types and *Universe types* are families of type systems which use annotations to restrict references between objects in order to increase static predictability about program behaviour. In this paper we describe two variations of these type systems and compare their applicability in different situations.

1 Introduction

In most object-oriented languages, objects are passed around by reference. Upon "meeting" another object (eg. passed as method argument), its reference can be saved and used later to read or modify referenced object's state. This kind of flexibility allows many powerful programming patterns, but as a downside it brings along very unpredictable heap structure, where any object can potentially reference and modify any other object. This situation complicates program understanding and makes many static analyses difficult or impossible.

Creating different (and possibly distant) references to same object is called *aliasing*. Difficulties with aliasing arise because when looking at (or analyzing) the source code of a class we cannot be sure that we see all the operations that can possibly affect the objects of this class at runtime. During program runtime, a reference to a part of the object may reach to a different part of the program and can be used from there in unexpected ways. Because of such coupling, components of OO programs can't be analyzed in modular way. Besides just complicating program dynamics, aliasing is often source for security related bugs[1].

Programming languages support aliasing, because it offers several benefits. Besides allowing more efficient implementations (passing objects by reference and using destructive updates instead of copying), flexibility of aliasing can be used in complex algorithms requiring object sharing or dynamically changing relationships between objects, as "[...] aliasing creates *implicit communication channels*" [14].

Although useful, the full power of aliasing is very often not needed and the possibility of it just creates extra burden for understanding the program. Even for complex problems, some restricted form of aliasing would be usually sufficient. To keep the benefits of aliasing without complicating simple problems, several schemes of *alias control* have been proposed.

At first sight, topic of alias control seems to be related to visibility annotations (like `private`), which also restrict arbitrary access to interior of objects. The difference is that `private` can only forbid accessing this particular *field* (*name encapsulation*), but ownership types can restrict passing around object *references* (*object encapsulation*).

In this review we look into two type systems which restrict aliasing by differentiating between references pointing to objects *inside* the current object vs. references pointing *outside* the current object. In other words, the distinction is made between objects *owned* by the current object vs. rest of the objects.

This paper is organized as follows. In section 2 we introduce some general terms and give an overview how *parametric ownership types* are used to control aliasing. Section 3 describes *universe types*, which is a different take on the same problem with different strengths and weaknesses. Section 5 concludes by comparing merits of and problems of either approach and listing some applications enabled by alias control.

The review is mostly based on [9, 4, 11], which have been the most influential of early papers describing parametric ownership types and universe types.

2 Parametric ownership types

Parametric ownership types (often called simply *ownership types*) annotate all variables (including fields, method parameters) in the program with ownership information. They enforce the *owner-as-dominator* encapsulation discipline: objects can reference only objects *owned* by them or other objects in the same

[1]A standard example of this is a bug in version 1.1 of Sun's JVM, which allowed modification of a private array (containing information about applet's signers) using a reference given out for reading the information.

context, no incoming references from outer contexts are allowed.

Simplest usage of ownership types would be dividing fields of an object into two distinct groups: its *representation* and *nobody's representation.* Objects referenced by representation fields are integral part of the parent object and their lifetime never exceeds that of the parent. We say that parent object *owns* or *contains* the objects of it's representation.

Second group of fields consists of references to objects which can be thought of living outside the current `this` object and may be referenced by several different objects. Objects in this group don't belong to any particular parent object – they are *nobody's representation.*

We use term *object context* to denote a group of related objects. All objects owned by an object form a separate context and all objects without an owner form together a global *root* context. Note that object context is (usually) only a compile-time concept without any representation in runtime. In some papers, term *region* can be used instead of *context.*

2.1 The object contexts `rep` and `norep`

Parametric ownership type system provides two special annotations – `rep` and `norep` – to denote representation of `this` object (it's owned context) and nobody's representation (root context), respectively. Figure 2.1 demonstrates usage of these two annotations using a simple example in a Java-like language.

Most important idea in this example is that a Car contains different kind of references. Reference to an Engine is marked as representation, meaning that the object this reference points to is owned by Car, it lives in the context of it's parent Car and is completely under it's control. Note that eg. Java offers *name protection* by forbidding outside access to a *field* declared `private`, but contents of a private field can still be returned from a method. Ownership types provide *object protection* – it is guaranteed that references to `rep` objects can't be accessed outside the parent object.

In this example, encapsulation of `engine` object forbids accidental or deliberate violation of a precondition mentioned in method `Car.go()` – the engine should be started only when the driver is present. We can be sure of it, because according to *rep* annotation, all accesses to a Car's engine are described in the source-code of Car class [2].

Driver of a Car, on the other hand, is intended to be sharable by different objects in the system, therefore it is marked as not belonging to any object and is created in a special context, called *root* (or sometimes *world*). This context is global and accessible to all objects. Therefore it can be used for eg. sharing

[2]Here we are ignoring subclassing, but usually it's possible to additionally analyze all subclasses, to check that required conditions and invariants still hold

```
class Engine {
  void start() { ... }
  void stop()  { ... }
}

class Driver { ... }

class Car {
  // an Engine belongs to Car
  rep Engine engine;   // so it’s part of representation of this

  // a Driver doesn’t belong to Car
  norep Driver driver; // it’s nobody’s representation

  Car() {
    engine = new rep Engine(); // create new Engine as representation of this
    driver = null;
  }

  rep Engine getEngine() { return engine; }

  void setEngine(rep Engine e) { engine = e; }

  void go() {
    if (driver != null) engine.start();
  }
}

class Main {
  void main() {
    norep Driver bob = new norep Driver();
    norep Car car = new norep Car();

    car.driver = bob;
    car.go();

    // car.engine.stop(); // type error, can’t reference engine from outside Car
    // car.getEngine().stop(); // type error, same reason

    rep Engine e = new rep Engine();
    // car.setEngine(e); type error, wrong rep
  }
}
```

Figure 1: Car Example

immutable *value objects* and objects which provide system-wide services. In terms of ownership contexts, standard OOP languages keep all objects in root context.

Note that neither `Engine` nor `Driver` class definition fixes the context where objects of that class live – context is specified in usage places as a (prefix) argument to constructor. In the same program, another class may include a Driver as part of it's representation and type annotations in respective variable/field declarations are used to guarantee that a `rep Driver` can't be mixed up with a `norep Driver` or with a `rep Driver` from another context.

Last line of the `main` method shows, that context `rep` is always relative to `this` object[3] and objects from different `rep` contexts (eg. engines from different instances of Car class) are incompatible.

2.2 Context parameters

Division of references into root context (`norep`) and individual object contexts (`rep`) offers quite limited form of alias control – it's not possible to express more complex ownership relationships between objects. To provide more flexibility, parametric ownership type system supports *context parameters* which are used similarly to type parameters introduced with Java 1.5 generics[4].

Example program in figure 2.2 defines and uses a stack data-structure, implemented using linked list, where the concrete context of data items stored in the stack is not prescribed. Instead, when creating a Stack instance, required context can be specified in angle brackets as context argument. Corresponding parameter (`o`) is used throughout the Stack class, just like special context annotations `rep` or `norep` in previous example. Note how context specified in Main class (`norep`) is passed along from Stack to Link. Only one context parameter is required in the classes of this example, but in general a class can have any number of these.

In addition to context parameters, Link class also demonstrates another special ownership annotation – `owner`[5]. This annotation, just like `rep`, is relative to `this` and it denotes the same context where `this` lives. In this example, the usage of `owner` annotation means that all linked Link-s need to live in the same context, and from the declaration of top Link in the definition of Stack, we see that (in this example) this happens to be the `rep` context of the Stack – it follows that all Links used by a Stack are also owned by it.

The context `owner` can be thought of as an implicitly defined context parameter which is given value at construction place of an object (in form of a prefix

[3]To emphasize this, in some papers keyword `self` is used instead of `rep`

[4]In this paper, parameters given in angle brackets are context parameters – type parameters, as in Java, are not used

[5]Called `peer` in some papers

```
class Link<o> {
  o Object data;

  // next Link lives in same context as this
  // its data is owned by o
  owner Link<o> next;

  Link (o Object data) {
    this.next = null;
    this.data = data;
  }
}

class Stack<o> {
  // initial link is owned by stack
  rep Link<o> top;

  Stack() { top = null; }

  void push(o Object data) {
    // each new link is also owned stack
    rep Link<o> newTop = new rep Link<o>(data);

    newTop.next = top;
    this.top = newTop;
  }

  o Object pop() {
    rep Link<o> oldTop = top;
    this.top = oldTop.next;
    return oldTop.data;
  }
}

class Main {
  // create an owned stack containing globally shared Objects
  rep Stack<norep> stack = new rep Stack<norep>();

  void storeSharedObject(norep Object obj) {
    stack.push(obj);
  }
}
```

Figure 2: Context Parameters Example

argument) – it's just a syntactical convenience and doesn't have any special semantics compared to "regular" ownership parameters.

3 Universes

Like parametric ownership type system, universe type system conceptually divides object store into separate contexts, called *universes*. Most important difference between these type systems is that parametric ownership restricts what can be aliased, whereas universe types restrict how aliases can be used. Universe type system is designed to incur less annotation overhead, and it uses a different approach for obtaining flexibility.

3.1 Basics of Universes

Universe type system uses two ownership annotations (`rep` and `owner`[6]), whose meaning is always relative to `this` object and a third annotation (`any`) with "absolute" meaning. Passing ownership information around via context parameters is not supported.

Like in parametric ownership types, `rep` references point to objects directly owned by `this` and `owner` references point to objects in the same contexts as `this`.

Variables or fields marked as `any`[7] can point to objects in any context but these references can not be used for modifying referenced object[8]. This approach is based on observation that object sharing becomes problematic only when the object is modified using different references. Therefore in universe type system distinction is made between read-write references and read-only references. Read-only `any` references can be passed around without restrictions and every other reference can be implicitly converted to an `any` reference.

Type system still guarantees, that references marked with `rep` and `owner` can be modified only by the owner of respective contexts. This property is called *owner-as-modifier*.

Note that `any` only marks references as being read-only – usually there exists at least one read-write reference to same object. For this reason this annotation is semantically different from `immutable` annotation proposed by different authors for different OO languages[9].

[6]actually most Universe papers call this annotation `peer`, but for regularity we will call it `owner`

[7]sometimes called `readonly`

[8]In order to distinguish between pure functions and methods with side-effects, effect annotations are used on methods

[9]In depth treatment of this issue is given by Boyland [5]. He also explains why it's good

3.2 `any` instead of context parameters: trade-offs in flexibility

Universe type system doesn't use context parameters in order to reduce syntactical complexity of required annotations. It tries to gain back flexibility by allowing unrestricted creating and passing of read-only aliases. This gives clear advantages when it's not necessary to modify objects shared by `any` references. If multiple read-write references to same object are required, then static information provided by the type system is insufficient. In order to solve this issue, universe types allow type casts which are checked at runtime. For supporting this, actual context information is transparently stored with the objects[10].

Figure 3.2 demonstrates definition and usage of a simple array based list where it is possible to store read-only references to any objects.

In order to be as general as possible, ArrayList uses `any` annotations for the data items stored in it. This allows user to store also `rep` and `owner` objects in it. If user later wants to modify those objects, it needs to cast read-only references returned from ArrayList to proper types. Runtime system will check that given object actually belongs to specified context.

Note that array variable uses two ownership annotations – one for array itself and second for contents of the array.

Annotation `any` is similar to `norep` in that it denotes globally shared objects, but because its restriction on modification it's not possible to express globally accessible mutable services.

4 Issues with Parametric Ownership types and Universes

Although there are several clear benefits in using either of the proposed type systems, there are also several problems to tackle before they can become widely used in commercial development.

Extra annotations can create two possible barriers: industry is not keen to switch over to new syntax and entering annotations can be seen as unnecessary extra work. Fortunately there are several solutions [1, 16] that show that ownership type systems can be specified using available syntactical features already present in Java (generic type parameters and Java annotations). Unfortunately these constructions are not syntactically optimal and therefore increase visual noise in source-code considerably.

to base immutability on ownership types

[10]for some objects this overhead can be avoided if it's possible to statically verify that casts are always valid

```
class ArrayList {
    // rep applies to array, any applies to array content
    rep any Object[] arr;

    ArrayList(int n) {
        arr = new rep any Object[n];
    }

    void set(int i, any Object value) {
        arr[i] = value;
    }

    any Object get(int i) {
        return arr[i];
    }
}

class User {
    void use() {

        rep ArrayList list = new rep ArrayList(2);
        list.set(0, new rep Object());
        list.set(1, new owner Object());

        rep Object obj; // prepare a read-write variable

        obj = (rep Object)list.get(0);
        // obj = (rep Object)list.get(1); // runtime cast error
    }
}
```

Figure 3: Universal List Example

Another important issue to take into account is restrictions that ownership types place on program design. As an example we consider specific variant of "iterator" design pattern.

External iterators are easy to write using both parametric ownership types and universe types, because they rely only on collections' public interface (and interface design principles usually match with ownership types requirements). But in order to get maximum efficiency, *structure sharing iterators* are needed, that link into internal representation of the collection. By the definition, structure sharing iterators can't be written using parametric ownership types, because linking from outside the owner into the representation is forbidden.

Structure sharing iterators can be written using universe types (because of unrestricted read access and possibility of downcasting). Figure 4 presents one possible implementation.

Note that universe types require slight change in the usual design of such iterator. `Iterator` must use services from `LinkedList` to modify `Node`-s, as latter is the only object with read-write access to `Node`-s.

Several other well-known design patterns cause problems for either or both parametric ownership types and universe types. Good overview is given in [13].

5 Conclusion and Applications

The two type systems described in this review can be regarded as two of the most popular branches of research regarding alias control in OO programs. In many cases both systems provide useful information for program analysis and their usage is quite intuitive and doesn't require too much extra annotations. The problems arise with more complex relationships between objects. Here each of the type systems has its own strengths and weaknesses.

Parametric ownership allows to pass context information along the structure of object relationships, therefore precise context information is always available. Its *owner-as-dominator* access policy ensures total control over owned objects, therefore analyses related to garbage collection and representation independence work better with parametric ownership types. On the other hand, this approach tends to be too inflexible for implementing some common patters (eg. collection iterators). Also, specifying context parameters can be too unwieldy, especially in combination with generic type parameters.

Universe types try to offer comparable utility without context parameters, by allowing unrestricted sharing of read-only references and enforcing *owner-as-modifier* access policy. This flexibility comes at the cost of reduced precision at compile time and therefore it needs extra time and memory resources at runtime. Yet, universe type system provides sufficient precision to enable *mod-*

```
// wrapper for links of the list
class LinkedList {
    rep Node first;

    // LinkedList owns Nodes and can modify them
    void set(any Node np, any Object e) {
        // assuming np is one of this list nodes
        rep Node n = (rep Node) np; // downcast is required
        n.data = e;
    }

    any Node get(int i) {
        ...
    }
    ...
}

class Iterator {
    owner LinkedList list; // owns nodes
    any Node pos;    // current position


    Iterator (owner LinkedList list) {
        this.list = list;
        this.pos = list.getFirst();
    }


    void setValue(any Object o) {
        // Iterator has direct (readonly) access to Nodes
        // but need to delegate modification to LinkedList
        list.set(pos, o);
    }

    ...
}
```

Figure 4: Iterator example

ular analysis, because it can prevent non-local state updates through arbitrary references.

During last few years many variations and combinations of these ideas have appeared in literature. Ownership types have been combined with uniqueness annotations[2, 8] – this makes it easier, for example, to track objects' movement between different ownership contexts.

In the type systems described above, whenever we needed to share something between two unrelated objects, we needed to share it with everybody. More precision is gained by allowing multiple owners for one object[6]. Ownership information has been also used together with complex effect systems allowing more flexibility in usage of `any` references[7].

Most popular area for applications of ownership information appears to be concurrent programming. Both parametric ownership types and universe types have been used as basis for type-systems avoiding race conditions and deadlocks[12, 10, 3] and for automatic parallelization[15].

Unfortunately ownership types haven't yet seen wide use in commercial setting. Seems that researchers still need to look for best balance between precision of the type system, flexibility of its application and amount of annotations required.

References

[1] Jonathan Aldrich. Ownership domainsin real world. *IWACO 2007*, 2007.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330, New York, NY, USA, 2002. ACM.

[3] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM.

[4] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223, New York, NY, USA, 2003. ACM.

[5] John Boyland. Why we should not add readonly to java (yet). *Journal of Object Technology*, 5(5):5–29, June 2006. Workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2005.

[6] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. *SIGPLAN Not.*, 42:441–460, October 2007.

[7] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37:292–310, November 2002.

[8] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 59–67. Springer Berlin / Heidelberg, 2003.

[9] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, 1998.

[10] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, September 2007.

[11] Werner Dietl and Peter Müller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 4(8):5–32, 2005.

[12] Eric Kerfoot, Steve McKeever, and Faraz Torshizi. Deadlock freedom through object ownership. In *IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages 1–8, New York, NY, USA, 2009. ACM.

[13] Stefan Nägeli. Ownership in design patterns. Master's Thesis, 09/2005 – 03/2006, March 2006.

[14] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[15] Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 933–940, New York, NY, USA, 2009. ACM.

[16] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2010)*, Revo, NV, USA, October 19–21, 2010.