

# The Interdependence of Effects and Uniqueness<sup>\*</sup>

John Boyland

Department of EE & CS, University of Wisconsin-Milwaukee  
boyland@cs.uwm.edu

**Abstract.** A good object-oriented effects system gives the ability to define abstract regions (or “data groups”) of state within objects that can be extended in subclasses. Then one can specify (for instance) read and write effects on these abstract regions. Additionally, effects on “wholly owned subsidiary” objects should be seen as effects on regions of the owning object. For instance, an assignment within a bucket of a hash table should be seen as an effect on the hash table alone. Correctness of this transfer of effects depends on the bucket being accessible only through the hash table; it must be *unique*.

Uniqueness can be guaranteed using *destructive reads* (in which a unique variable can be used at most once). Destructive reads are inconvenient, so most uniqueness systems permit *borrowing reads* as well, in which a temporary alias of a unique variable is permitted. But if the unique variable is read during the lifetime of this alias, the uniqueness invariant fails. So we wish to ensure that this read effect does not happen. For modularity reasons, we use effects annotations on methods to check for such read effects.

Thus we see that effects and uniqueness depend on each other. Our position is that the use of annotations breaks the cyclic dependence as long as the annotations are given semantics independent of the analyses. As a semantics of uniqueness annotations is already available, we then sketch a semantics of effects annotations independent of uniqueness. Thus decoupled, one can prove the correctness of a uniqueness analysis and an effects analysis without regard for the other.

## 1 Interdependence

Properties of code effects and alias confinement are important when analyzing the meaning of complex programs. Putting checked annotations on methods aids sound modular reasoning, because a component of a system can then be independently verified.

---

<sup>\*</sup> Work supported in part by the National Science Foundation (CCR-9984681) and the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF under contract F30602-99-2-0522. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

```

class Point {
  public region Position;
  private int x in Position;
  private int y in Position;
  public scale(int sc)
    writes Position
  {
    x *= sc;
    y *= sc;
  }
}

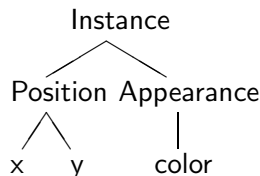
class Point3D extends Point {
  private int z in Position;
  public void scale(int sc)
    writes Position
  {
    super.scale(sc);
    z *= sc;
  }
}

class ColorPoint extends Point {
  public region Appearance;
  private int color in Appearance;
}

```

(a) (b)

**Fig. 1.** The definitions of (a) classes `Point` and `ColorPoint`, and (b) class `Point3D`



**Fig. 2.** Hierarchy of regions for class `ColorPoint`

In this section we motivate effects and uniqueness systems and show how each may depend on the other in the context of a hash table example. In Section 2, we discuss how the interdependence can be resolved by giving independent semantics to uniqueness and effects annotations. Since a semantics of uniqueness already exists, we propose the rough outline of a semantics of effects annotations.

### 1.1 Effects

In an object-oriented effects system (such as of Greenhouse and Boyland [8] or Leino [11]) the mutable fields of the object are abstracted as “regions” of state (the term used in this paper) or “data groups” [11]. This abstraction permits the effects of public methods to be declared using public abstractions that hide the names of actual fields. Regions are inherited along with fields and can be extended to include new regions and fields. Thus a subclass can extend the behavior of the superclass’s methods to read or write additional fields. Figure 1 gives an example of a `Point` class with two subclasses illustrating extension

possibilities. The non-executable annotations are in *slanted* style. Figure 2 shows the hierarchy of regions within the `ColorPoint` class.

Often, not all the notional state of an object is actually contained directly within it. Part of the state of the object is contained in “wholly owned subsidiary” objects, private objects known only to the implementation of the class. Consider for example, Java’s `Vector` class; the contents of the vector are stored in an array, which may need to be changed if the vector outgrows the array. The code for `addElement` is basically as follows:

```
public void addElement(Object elem) {
    ensureCapacity(size+1);
    contents[size] = elem;
    ++size;
}
```

where `size` and `contents` are private fields. Ignoring the effects of the call to `ensureCapacity`, this code has two side-effects: the array in `contents` is updated, and the private field `size` is changed. In this case, the `size` field is in a region `Size`. Thus the second change can be given as a (read and a) write to the `Size` region of the receiver (`this`) of the method call. The class has a second region `Elements` which contains the array field, but the array assignment does not actually change the array field itself, it changes the elements of the array that is stored in the field. The indirection causes a problem; it is not useful to the caller to say that “some array somewhere is changed” or even “some array which is referred to by a field in the region `this.Elements` is changed.”

Instead of treating the array as a separate object, it is preferable to consider the array to be part of the vector, which is reasonable since the implementation arranges that every vector has its own array. So we use a *transfer of effect* from the array to the vector and declare the effects of the method as “state in this vector is changed.” The transfer is declared in the `Vector` class using some syntax such as the following:

```
private Object[] list in Elements with [] in Elements;
```

Here the field is placed in the `Elements` region and the individual array elements (in a region `[]` of the array) are placed in the `Elements` region of the vector. Figure 3 gives a fragment of a hash table class using effects transfer. The effects on fields of the buckets are transitively transferred to the hash table object itself.

The soundness of this *transfer of effect* depends on several conditions. For our vector example, it is not enough that the array field be private; rather, the client must not have access to the array object through some other means. Otherwise, it could observe the effect on the array. For instance, two vectors must not use the same array, otherwise a change in one vector could be observed through the other vector. The invariant that we need is called a “uniqueness invariant.” If an object is unique, effects on its regions can be safely mapped to effects on the object which refers to it. The effects analysis checks an effects transfer using uniqueness annotations.

```

class Bucket {
  public region Key, Value, Structure;
  Object key in Key, value in Value;
  Bucket next in Structure with Key in Key,
                                   Value in Value,
                                   Structure in Structure;
  Bucket(Object k, Object v, Bucket n) { ... }

  Object get(Object k)
    reads this.Key, this.Value, this.Structure, any(Object).Equal
  {
    if (key.equals(k)) return value;
    else if (next == null) return null;
    else return next.get(k);
  }
  :
}

public class Hashtable {
  region Key; region Value; region Structure;
  private Bucket[] buckets in Structure with [].Key in Key,
                                             [].Value in Value,
                                             [].Structure in Structure;

  private int size = 0 in Structure;

  public synchronized Object get(Object k)
    reads this.Key, this.Value, this.Structure, any(Object).Equal
  {
    int h = k.hashCode() % buckets.length;
    Bucket b=buckets[h];
    if (b == null) return null;
    else return b.get(k);
  }
  :
}

```

**Fig. 3.** Hash tables with effects transfer

## 1.2 Uniqueness

There have been a number of uniqueness proposals: Islands [9], Linearity [2], Eiffel\* [16], Balloons [1], Virginity [15], and Alias Burying [3]. Islands and Balloons are not useful for modeling a vector class because they would not permit a vector to hold any shared references. Essentially the contents of the vector would have to be unique or immutable references. Flexible Alias Protection [18] describes how to avoid these problems but uses ownership [6] rather than uniqueness.

Baker’s Linearity and Minsky’s Eiffel\* (as well as Hogg’s Islands) use *destructive reads* in which a read of a unique variable also implicitly stores null in it. Destructive reads ensure uniqueness or null; instead of having multiple aliases to an object, all but one will hold null. Destructive reads, however, are not a satisfactory solution. First of all, it is difficult to program using such “slippery” variables: many methods that take unique variables must also return them as well as their normal result. Figure 4 shows how to code the hash table fragment from Fig. 3 in a system with strict destructive reads (which are underlined). We assume that operations such as `==` and `[.]` are overloaded to work for unique objects, so that comparisons and array accesses can be made without destroying unique objects. The second problem is that the code now has many more side-effects than previously, which will be difficult to ignore in an effects analysis.

More importantly, the code must now be prepared for the fact that the unique variable may actually be null. For example, the hash table `get` method must be prepared for the possibility that the field holding the array, or the array element holding the head of the bucket chain might currently be null due to an ongoing `get` call. Such a situation may take place, for instance, if the `equals` method for an object requires looking something up in a hash table. Aliasing errors may thus be transmuted into null pointer errors. While possessing the virtue of immediate detection, null pointer errors can still have devastating run-time consequences for the program. More desirable is a static checking system that can flag potential alias errors at compilation time. For instance, the hash table class does not put any fields into the `Equals` region (that may be read by the `equals` method). If an `equals` method tries to access a hash table, effects analysis would flag the access as an error.<sup>1</sup>

For these reasons, some systems with destructive reads (such as Eiffel\* and Islands) provide “non-consuming” or “borrowing” reads in which certain kinds of so-called “dynamic” aliases of unique variables are permitted. Methods can be written to take borrowed receivers or parameters and promise not to store them anywhere. At the conclusion of the call, then, uniqueness is restored automatically. Alias Burying similarly supports controlled aliasing, but as soon as the unique variable is read, the aliases must be dead (“buried”). In particular, if a method call may read the unique field, even indirectly, it may not be passed

---

<sup>1</sup> Unfortunately the new collections framework requires `equals` on maps to compare contents. Personally, I believe it was a mistake to require mutable containers to override `equals` and `hashCode`.

```

class Bucket {
    Object key, value;
    unique Bucket next;

    Bucket(Object k, Object v, unique Bucket n) { ... }

    Pair<Object,unique Bucket> get(Object k) unique
    {
        if (key.equals(k)) return value;
        else if (next == null) return null; // we assume == doesn't destruct
        else {
            Pair<Object,unique Bucket> p = next.get(k);
            next = p.second;
            return new Pair<p.first,this>;
        }
    }
    :
}

public class Hashtable {
    private unique Bucket unique [] buckets;
    private int size = 0;

    public synchronized Object get(Object k)
    {
        // NB: if buckets is null, we die with a NullPointerException
        Pair<int,unique Bucket unique []> p = buckets.unique_length();
        buckets = p.second;
        int h = k.hashCode() % p.first;
        unique Bucket b = buckets[h];
        Object result;
        if (b == null) {
            // NB: b might be null due to an ongoing 'get' call
            result = null;
        } else {
            Pair<Object,unique Bucket> p2 = b.get(k);
            result = p2.first;
            b = p2.second;
        }
        // need to restore array element
        buckets[h] = b;
        return result;
    }
    :
}

```

Fig. 4. Hash tables with destructive reads (underlined)

an alias of that same field. With Alias Burying the uniqueness annotations given in Fig. 4 can be used with the code of Fig. 3; we do not need destructive reads.

Virginity and Alias Burying both provide ways to check uniqueness statically without changing the underlying language semantics. In an interesting convergence, both systems require “reads” clauses to ensure that uniqueness is not compromised [3, 14].

When a unique pointer is passed out of the scope of the owning object in a call (for example by calling a method), there must not be another call on the owning object that uses the unique field until the first call is complete. Otherwise, the uniqueness invariant would be violated; in the case of destructive reads, a null pointer would be encountered unexpectedly. In the case of alias burying, one of the called method’s parameters is suddenly no longer valid. Thus the static analysis needs to ensure that the unique field is not read during the dynamic lifetime of the call. The Alias Burying paper [3] suggests listing the complete set of fields read during every procedure but concedes that this requirement breaks information hiding. A much better solution is to use an object-oriented effects system, but that brings us full circle.

Thus we see effects analysis depends on uniqueness analysis which depends on effects analysis. Leino has also noticed this interdependence [12].

## 2 Resolving the Interdependence

In this section, we show how the interdependence can be resolved soundly by giving a semantics to the annotations. Then we show how one can give semantics to uniqueness annotations and finally we sketch an idea for giving meaning to effects annotations.

### 2.1 Separating the Analyses

The interdependence is a concern to us when we are trying to determine whether the effects analysis and uniqueness analysis are both sound, and if so, writing a proof. If both analyses depend on each other, we need to prove the correctness of both together. In our situation, however, we are working with annotations on methods and classes that summarize their behavior in particular ways. The annotations provide some indirection: in the same manner as type checking, analysis checks the annotations on an entity while using the annotations on other entities, or (in the case of self-reference) itself.

Proving the soundness of the analysis, then, naturally requires that the annotations be assigned some meaning against which the analysis is checked. Furthermore, to avoid tautologies, and to permit the substitution of more accurate analyses, the semantics should be defined at a lower level than the analyses. In particular, we wish to avoid giving an annotation a meaning such as “Analysis A gives result B when applied to this method.” Rather we are interested in meanings such as safety properties: “Event E will never happen while executing this method as long as the program state at the point of entry satisfies predicate I.”

Assuming then that we can place the semantics on a complete (semi-)lattice and define our analyses monotonically on this lattice, proving the correctness of the analysis becomes an application of the theory of abstract interpretation [7]. In particular, we can prove the soundness of a uniqueness analysis separate from any effect analysis, and vice versa.

## 2.2 Semantics for Uniqueness

In a current paper [4], we give a capability-based low-level language that can be used to give a meaning to uniqueness annotations. Uniqueness is expressed through the exclusive holding of read and write access rights. Uniqueness invariant failures are converted into capability failures, so that any analysis that ensures the absence of capability failures ensures the correctness of the uniqueness annotations. Furthermore, the only way lack of an access right can be observed is through a capability failure, and thus if a program is guaranteed to never have failures, it can be executed in an environment that ignores access rights completely. In particular, no space is needed to store the access rights.

## 2.3 Semantics for Effects

A useful semantics of effects is not yet available. Our earlier paper [8] gives an indirect basis for defining the soundness of effects annotations: whether a conflict detection analysis using the effects annotations always catches data dependencies between adjacent program portions. But it does not give a semantics to the annotations directly, and the conflict detection analysis is overly conservative in several situations.

Leino’s abstract variables [10] (from which data groups were derived) have a clear meaning in the context of a program specification. However, because of effects transfer, when a module specification uses `modifies` clauses<sup>2</sup> the client of a module cannot make interesting use of the information in a sound manner [13]. Any problem in effects transfer shows up as unsoundness in the client, which is “unfair” because the client has no control or even awareness of the transfer.

The semantics of effects could be specified using uniqueness or ownership [17, 5] but that would tie us to a particular system, not directly related to effects. Uniqueness systems find it difficult to model doubly linked lists and ownership systems find it difficult to model transfer. Thus we prefer an effects semantics not depending on the particular method for alias containment.

We suggest instead that the implementation of a method be charged with ensuring the soundness of effects transfer. In particular, the state thus mapped into the state of another object must not be available to any caller that does not have access to the object through which the transfer is effected. Our position is that this intuition can be formalized through a run-time hierarchy of actual regions. Effects transfer is implemented by mutations on this tree.

---

<sup>2</sup> Leino does not currently use `reads` clauses.

At the start of a program, the entire state (the set of all fields) is available for reads and writes. Whenever a method with an effects annotation is called, the available state is pared down to the intersection of the currently available state and the state implied by the annotations. The available state is then restored after the call. When a new object is created, all of its fields are made available for both reads and writes. When a field with an effects transfer is assigned to, we check that the object's state is fully available for reads and writes (except perhaps for immutability), and then transform the region hierarchy. If a read of a field outside the permitted area occurs, the state is presumed immutable, because reads of immutable state need not be declared. It is marked as such for the duration of the program. When a write occurs, the system checks that the field is in the area currently permitted for writes and that the field is not marked immutable. An error causes evaluation to get stuck.

In this way, we achieve a semantics of effect annotations and transfer that does not depend on a particular definition of uniqueness or ownership. The details of this suggested semantics remain to be worked out.

### 3 Summary

Two apparently different problems, (1) describing the reads and writes of methods and (2) upholding uniqueness invariants, are nonetheless interdependent. The connectivity presses us to define the semantics of effects separately from the semantics of uniqueness. We currently have a promising semantics for uniqueness, but an effects semantics which has the desired properties remains to be fully fleshed out.

### Acknowledgments

I thank Aaron Greenhouse, Bill Retert and the FTJP 2001 reviewers for their many useful comments. I also thank Rustan Leino for a good technical conversation which encouraged me to write up this interesting interdependence. All omissions and errors are strictly my own.

### References

1. Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, Jyväskylä, Finland, June 9–13, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, Berlin, Heidelberg, New York, 1997.
2. Henry G. Baker. ‘Use-once’ variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.
3. John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

4. John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. To appear in ECOOP 2001, 2001.
5. David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. To appear in ECOOP 2001, 2001.
6. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, pages 238–252. ACM Press, New York, January 1977.
8. Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, Lisbon, Portugal, June 14–18, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer, Berlin, Heidelberg, New York, 1999.
9. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Phoenix, Arizona, USA, October 6–11, *ACM SIGPLAN Notices*, 26(11):271–285, November 1991.
10. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, Pasadena, California, USA, 1995. Available as Technical Report Caltech-CS-TR-95-03.
11. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):144–153, October 1998.
12. K. Rustan M. Leino. Some thoughts about rep exposure and alias confinement. December 2000.
13. K. Rustan M. Leino and Gren Nelson. Data abstraction and information hiding. SRC Research Report 160, Compaq Systems Research Center, Palo Alto, CA, November 2000.
14. K. Rustan M. Leino and Raymie Stata. Smothering rep exposure with reads clauses. November 1999.
15. K. Rustan M. Leino and Raymie Stata. Virginitly: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, April 1999.
16. Naftaly Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, Linz, Austria, July 8–12, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer, Berlin, Heidelberg, New York, July 1996.
17. Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In Sophia Drossopolou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *2nd ECOOP Workshop on Formal Techniques for Java Programs*, Nice, France, June 12. 2000.
18. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming, 12th European Conference*, Brussels, Belgium, July 20–24, volume 1445 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York, 1998.