Why we should not add **`readonly`** to Java (yet)[*]

John Boyland[†]

June 30, 2005

Abstract

In this paper, I examine some of reasons that "readonly" style qualifiers have been proposed for Java, and also the principles behind the rules for these new qualifiers. I find that there is a mismatch between some of the motivating problems and the proposed solutions. Thus I urge Java designers to proceed with caution when adopting a solution to these sets of problems.

1 Proposals for "readonly" in Java

The purpose in having qualifiers such as "readonly" on types in a programming language is so that programmers can enlist the compiler (and loader) in enforcing rules about the proper use of data. One part of the program may be willing to grant access to data to another part of the program only if it can be guaranteed that the other part does not mutate the data. Several proposals have been made to add enforceable "readonly" qualifiers in Java programs:

JAC [20] Kniesel and Theisen's system "Java with Access Control";

Universes [23] Müller and Poetzsch-Heffter's system for alias and dependency control.

ModeJava [31] Skoglund and Wrigstad's mode system for read-only references in Java.

Javari [3] Birka and Ernst's system for Java with "Reference Immutability." (This work has been updated and will be presented at OOPSLA 2005, but no preprint is publicly available yet.)

In this section, we compare these proposals with emphasis on their broad similarities. We also compare with "const" in C++ [32] and with our earlier paper on capabilities [8].

A motivating example is used in Section 2 which demonstrates short-comings of the "readonly" concept, in particular "observational exposure," which is further explained and criticized in Section 3.

1.1 Basic Rules

The basic idea in these systems is that a type may be given the "readonly" annotation. Formally, if T is some class, then `readonly` T represents a super-type of T. In other words, it is permitted to implicit coerce a reference of type T to `readonly` T (widening conversion), but the reverse coercion requires an explicit cast. This rule carries over in the obvious way to parameters, return values, local variables and fields. The receiver of a method may be typed as `readonly`, in which we call the method a *read-only* method.

The main restriction on a reference value of a "read-only" type is that the reference cannot be used to change a field. For a quick example, consider the artificial class in Fig. 1. Ignoring the

[*] Work supported in part by the NASA/Ames High Dependability Computing Program (NCC-2-1298)

[†] University of Wisconsin–Milwaukee, USA, `boyland@cs.uwm.edu`

```
class B {
  A f1;
  readonly A f2;
  mutable A f3;
  mutable readonly A f4;
  readonly A method1() readonly {
    this.f1 = new A(); //! Writes field of readonly this
    this.f3 = null;
    if (...) return this.f1;
    else return this.f2;
  }
  A method2(A p1, readonly A p2) {
    this.f1 = p2; //! Field needs a read-write ref.
    this.f2 = p1;
    this.f3 = p1;
    return this.f4; //! Result needs a read-write ref.
  }
}
```

Figure 1: An artificial class illustrating read-only rules.

`mutable` annotation for now, this example illustrates the rule. In `method1`, the receiver is `readonly` (indicated directly before the body) and thus `this` has type `readonly B`. Thus it is not legal to write field `f1` here. However, we are free to *read* these fields and since the return type is a "read-only" type, we can return the contents of either field `f1` or `f2` without error.

The second method `method2` is declared normally (without `readonly`) and thus doesn't have this restriction, but must still follow the types of the fields, and thus the write to `f1` fails because a "read-only" reference is an inappropriate value to store in the field; an explicit cast would be needed. However, the write to `f2` succeeds because the "read-write" parameter `p1` can be implicitly coerced into a "read-only" reference to be stored in `f2`. The write of `f3` does not require any coercion.

This example shows the difference between `readonly` which applies to the reference stored in a field (protecting the fields of the object referred to), and Java's existing `final` annotation which refers to the field itself, making it non-updatable in any method. This is the distinction that C++ makes between a pointer to a "const" object (whose data members cannot be written) and a const data member with a pointer in it (the pointer can be used for mutation). C++ can make do with one keyword (albeit somewhat confusingly) because pointers are explicitly typed.

Java's `final` annotation protects a field from being updated regardless of whether the method is read-only. JAC and Javari include a `mutable` annotation (borrowed from C++) that has the opposite effect: it makes a field updatable, again regardless of whether the method is read-only. Thus we see that `method1` is permitted to update field `f3` despite being "read-only." Aside from this behavior, `mutable` has no effect, and thus we see that `method2` cannot return the contents of field `f4` since a "read-write" reference is needed. Indeed `mutable` would be the direct opposite of `final`, were it not for transitivity, as explained in the next section.

1.2 Transitivity

In C++, some data members have pointer type, and others have object type. In a const object, an object data member is also const. In other words, if I have a pointer to a "const" object and get a reference to the object stored in this data member, this reference is also "const" (or rather it is also to a "const" object). This transitivity of "constness" is entirely reasonable since the second object is stored within the first. On the other hand, if the first object has a *pointer* to a third object, then if we fetch this pointer, there are no restriction on mutating this object. Again, this *lack* of transitivity is reasonable if we consider the state of an object to consist solely of what is stored inside it.

However, even in C++, some object's state notionally extends to other objects stored on the heap. For instance a "map" object includes pointers to nodes in a red-black tree. Changing any of these nodes conceptually changes the map. And indeed a "map" will protect its nodes and "voluntarily" propagate "constness" to maintain desired invariants.

In Java, there is no possibility of storing one object inside another (at the language level), and thus all subsidiary objects must be referred to through (implicit) pointers. The question is then whether transitivity should apply or not. The *right* answer should depend on whether the referred to object is part of first object or not. If it is conceptually part of that object's representation, then transitivity applies, otherwise it does not. This is the rule used in "flexible alias protection" [24].

Unfortunately, of the four proposals described above, only Universes distinguishes the representation fields from other fields.[1] All four proposals decide that the safe approach is to assume all read-write fields in an object refer to its representation. This heuristic is reasonable for many classes, but notably not for container classes, since the elements in the container are not notionally part of the container. For this reason, in our low-level capability system, "readonly" was specifically not transitive [8].

In the systems with `mutable`, it is transitive. Thus unlike `final`, `mutable` not only applies to the field so annotated, but also to the object whose reference is stored in the field, unless the field is also declared `readonly`.

In any of the four systems, transitivity is illustrated in that `method1` would not be permitted to fetch a read-write reference from field `f1`. The `readonly` annotation on the receiver carries over to the value fetched from a field of `this`. How then can we write a "getter" function for `f1` that returns a read-only reference if and only if the receiver is read-only? In Universes such a method is illegal since it exposes the representation of the object. JAC allows it with the rule that a "non-read-only" return type of a read-only method is understood as being linked with the actual receiver mode. In ModeJava, the annotation `context` on the return type indicates the same situation. In C++, the solution is to overload the method; the programmer writes two methods with identical (and short) bodies but with the two possible signatures. Javari permits this solution, but also supports genericity in which the mode is a parameter to the method as an extension to Java 1.5's generics.

1.3 Dynamic casts

In C++, the programmer can "cast away constness" with impunity, which is in accordance with the general principles of the language favoring flexibility over safety.

In Java, where safety is much more important, a cast on a reference is used to perform a "narrowing conversion." At run-time, the object (if not null) is tested to see if it indeed is of the desired class, and if not an exception is thrown. So-called *dynamic casts* require that run-time type information be saved. In Java and in C++ objects with virtual methods, this comes at little cost. In the case of `readonly` the cost is non-negligible since it would involve keeping an extra bit in each pointer (similar to what is done in capabilities [8]). The potential cost (of a safe solution) is cited as the reason why JAC does not permit "read-only" to be cast away.

ModeJava supports something analogous to a dynamic cast to test whether a (statically) "read-only" pointer "really" is "read-write." Thus a read-only method may test its receiver and perform different actions depending on whether the receiver was actually "read-only" or not. This potentially surprising behavior weakens the meaning of `readonly` and necessitates forced read-only conversions.

In Javari, casts have a peculiar if practical semantics. Dynamically casting a readonly reference as `mutable` away always succeeds, but the pointer is marked as "read-only" so that if it is subsequently used for mutation, an exception is thrown. The analogy is made with the (statically unsafe) type

[1] And Universes severely restricts the non-representation read-write references that an object may possess.

rule that permits arrays to be implicitly coerced into an array of a super-type. If the array is used to store something inappropriate, an exception is raised. In both cases, a dynamic check on uses protects against the dangers of a permitted type-unsafe coercion. This rule is practical since it permits code using `readonly` to safely co-exist with legacy code that lacks proper annotations.

Unfortunately, this rule goes against the spirit of a dynamic cast (which is supposed to check the reference) and also means code may raise unexpected exceptions far from where the erroneous cast occurs. Java's `ArrayStoreException` is seen as a blemish on the language, required because of an over-permissive type rule. It would be unfortunate if adding "readonly" required another such check. Furthermore implementing the cast in this way requires that (some) pointers include an extra bit.

One advantage of Javari's rule is that "readonly" is iron-clad: if a method takes a `readonly` receiver or parameter, then it absolutely cannot modify the state through the reference, even if the actual receiver or parameter is "read-write." This property compares favorably with `readonly` in Universes, as seen next.

Universes have yet another semantics for cast. The system distinguishes representation objects from other objects similar to ownership type systems. Every object has an owner. A read-only reference can be cast to "read-write" by its owner. Thus at run-time, the dynamic cast compares the owner of the object to the current context, and succeeds if they are the same. The cost is thus borne per-object rather than per reference. A surprising implication is that a "read-only" reference is not really read only, it simply cannot be used *externally* for mutation.

1.4 Summary

The four systems reviewed share most of the same rules in which `readonly` adds a layer on the type system. All three assume (or require) that read-write references in a class are part of the representation and thus enforce transitivity of "read-only." Dynamic casts (with varying semantics) can be supported at some additional run-time cost in object or pointer representation.

2 Example

The desire for a "readonly" qualifier is motivated by considerations of software constructions. This section uses illustrative examples in Java where the lack of "readonly" qualifiers exposes a module to misuse unless defensive programming is used. The examples chiefly come from the most recent read-only proposal (Javari). In several of the cases, I argue that the underlying issue does not fit the idea of a "readonly" pointer.

2.1 The case for "readonly"

Figure 2 illustrates a number of situations where the programmer may wish the compiler would enforce good usage, when it does not.

1. The "intersect" method in its comment promises not to change (mutate) its parameter. However this commitment is not expressible in the signature of the method, and thus the compiler cannot be called upon to enforce the signature. In Javari (or using any of the other proposals, with perhaps minor notational changes), the signature could be written

   ```
   void intersect(readonly IntSet set)
   ```

   The compiler would then enforce that the parameter was not in fact mutated.

2. The next declaration of interest is the constructor. It accepts an array of integers and uses this array as its representation. This method exposes the representation to the client, since the

```
/** This class represents a set of integers. **/
public class IntSet {
  /** Integers in the set with no duplications. **/
  private int[] ints;

  /** Removes all elements from this that
   * are not in set, without modifying set. **/
  public void intersect(IntSet set) {
    ...
  }

  /** Makes an IntSet initialized from an int[].
   * Throws BadArgumentException if there are
   * duplicate elements in the argument ints. **/
  public IntSet(int[] ints) {
    if (hasDuplicates(ints))
      throw new BadArgumentException();
    this.ints = ints;
  }

  /** Number of distinct elements of this. **/
  public int size() {
    return ints.length;
  }

  public int[] toArray() {
    return ints;
  }
}
```

Figure 2: A partial implementation of a set of integers.
(Figure 1 from Birka and Ernst's "A Practical Type System and Language for Reference Immutability" [3])

```
public class IntSetView extends JPanel {
  private final IntSet model;
  /** Construct a view of the given integer set.
   * When the set changes, the client should
   * call repaint(). **/
  public IntSetView(IntSet set) {
    ...
  }
  ⋮
}
```

Figure 3: A class to view integer sets.

client can retain the pointer to the array and then, for instance, set all the elements to zero, thus breaking the representation invariant. The client may even be unaware of this problem, assuming that the constructor would make a copy of the array. Birka and Ernst explain that if the array parameter were annotated "readonly" the compiler would notice the (now illegal) initialization of the field with the parameter and flag the error.

3. The caller of the `size` method is not expecting the method to perform a side-effect. In particular it should be possible to call this method even when one has a "readonly" integer set instance. For this reason, Birka and Ernst suggest using a "readonly" qualifier for the (implicit) receiver.

4. Turning to the `toArray` method, we see another case of representation exposure. The signature as given does not prevent the client from breaking the invariant by changing the values in the array.

   As Birka & Ernst observe, if the return value were designated "readonly" (and the compiler enforced this designation), the client would be unable to modify the array elements, and thus could not upset any invariants. The `toArray` method in the Java collection framework is supposed to return a separate (mutable) array. The use of "readonly" here would make it clear that this set does not conform to the framework, a useful result.

5. Finally, consider another class that maintains a graphical view of an integer set as seen in Figure 3. (The example here does not come from an earlier paper.) The class is not intended to modify the set, although it is expected to *view* modifications performed elsewhere. Using a "readonly" annotation on the set will ensure that the view behaves as expected in this regard.

In summary, a "readonly" qualifier enforced by the compiler can aid in preventing dangerous exposure and enable informal guarantees in comments about non-mutation to be made formal and checkable.

2.2 Shortcomings of "read-only"

I now go through the same examples again, and discuss some ways in which "read-only" captures only some of the intended properties.

1. Regarding the "intersect" method, there is an additional property of the method that most users would expect: that the method does not retain the reference to the parameter `set` after the method returns. Suppose the parameter were saved in order to "memoize" the intersection operation. There is a danger that changes in the set would invalidate the memo cache. Retention may also cause a space leak. A "readonly" annotation cannot prevent such retention.

2. The constructor takes an array. There are actually three different reasonable designs behind a constructor of this form:

   - The client is expected to release the array to the control of the ADT. In other words, the parameter represents the transfer of a "unique" pointer;
   - The array is intended to be immutable, not changed by either the ADT or the client.
   - The array is intended to be copied by the ADT, and not retained.

   None of these three situations is fully expressed by using a "readonly" annotation. In the first case, the array is intended to be mutable and thus cannot be protected by "readonly." In the second case, it is insufficient since it does not prevent the client from mutating the array. In the third case, the non-retention (as already explicated) is not expressed.

3. With the `size` method, the caller is again likely to assume that the reference to the receiver will not be retained.

4. With the `toArray`, using “readonly” to qualify the result prevents representation exposure (exposing the ADT representation to mutation by external agents), but the result would still permit *observational exposure* in which the ADT representation is visible to the outside, for reads only. As long as the set doesn’t change, there is little problem, but when it does, the way in which the array is used will be visible. If the array were to be recycled and used in a different set, the client would notice surprising changes.

   In C++, an equivalent situation occurs with regard to iterators into vectors, which are usually implemented as pointers directly into the array. In this case, one is not permitted to retain an iterator when the vector changes. This requirement cannot be expressed in the language (even though C++ has the “const” keyword). Java collections classes have a similar unenforceable requirement, although in this case, the iterators do not expose implementation internals.

On the other hand, in the case of the graphical view in Figure 3, we find that retention is indeed expected, even while the set is not assumed to be immutable. This example shows a case where the semantics of a “readonly” type qualifier fits the design intent well.

Thus sometimes, as seen in the final example, a “readonly” qualifier correctly expresses intent but frequently it does an insufficient job of expressing the intent and encouraging good software practice. The two issues of retention and observational exposure in particular are seen even in those examples used to motivate the addition of a “readonly” qualifier. This section has already pointed out some of the problems of retention. In the following section, I argue that observational exposure has a negative effect on software.

3 Glass Walls are Not Enough

Good software practice requires true privacy, not just lack of external change. The problem of *representation exposure* is well known, in which the (changing) internal representation of an abstract data type or object is made available to agents outside the implementation context. Less well known are the problems of *observational exposure* in which these outside agents are only permitted to read the data.

3.1 Representation exposure

Representation exposure is deleterious for modularity because it permits an external agent to mutate the representation state of an abstract data type or object. The implementation may require certain object invariants for correct functioning. Ensuring that these invariants are maintained is much more difficult when changes can occur from outside the implementation module. Section 2 gave an example of an integer set that uses an array of integers with the invariant that the integers in the array were all distinct. If the array is exposed, someone could easily break this invariant without being aware of it. As Leino argues, an invariant should only need to be checked in a scope in which it is known [22].

To the novice programmer, it may appear sufficient that representation fields in the object be declared private, but (in)visibility of names is not sufficient to prevent accessibility of data. *Aliasing*, in which the same piece of data can be accessed using different names may subvert the protection provided through naming. Thus in order to protect against representation exposure, it is necessary to check every piece of privileged code that assigns a pointer in the representation or accesses a pointer to the representation.

Aliasing is a very useful property and is intimately connected with the idea of *object identity*, that it matters which object one refers to, not just the object’s contents. Thus, I don’t wish to banish it from the language; rather it needs to be controlled. Now in some situations aliasing has entirely benign effects: when the state pointed to is immutable (cannot be changed and will not be changed) the various uses cannot conflict, they may even be unaware of each other, assuming automatic deallocation of memory.

One way to control aliasing of mutable state has been proposed through the use of ownership types (for instance the work of Clarke [13, 11] or Boyapati [6]). The *objects* (not just the fields) in the representation are indicated as such and are protected by a type system that does not permit access to representation objects from outside the owner. In these ownership type systems, all access are controlled; no distinction is made between reads and writes.

Other similar proposals (e.g. Universes [23]) note that an external agent cannot break an invariant with only read access and thus permit "readonly" references to representation state. Indeed, the relative safety of "readonly" pointers into the representation motivates even those "readonly" proposals that do not include an ownership-like system. However, even read access should not be granted, as I argue now.

3.2 Observational exposure

Observational exposure (in which read-only access is granted to *mutable* internal representation objects) has the following bad consequences:

- The inner workings of the "abstract" data type become part of its interface;
- The ADT may be observed in an "invalid" state;
- Concurrency errors may develop;

Each of these closely related points is expanded in the remainder of this section. I argue that one should hide the representation completely; "glass walls" that permit observation while protecting integrity are insufficient.

3.2.1 Observational exposure increases coupling.

If the inner workings of an abstract data type are seen by outsiders, they are no longer "inner workings" but rather part of the interface. If this observing is intended to do anything useful, then the states that are observed must be properly specified. Putting all this information in the interface will make it much more difficult to evolve the ADT, because changes will be resisted by clients.

The developer, on the other hand, may simply refuse to give the specification for the data observed. In this case, clients are likely to guess an API and make unwarranted assumptions. "That's not my problem." the developer of ADT may claim, but bad software structure is a function of the software system as a whole. Indeed observational exposure can be seen as the dual of classic representation exposure: in each case one part of the system is confused when properties of data they are observing change unexpectedly because of another agent mutating that data. In other words, the problem is due to a write access on one side of a supposed abstraction barrier interfering with read access on the other side.

3.2.2 Observational exposure may expose data in invalid states.

Another way of describing the problem of exposure is that either the entire representation is exposed, there is no abstraction barrier at all (in which case modification from the outside can be expected to maintain invariants), or else some aspects of the representation are still hidden. In the latter case, the observer of some of the representation sees an incomplete picture, which may appear to be invalid. Even if the part exposed is valid at the time of exposure, if the exposed reference is retained, the state it refers to may become invalid later.

For instance, suppose the `IntSet` kept a separate field of the number of elements in the array which are used. When the client asks for an array and this field is not equal to the length of the array, the array is first resized to fit the size of the set. So the client doesn't observe this extra behavior at first. But if (say) one element of the set is removed, the implementation may decide to continue using the same array, but not regard the last element part of the array. A client who has retained the array may see a duplicate element at the end of the array. The array will appear invalid.

3.2.3 Observational exposure may lead to concurrency errors.

If an object may be used (and changed) by multiple threads, it is important that the mutable state be protected by mutual exclusion locks. Lea describes the problem and solutions in detail [21]. Greenhouse describes the design needed and how Java code can be checked against a formally declared design [17, 16]. The recommendations are summarized here. (Similar rules and checking are proposed by Boyapati and Rinard [5].)

Logically each piece of state is protected by a single lock object. Often that lock is the object whose fields are the state, but sometimes the lock is a completely different object. For instance an ADT may use a single (private) lock object to synchronize all accesses to fields of any of its internal objects. Or an ADT may use multiple locks to protect different groups of its objects.

Threads should synchronize on the protecting lock before accessing the data. This requirement applies not only to writes, but also to reads because otherwise one may observe the data in an invalid state. This invalid state may not only be due to transitory conditions during a mutation, but also because of memory system effects (because of local caching in a multiprocessor). For example, suppose that a method of the integer set ADT allocates a new array, initializes the elements and then assigns this array to the `ints` field. Another thread that does not properly synchronize *may* see the field change to point to a new array before the elements in that array are initialized.

Furthermore, if multiple locks are involved, and some actions on an ADT require accessing more than one, then it is essential that these locks be acquired in a fixed order or deadlock can easily ensue when threads access the locks in contrary orders.

If an ADT permits observational exposure, the client may not know what locks protect the state and what order in which to acquire locks. Not synchronizing on the lock or synchronizing on the wrong lock makes the results obtained almost meaningless, whereas synchronization in the wrong order may lead to deadlock. Thus observation exposure is only safe if the details of which objects protect which state is revealed and the required synchronization order is detailed. Such an interface may severly limit ADT implementation flexibility.

In conclusion, observational exposure, while less dangerous to an ADT than representation exposure, nevertheless yields improperly structured software. In fact, viewed from the vantage of the program as a whole, the problems are symmetric. Thus I submit we should not accept a proposal to solve the one problem while ignoring the other. Abstraction walls should be opaque.

4 Other Related Work

Noble, Vitek and Potter in Flexible Alias Protection [24] noted the problem that a read-only reference could nonetheless cause dependency problems if the value could be mutated elsewhere and cause the read-only reference to return new values. They coined the term "argument dependence" to refer to this kind of coupling. These researchers also showed the importance of distinguishing the representation of an object from references to external objects unlike earlier alias control systems such as Islands [18] and Balloons [2]. The idea of designating the "representation" was developed further with Clarke [14, 13, 11]. Then with Drossopolou, Clarke showed how an ownership system could be used to encapsulate effects [10]. In these systems, the ownership system prevent any exposure of representation, and thus there was little need for "readonly." Clarke and Wrigstad [12] have shown one way how to integrate ownership and uniqueness, Aldrich and others [1] another.

Schärli and others [30, 29] define encapsulation policies so that dynamically typed languages can support different encapsulation policies for different users. Read-only abilities are defined by restricting access to methods that change state, a generalization of using special read-only interfaces. Unlike those earlier techniques, which rely on static typing, the policy is attached to the reference dynamically in the same way as a capability [8] includes both rights and the object pointer.

In an early paper [28], Reynolds defines "interference" as when a piece of mutable state is accessed by two supposedly separate parts of a program, with at least one part performing a write. O'Hearn and others revisited this work with a mixture on linear and non-linear logics (SCIR) [26]. The approach was changed from one which inferred effects to one in which statements were checked against allowable accesses. Write access was indicated in the linear part of the type, read access in the non-linear part.

Similarly, Walker and others [33] define a capability language (CL) in which linear (non-duplicable) capabilities indicate unique, mutable state and duplicable capabilities indicate shared, immutable state. As in our work, the permission is separate from the pointers that point to the state and alias types are used to connect the two. But using nonlinearity for read access does not permit a write capability in either SCIR or CL to be temporarily duplicated and then later returned to write status except in limited situations.

Fractional permissions [7] solved this problem by preserving linearity (read permissions are split, not copied). Bornat and others [4] have shown how the idea of fractions can be used with O'Hearn and Reynolds' separation logic [27, 19, 25] and indeed how the idea can be extended to handle "counting" as well as "splitting." Permissions can be extended to model ownership using adoption [9].

5 Conclusion

This paper reviews how "readonly" has been proposed to be added to Java. I argue that the "transitivity" rule is too restrictive, and that "read-only" is insufficient to prevent deleterious observational exposure. Combined with an ownership system of some sort (or with permissions [9], these problems could be overcome, but the mechanisms have not yet been worked out fully. Adding "readonly" to Java using one of the current proposals would thus set into stone an overly restrictive type rule. In conclusion therefore, I would recommend that no action be taken yet to add `readonly` to Java.

Acknowledgments

I thank Bill Scherlis, Edwin Chan, Aaron Greenhouse, Tim Halloran and the others in the Fluid project at CMU for their collaboration and conversations. I thank Bill Retert, Scott Wisnieski, Tian Zhao and many anonymous referees for their comments on drafts.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Seattle, Washington, USA, November 4–8, *ACM SIGPLAN Notices*, 37(11):311–330, November 2002.

[2] 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.

[3] Adrian Birka and Michael Ernst. A practical type system and language for reference immutability. In *OOPSLA'04 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, British Columbia, Canada, October 26–28, *ACM SIGPLAN Notices*, 39(10):35–49, October 2004.

[4] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Long Beach, California, USA, January 12-14. ACM Press, New York, 2005. To appear.

[5] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA'01 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Tampa, Florida, USA, November 14–18, *ACM SIGPLAN Notices*, 36(11):56–69, November 2001.

[6] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, California, June 8–11, *ACM SIGPLAN Notices*, 38:324–337, May 2003.

[7] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, San Diego, California, USA, June 11-13, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, Berlin, Heidelberg, New York, 2003.

[8] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, Budapest, Hungary, June 18–22, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer, Berlin, Heidelberg, New York, 2001.

[9] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Long Beach, California, USA, January 12-14. ACM Press, New York, 2005.

[10] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Seattle, Washington, USA, November 4–8, *ACM SIGPLAN Notices*, 37(11):292–310, November 2002.

[11] David Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[12] David Clarke and Tobias Wrigstad. External uniqueness. In Benjamin C. Pierce, editor, *Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10).* January 2003.

[13] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, Budapest, Hungary, June 18–22, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76. Springer, Berlin, Heidelberg, New York, 2001.

[14] 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.

[15] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practial linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 17–19, *ACM SIGPLAN Notices*, 37:13–24, May 2002.

[16] Aaron Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2003.

[17] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '02)*, Orlando, Florida, USA, May 19–25, pages 453–463. ACM Press, New York, May 2002.

[18] 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.

[19] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of the Twenty-eighth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, London, UK, January 17–19, pages 14–26. ACM Press, New York, 2001.

[20] Günter Kniesel and Dirk Theisen. JAC – access right based encapsulation for Java. *Software Practice and Experience*, 31(6), May 2001.

[21] Doug Lea. *Concurrent Programming in Java.* The Java Series. Addison-Wesley, Reading, Massachussetts, USA, second edition, 2000.

[22] K. Rustan M. Leino and Gren Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[23] 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.

[24] 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.

[25] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Conference Record of POPL 2004: the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Venice, Italy, January 14–16, pages 268–280. ACM Press, New York, 2004.

[26] Peter W. O'Hearn, Makoto Takeyama, A. John Power, and Robert D. Tennent. Syntactic control of interference revisited. In *MFPS XI, conference on Mathematical Foundations of Program Semantics*, volume 1. Elsevier, 1995.

[27] John Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, Los Alamitos, California, July 22–25 2002.

[28] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, USA, pages 39–46. ACM Press, New York, January 1978.

[29] Nathanael Schärli, Andrew P. Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA'04 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, British Columbia, Canada, October 26–28, *ACM SIGPLAN Notices*, 39(10), October 2004.

[30] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, , and Roel Wuyts. Composable encapsulation policies. In Martin Odersky, editor, *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, Oslo, Norway, June 14–18, volume 3086 of *Lecture Notes in Computer Science*, pages 26–50. Springer, Berlin, Heidelberg, New York, 2004.

[31] Mats Skoglund and Tobias Wrigstad. A mode system for readonly references. In *ECOOP Workshop on Formal Techniques for Java Programs*, Budapest, Hungary, June 18. 2001.

[32] Bjarne Stroustrup. *The C++ programming Language*. Addison-Wesley, Reading, Massachussetts, USA, third edition, 1997.

[33] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.