Islands: Aliasing Protection In Object-Oriented Languages

John Hogg

Bell-Northern Research
Ottawa, Ontario

Abstract

Functions that are guaranteed not to have side effects are common in modern procedural languages, but not in object-oriented languages. Certain types of state changes are essential in object functions; the difficulty lies in permitting these while banning undesirable side effects. A simple way of doing this is presented. Using this as a base, we can introduce *islands* of objects which can statically ensure non-aliasing properties in a very non-restrictive way. Islands make construction of opaque object components more practical. They also make formal treatment of object behaviour more feasible, since the object structures they encompass can be truly opaque to their clients.

1 Introduction

Object-oriented languages have a light side and a dark side. The light side is that the programming model makes rapid prototype implementation much easier, since components can be easily reused. The dark side is that as these prototypes mature, the components can manifest strange behaviours due to unforeseen interactions and interrelationships. The big lie of object-oriented programming is that objects provide encapsulation. In order to accomplish anything, objects must interact with each other in complex ways, and understanding these interactions can be difficult.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 89791-446-5/91/0010/0271...$1.50

This paper presents two ways of making object interaction more predictable. The first is well-known to the programming world, but has not yet been embraced by the object-oriented community: the division of routines into procedures, which change the system state but do not return values, and functions, which return values but are guaranteed not to change visible system state. (This distinction is made informally in some object-oriented languages. However, compliance is the responsibility of the programmer.) Since functions must be allowed to create objects, a complete ban on changing state is not acceptable, so the problem is slightly more involved than in the traditional case. Nonetheless, the solution is straightforward, and involves a new version of the traditional read-only access mode.

The second part of the paper extends access modes to present something more novel: *islands* of objects. Islands can be used to statically guarantee that an object is not aliased, while placing minimal restrictions on its use. For instance, an object can be stored in a container object, changed by being passed to an external (and possibly heavily aliased) object for use in a procedure method, and later passed on to an object totally different from the one that inserted it in the container in the first place. At the end of these operations, the new "owner" of the object can still be sure that it is unaliased. The syntactic cost of this is about three times that of providing side effect-free functions, and in addition, a fairly natural "destructive read" operation must be added.

OOPSLA'91, pp. 271–285

Islands also make proof systems that depend upon knowledge of explicit relationships between objects more feasible. Within an island, objects are treated as "friends" in the sense of C++ [Str86], and analysed together. Externally, however, the island can be dealt with as a black box. The concepts of islands are widely applicable to different languages and models, and to different theoretical and practical needs.

2 Side Effect-Free Functions

The idea that routines should be divided into those that return a result ("functions") and those that change their environment ("procedures") is now firmly established in the world of traditional procedural programming languages. To the theorist, formal analysis is greatly simplified. Boehm has presented a logic for Russell, a language with functions having side effects [Boe84], but the Russell definition of "side effect" is quite non-standard. In general, the problem is unsolved.

For the pragmatic software engineer, side effect-free functions make error detection easier, and unforeseen interactions between components are considerably decreased. It seems natural to extend this approach to the object-oriented world, and provide languages in which procedures do not return values, and functions do not have side effects.

Before this can be done, however, the term "side effect" must be defined. Intuitively, a programming construct (henceforth, an expression) has a side effect if it changes the state of the system. However, consider the simple expression $3+4$. Under a common model of object behaviour, this will result in the creation of a new instance of class Integer with value 7. The system state has changed, yet we do not think of addition as having side effects.

The answer, of course, is that these system changes are not *visible* to any object in the system. The newly-created instance is not accessible from any existing instance. More to the point, no instance reachable prior to the evaluation of the expression (in fact, no instance *existing* prior to the evaluation) is altered. From the theoretician's view of an expression as a mapping from states to states, no predicate on the initial state will be falsified by the mapping. A programmer will say that an assertion will always be unaffected by the evaluation of an expression. We therefore have a nice intuitive understanding of what we want to achieve, and can easily formalize it.

The tool usually used to achieve freedom from side effects in modern procedural languages is some sort of read-only access mode. An *access mode* is a statically-testable restriction on the operations that may be applied to a variable. In many languages (such as Pascal) read-only is the default mode for parameters: a parameter is read-only unless it is explicitly imported with a *var* mode. A read-only parameter may not be assigned to, and may only be exported to other routines with a read-only mode. However, this is not sufficient in an object-oriented language, in which objects are referred to using pointer semantics. It is not enough to protect the pointer; the object at the other end of it must also be guaranteed to be unaffected by an operation which purports to be free from side effects.

This requirement for protection at two levels has been recognized before, with respect to the Eiffel language [Mey88].

2.1 Eiffel

The Eiffel language has syntactic provision for value-returning functions and procedures without return values. Guaranteeing that a function will not have side effects requires recognizing the cases in which side effects may occur. Meyer presents a list of these cases. They are:

1. assignment to an attribute (instance variable) x;
2. procedure call having side effects on an attribute x;
3. local call having side effects (which in some languages could be considered a form of the previous case); and

4. using an attribute as an argument where the called routine produces a side effect on the corresponding parameter.

Meyer states that it would be simple to enforce these rules, but that side effects which only change the "concrete" state of an object and not the "abstract" state should be allowed. In other words, internal state changes that cannot be detected through any sequence of method invocations should not be considered to be side effects at all. He therefore relies on the programmmer to avoid unwanted state changes.

Unfortunately, the list given above is incomplete. Consider the following code:

```
class C export
   hiddenSideEffects
feature
   protected: MUTABLECLASS;
   hiddenSideEffects(): STRING is
      local temp: MUTABLECLASS
      do
         temp := returnParam(protected);
         temp.mutate;
         Result := "protected has just been altered"
      end;

   returnParam(arg: MUTABLECLASS):
         MUTABLECLASS is
      do
         Result := arg
      end
   end

class MUTABLECLASS export
   mutate
feature
   value: BOOLEAN;
   mutate() is
      do
         value := not value;
      end;
```

Here, all of the Eiffel side effect rules have been followed, but the attribute *protected* loses its supposed protection by being the result of a function. When *hiddenSideEffects* returns, the object that its attribute *protected* refers to will have its value negated, even though the only call of a routine with side effects was on the local variable *temp*.

Clearly, return values are a potential source of unexpected side effects in an object-oriented language. This is especially true because we cannot just decree that all function results are *read*, for the reasons explained earlier: the function may have constructed a structure of objects which is not visible in the context (i.e., the object) from which it was called, but is to be returned for use in that context. In non-object-oriented languages, the ability to generate structures of objects in this manner is not so central to the programming paradigm. This is why the designers of the Turing language [HMRC88] could reasonably ban the creation of new collection elements in a function. When an attempt is made to simulate an object-oriented language in Turing, however, the restriction is immediately apparent. The natural way to simulate the instances of a class of objects uses a collection.

Fortunately, there is no fundamental reason why the desired behaviour of functions cannot be obtained, as we shall see.

2.2 Access Modes and Types

First, a word about typing. An interest in freedom from side effects is usually accompanied by concerns about other aspects of program predictability, and therefore side effect-free functions are found in typed languages. Furthermore, attempts to understand object behaviour are presently concentrated on types and type inheritance. To ensure that there is no misunderstanding, we will state the obvious: the access modes presented here and later on in the paper are completely orthogonal to any notion of typing, and of inheritance. Side effect-free functions (and later, islands) will be most useful in the context of a typed language, but types are not necessary. To drive the point home, the remainder of the examples in this paper will be expressed in a Smalltalk-like language [GR83]. This is not meant to suggest that Smalltalk is the most appropriate

vehicle for anything presented here. The ideas are clearly translatable to other object-oriented languages. In fact, since the work presented here is orthogonal to inheritance, it applies to object-*based* languages in general and not just object-*oriented* languages, using the terminology of [Weg87].

Since parameters and results have associated access modes, each method invocation must be checked for legality. This is done by treating the set of modes associated with an invocation as an extension of the method selector. Mode conformance is therefore checked at runtime, and an implementation would supply a mechanism similar to doesNotUnderstand to deal with cases of mismatch between access modes of the invoking message and the methods supplied by a receiver object. In a statically-typed language, signature mode conformance would also be statically checked.

2.3 Rules for Side Effect-Free Functions

We will now provide a set of rules for ensuring freedom from side effects in functions. They are based on the traditional notion of a read-only (henceforth, *read*) access mode.[1] In an object environment, however, this mode must refer to both the variable and the object to which it refers. The restrictions on *read* are simple:

Definition 1 (Read) *A variable whose declaration has the label* %read *is* read. *If within a method the receiver (***self***) is* read, *then all instance variables are implicitly* read. *A* read *expression is a* read *variable or the result of a* read-*valued function.*

1. *A* read *variable may not be assigned to.*
2. *A* read *expression may only be exported as* read. *An expression is exported when it is used as a parameter of a method invocation or the result of a function method.*
3. *A* read *expression may not be the right side of an assignment.*

The first restriction obviously protects the *read* variable itself from being assigned to. The second restriction ensures that the object that the variable refers to is protected when it is used in other contexts. The third restriction protects the object within the current context. If a reference may be retained in an instance variable, then the protected object may be altered later in an execution by some other method. The persistent environments that are the central concept of object-oriented programming make it difficult to specify during one entry to a context (an instance) that some reference should be treated in a special way when the context is returned to later. Permitting assignment to temporary variables (which must subsequently be treated as *reads*) is somewhat simpler, but results in considerable syntactic baggage and also incompleteness, for negligible advantages.

The idea of a *read* access mode is usually expressed in a syntax that is the reverse of the one used here: no variable may be changed unless it is declared *var*. Using *var* as a default and making *read* restriction explicit is a matter of taste, and makes it easier to express what *read* entails. The difference is minor.

Definition 2 (Function) *In a function method definition, all parameters (including the receiver,* **self***) are* read.

Within a function method, then, all instance variables (and the objects they refer to) are *read*. A function invocation will have no visible side effects.

These rules imply that there are two sorts of function: those that return *read* results, and those with "free" results. This addition to the signature of a function method is the price that must be paid for adding side effect-free functions to the pointer semantics of an object-oriented language.

[1] An alternative name for this mode is *const*. However, that suggests that an object that is the value of a *const* variable cannot change. If the object or the structure of objects reachable from it is aliased, i.e., is the value of another possibly non-*const* variable, then the nomenclature is misleading, since it may change "under foot". A mode is a property of a variable and describes the way that an object and its structure may be accessed. The mode is *not* a property of the object or its structure.

2.4 A Function Example

The following example is expressed in an extended Smalltalk in which method definitions have signatures annotated by access modes. In Smalltalk, every method can have side effects, and also returns a value. Since we have divided the world into procedure and function methods, it becomes convenient to make this distinction in the language, and therefore function selectors have been distinguished from procedure selectors by an initial upper-case letter. The mode of the receiver during the execution of a method is prepended to the method name, and instance variables and parameters have their modes appended. The mode of a function result (if explicit) follows a colon appended to the signature.

The example concerns a class of complex objects having a method **MaxAbs** which returns the receiver or its parameter, depending upon which has the greater absolute value. We must ensure that regardless of the class of the components of the receiver and the parameter, this method cannot change the visible state of the system. One way of ensuring this is as follows:

```
class name                  Complex
instance variable names     x
                            y

instance methods

...
    %read MaxAbs: aComplex%read :%read
        ((self Abs) > (aComplex Abs))
            ifTrue: [↑self]
            ifFalse: [↑aComplex]

    %read Abs :%read
        ↑((x*x) + (y*y)) Sqrt
```

This code does not specify the behaviour of the system by itself; that will depend upon the classes of objects used to represent the x and y values of the receiver of the method, and upon the parameter. However, the demands made by these methods are fairly severe. They return *read* results, which restricts the use to which they can be put.

An alternative to this would be the following implementation, which returns a result that can be used in further processing:

```
class name                  Complex
instance variable names     x
                            y

instance methods

...
    %read Max: aComplex%read
        ((self Abs) > (aComplex Abs))
            ifTrue: [↑self Copy]
            ifFalse: [↑aComplex Copy]

    %read Abs :%read
        ↑((x*x) + (y*y)) Sqrt

    %read Copy
        ↑(Complex new) x: x y: y

    %read x: newX%read y: newY%read
        x ← newX Copy.
        y ← newY Copy
```

This is functionally an improvement, since the result of Max may now be assigned to a variable in the calling method since there it has no *read* mode restriction. However, it is clearly quite inefficient, since copies must be made to ensure that *read* instance variables remain protected.

The solution to this involves introducing an *immutable* mode, in the sense of [LG86]. Primitive objects are almost always defined so as to be immutable: after an immutable instance is created, it can never be changed, because there is no operation that can change it. As a result, it needs no explicit *read* protection. This mode would also be useful in the second half of this paper. An immutable object need not be protected from aliasing, because the effects of this aliasing will never be seen by any holder of an alias. Immutability can considerably simplify analysis. However, this idea will not be further described here.

3 Islands

Functions that are free from side effects are valuable in themselves to both the theoretician and the practical programmer. They also provide a good introduction to the main contribution of this paper: the notion of *islands*.

The main problem in constructing a usable formal semantics for an object-oriented language is arguably *aliasing*. An object is aliased (with respect to the context of another object and its associated state) if there are two *paths* to it. A path is a sequence of variable names with each variable name denoting a context (i.e., an object) in which the succeeding variable is evaluated. If aliasing exists, then it becomes very difficult to determine whether an operation will change the state of a seemingly-unrelated object.

The aliasing found in a "pure" object system is not the same sort of aliasing that occurs in traditional procedural languages. Every object system variable is a pointer to an object, and there is no way in which two variables can refer to the same pointer. However, the distinct pointers may refer to the same object. Readers familiar with object systems will find the following description obvious. Those new to objects (even though they may have a strong background in semantics) may find it helpful.

Figure 1 gives two examples of aliasing, one harmless and one not. In Part (a), we have a primitive object (an integer) known in the current context by two aliases, the variables x and y. If x ← x + 1 is now executed, there is no problem; x points to a new instance, and the value and behaviour of y both remain unchanged.

Now consider Part (b) of the figure. Here, x and y both refer to an instance of a user-defined class that contains a value, and methods for accessing and altering that value. A method x incrementValue (with the obvious semantics of the diagram) will now leave the *value* of y unchanged, but it will change the state of the object to which y refers, and thus its behaviour. In other words, a construct will have changed the meaning of a variable in a way that is not syntactically detectable. Note that object-oriented aliasing problems only occur "at one level of indirection".

Aliasing can be divided into two types: *static* aliasing, and *dynamic* aliasing. An object is aliased statically if the two different access paths are both composed entirely of chains of instance variables. The aliasing is dynamic if at least one of the access paths has a prefix consisting of temporary variables or parameters. A dynamic alias will therefore disappear at the end of the execution of the method in which it appears. By contrast, a static alias may make its existence felt during some later invocation.

This persistence makes a static alias much more troublesome than a dynamic alias. A dynamic alias (i.e., the use of parameters, or most uses of temporary variables) has no effects beyond the scope in which it occurs. From an operational viewpoint, a static alias can cause unpleasant surprises at an arbitrarily distant point in an execution. By the time the bomb goes off, the chains of variables in instances by which an aliased object is known may be arbitrarily long. From a denotational perspective, a static alias causes problems because the meaning of an expression is dependent upon its context; the meaning of a function invocation may be affected by a preceding procedure invocation even though the invocations share no variables.

3.1 Previous Work

3.1.1 SPOOL

The most closely-related work in the literature to that presented here is the proof system for SPOOL [AdB90]. Some of the same ideas can be seen in the semantics constructed for the non-object-oriented language Turing [HMRC88]. The aliasing problem is that an operation involving one set of identifiers may affect the visible state of a disjoint set of identifiers. This occurs because some object is reachable from the current context through more than one chain of objects containing pointers (i.e., variables) to other objects. If these chains and their relationships can be made explicit, then the problem

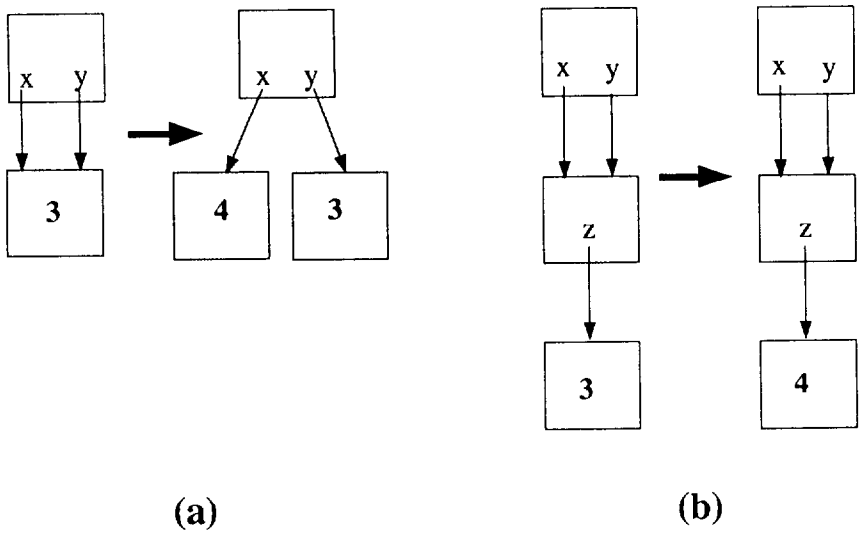

Figure 1: Aliasing problems

is solved. Changes made to an instance through one alias can be reflected in other aliases.

The SPOOL proof system is based on a Hoare logic, following the tradition of [Coo78]. Its predicate language contains not only simple variable names but also *global expressions*. A global expression g.x has the value of the variable x in the context of the global expression g. Conditionals may appear in global expressions; otherwise, they reduce to chains of variable names. The SPOOL proof system expresses predicates in terms of these chains of variables. A Hoare formula describes the body of a method as seen from within the context of its receiver. Proof rules express the same formula from the context of the caller by prepending the variable indicating the receiver to each variable chain used in the body. A formula describing an object at a high level in the system therefore has a body consisting of some set of method invocations. Its predicates, however, have no invocations, but rather variables that are long chains.

There are three obvious problems with this approach. From the description above, it is clear that the technique does not scale. It may be reasonable for small sets of classes, but is unusable in anything larger than a toy system. Long chains of variable names are not meaningful.

The use of paths also massively violates object encapsulation: to determine the behaviour of an object, not only must its implementation be examined, but also the implementation of all of its acquaintances. This bodes ill for the late binding of behaviour that is the essence of object-oriented systems.

Finally, a variable chain system is impractical because it does not provide modularity of proofs. Object-oriented libraries are supposed to provide the user with a set of components that can be used to construct larger components and entire systems. If each use of a component requires proofs about it to be reconstructed, then very little has been gained. A practical proof system must encapsulate not only code, but also specifications, so that a component can be retrieved from a library as a true black box.

3.1.2 FX

The polymorphic effect system of FX [LG88] is another attempt to solve the aliasing problem. Expressions have types, and in addition have *effects*, which describe side effects (such as reading, writing, or allocating), and *regions*, which describe where these effects may occur. Regions are explicitly specified by the programmer. One expression can be shown to have no effect on another if the

effects of the first occur in a region not used by the second. Furthermore, an expression with effects can be shown to have none that are visible to the rest of the system if the region in which they occur is only visible from within the expression. An expression that only has such *masked* effects corresponds to a function as described in the first part of this paper.

Like the earlier work of Reynolds [Rey78], effects are conservative. If the programmer specifies regions with large granularity, potential interference will be indicated where none actually exists. If small regions are used, then the task of region management can overwhelm the programming underneath it. Again, calculating the *reach* (the regions accessed by an expression) can require abandonment of the simple encapsulation that we would like to see in a set of objects.

3.2 Island Strategy

In practice, programmers manage to muddle along even in the presence of aliasing without shooting themselves in the foot too often. We can speculate that this success is due to the fact that aliasing tends to be local, and between objects of closely-related classes. A programmer working within one of these classes will have a good idea of how other classes will be affected by any aliases present, and may well depend upon this knowledge. Approaches to object design such as CRC (Class, Responsibility, Collaboration) [BC89] would seem to empirically support this hypothesis.

Assuming, then, that *islands* of aliasing are used in an informal way at present, an obvious idea is to formalize the concept, and control the existence of aliases in a rigorous manner. First, we will expand upon what an island should be, and what guarantees it should provide.

An island is the transitive closure of a set of objects accessible from a *bridge* object. A bridge is thus the sole access point to a set of instances that make up an island; no object that is not a member of the island holds any reference to an object within the island apart from the bridge.

Container structures make natural examples of islands. Since it must be possible to insert other objects (and structures of objects) into a container and later remove them without destroying the alias protection of the island, several other properties are required. It must be possible to pass a structure into an island with a guarantee that no other references to it are held, and later it must be possible to retrieve the structure with the same assurance. While a structure is held within an island, it should be usable: that is, it should be possible to invoke both functions and procedures on it, and to use it as a parameter to other functions and procedures external to the island, while still maintaining the guarantees of non-aliasing.

For a more concrete view of the concept, see Figure 2. The boxes represent instances, and the ovals indicate the boundaries of islands. The arrows represent references from instance variables to objects. Here, we see a fragment of some global system state that includes a container island. An implementation of this example will be presented later.

The topmost instances are users of the container. One instance inserts items which it guarantees to be unaliased, and this allows the other to have the same guarantee when it removes the items from the container. The oval island boundary represents a wall across which no static references can exist, except through the bridge object at its top. Even dynamic references must be granted by the bridge.

Since the only access to the island is through the bridge, it follows that the state of an island remains unchanged between methods invoked on the bridge. When a bridge is itself unaliased, its sole holder is assured that any construct that does not contain a reference to the bridge cannot affect its (transitive) state. A lack of aliasing is thus a property to be jealously protected. The main bridge in the diagram here does not have that property. However, if we look inside the island, we see a set of smaller islands which are unaliased, representing the items held in the container.

These items are protected, but they are not inaccessible. Figure 3 shows the dynamic use of an item by an object outside of the island. The heavy lines

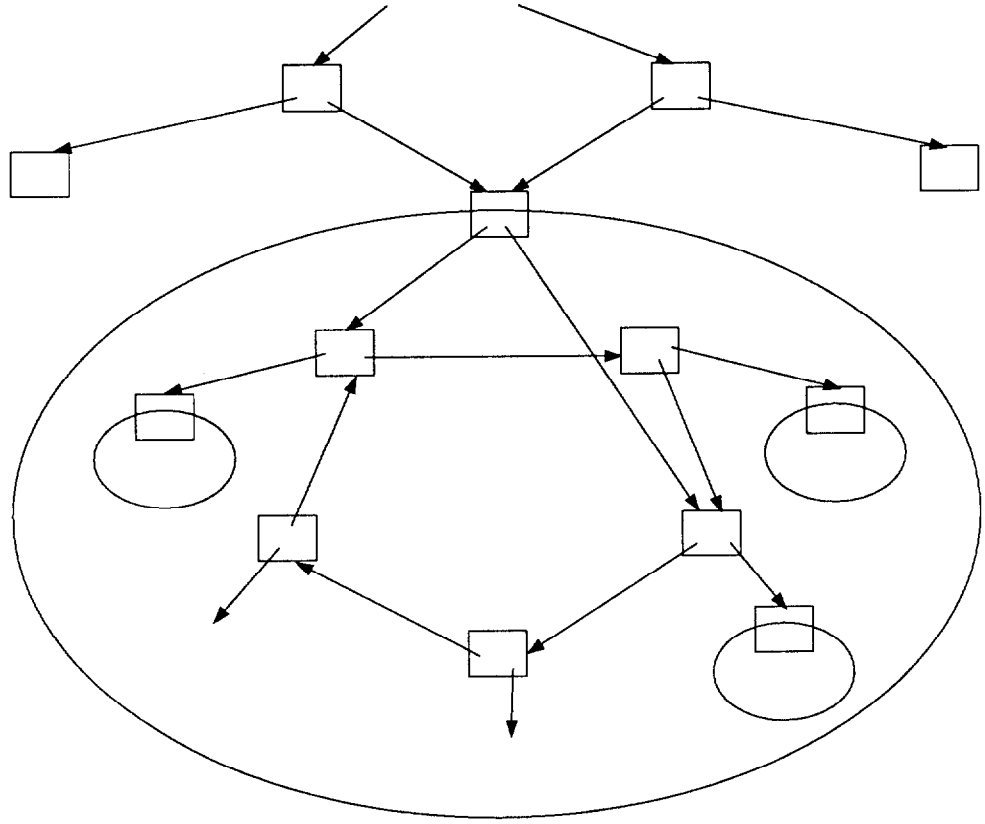

Figure 2: Static references around an island

are (temporary) references from parameters to objects. The numbers on the objects represent their order in the calling stack, and it can be seen that the external object has sent a message to the protected item, which may change its state as a result. Since this access is through the bridge, however, it is controlled: invariants on the state of the island as a whole can be proven using a proof system. Also, when the methods terminate and the calling stack has unwound, the situation of Figure 2 will return, and there will again be no references across the island boundary.

An island provides a true encapsulation of its components: all access is through a single bridge, and therefore the state of the island can never change without this change being visible to the bridge. Within the island, path variables (or other techniques) will in general be necessary to construct proofs about intra-island behaviour. However, the interface that the bridge presents to the world need not, should not, and in fact must not export references to island objects. This means that island-based systems should scale: the complexity of a proof in terms of path lengths is limited by the size of an island, instead of the size of an entire system.

3.3 Island Implementation

We now turn to the implementation of islands. First, a "destructive read" is needed. In addition to the previously-presented *read* access mode, two more access modes are required: *unique* and *free*. They are both orthogonal to the *read* mode, but mutually exclusive, so a variable x many be unrestricted, *read*, *unique*, *free*, *read* and *unique*, or *read* and *free*. The claim that the syntactic cost of islands is about three times the cost of side effect-free functions refers to this tripling of required access modes, and to a tripling of the rules associated with them.

The destructive read is simply an atomic operation that returns the value of a variable (i.e., the identity of the object to which it points) and sets

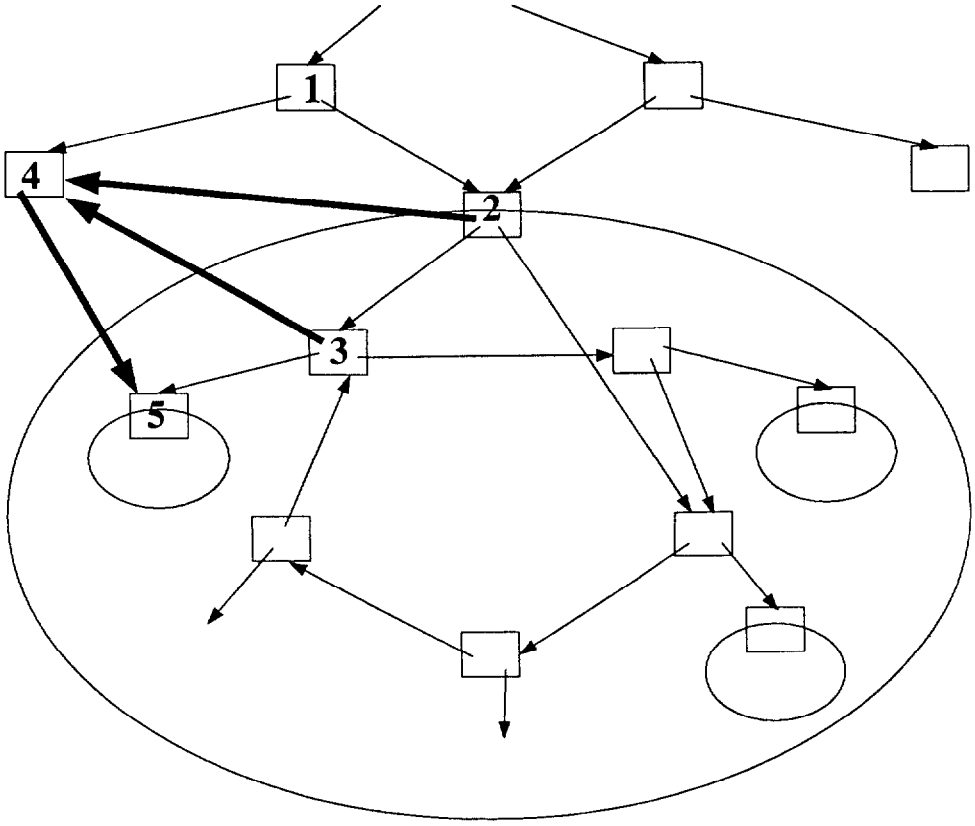

Figure 3: Dynamic references during island use

the variable to nil. It will be written as ↓ x. This can also be extended to expressions: ↓ (e) means that the value returned by the expression e is the identity of an object that is not held elsewhere—that is, the object is not the value of any instance variable.

The mode *unique* indicates that the object to which it refers has only one static reference in the entire system—i.e., its address is contained in only one instance variable. The mode *free* indicates that *no* other references to the variable exist anywhere in the system. Clearly, there can never be a *free* instance variable. A destructive read of a *unique* instance variable produces a *free* result, and a *free* value can be assigned to a *unique* instance variable.

Due to the lack of aliasing, an object held through a *unique* variable can only be affected by an expression in which the variable appears. This is extremely useful in proving properties about a system.

The rules for *read* access have already been given. The rules for *unique* are as follows:

Definition 3 (Unique) *A variable whose declaration has the label* %unique *is* unique. *A* unique *expression is a* unique *variable or the result of a* unique-*valued function.*

1. *A* unique *variable may only be assigned the result of a* free *expression.*
2. *A* unique *expression may not be assigned to anything.*
3. *A* unique *expression may only be exported (as receiver, parameter or result) as* unique.
4. *If a method receiver is* unique, *then every parameter and the result must be* read *or* unique *or* free.

These rules need some justification. The first says that a reference cannot be *unique* if it is held anywhere else, so an assignment to a *unique* variable can only take place if the value being assigned is released wherever else it may be held. Similarly,

an expression is not *unique* if it is assigned to another variable. Access modes must be protected on export, and the third rule ensures this.

The least obvious rule is the last one. *Unique* is a transitive property. If an object's acquaintances are aliased, then the observable state of the object may be unexpectedly changed, even if there is no aliasing of the object itself. Therefore, an object being accessed as *unique* must not import or export any unprotected references. An unprotected parameter could be retained by the object, however. This is shown in Part (a) of Figure 4. The light lines indicate the original variable references, while the heavy line is a reference retained after the execution of a method. A has passed B as a parameter of a method to C, which has retained the reference; thereafter B is visible to C, but may be accessed without going through it. Likewise, Part (b) of Figure 4 shows how an unprotected result B passed from C to A may result in exactly the same situation.

Unique instance variables and *unique* parameters are somewhat different animals. Neither may be copied without respecting the *unique* status, but in addition, only an instance variable may be destructively read to generate a *free* reference. They are generally so similar, however, that the same access mode has been used to identify them to decrease keyword (and concept) proliferation.

The definition of *free* is simple:

Definition 4 (Free) *A variable whose declaration has the label* %free *is* free. *A* free *expression is the destructive read of a* unique *instance variable or a* free *variable, the result of a* new *method, or a the result of a* free*-valued function invocation.*

1. *A* free *variable may only be accessed via a destructive read.*

Free variables and values are extremely transitory: as soon as they are touched, they disappear. Note that there are no restrictions on the modes of parameters methods with *free* receivers. This is because a *free* receiver has essentially already vanished over the event horizon from the original state. The structure that it refers to may return in the result of the function, but it will not have any alias in the rest of the system.

In addition, we need to modify the meaning of *read* slightly by adding this restriction, since a destructive read has side effects:

Definition 5 (Revised Read) 4. *A* read *variable may not be destructively read.*

We can now relax the definition of a function slightly:

Definition 6 (Revised Function) *In a function method definition, all parameters (including the receiver,* **self***) are* read *or* free.

Finally, we have the following definition:

Definition 7 (Bridge Classes) *If every method of a class has the property that every parameter and function result is* read, unique *or* free, *then every instance of that class is a* bridge.

A bridge class presents an external interface to a set of classes. A specification for a bridge may be given as a set of Hoare formulas, and the class may thereafter be used as a black box.[2] Classes used within the island will not be visible to the user, and will be protected by the island. Bridge class instances may then be used to construct larger islands and eventually entire systems. At each level, proofs will use path variables to refer to visible bridge instances, but not the instances within their islands.

Every structure underneath an instance of a bridge class is an island, but many islands are not headed by a bridge class instance. In particular, every structure underneath a *unique* instance (i.e., an instance referred to by a variable of *unique* mode) must be an island. However, bridge-class islands and *unique* islands are different in two major ways. A bridge-class object may be aliased, while a *unique* instance may not. Also, a *unique*

[2] We recognize that this is a very oversimplified and unrealistic view of the software engineering process, and that true black-box class usage will not be practical in the foreseeable future. Nevertheless, the principle still holds.

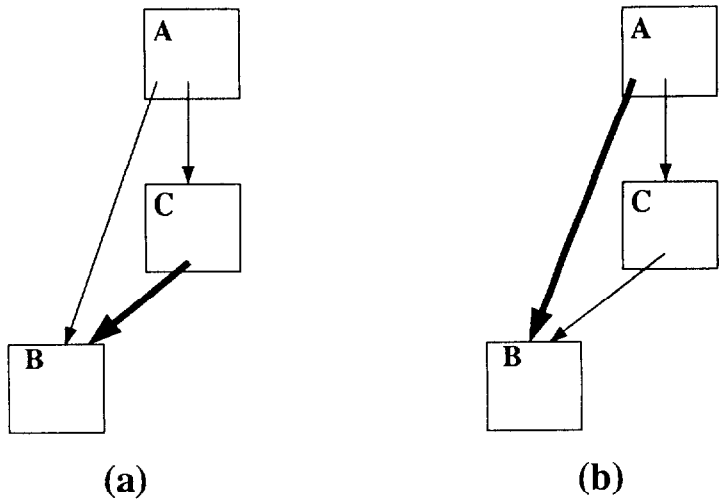

Figure 4: Alias creation through retained references

instance may later lose its uniqueness and therefore the protection of its island by being assigned to a non-*unique* variable using a destructive read. By contrast, the objects underneath an instance of a bridge class are permanently protected.

Islands have the nice property that they are not forced upon a user. A language may provide a set of access modes, but they can be ignored if the programmer so wishes, e.g. during prototyping. A set of collaborating classes can later be turned into an island by the addition of appropriate access modes. If this is not possible, it could indicate that the design provides insufficient encapsulation.

3.4 An Island Example

Figures 2 and 3 depicted an island implementation of a container class. We now present a concrete implementation of this. It is a circular queue implementation of a bounded buffer, but it also allows items in the buffer to be accessed. Items are inserted and removed in the normal way, but in addition are associated with keys. While an item is in the buffer, it may be changed by having its *intern-Mutate* method invoked. Additionally, an object external to the island may "borrow" an item as a parameter to its *mutate* method. In other words, dynamic aliasing of an instance can be permitted, while the subsequent non-aliasing of the instance is guaranteed.

class name DictionaryBuffer
instance variable names head
tail

instance methods

```
initialize: size%read
    | temp |
    head ← Slot new.
    tail ← head.
    size do: [
        temp ← Slot new.
        head next: temp.
        head ← temp ].
    head next: tail.
    tail ← head
```

```
insertKey: newKey%free
      value: newValue%free
   tail insertKey: ↓newKey value: ↓newValue.
   tail ← tail Next.

transfer: destination%unique
   head transfer: destination.
   head ← head Next

%read Find: searchKey%read :%read
   | temp |
   temp ← head.
   [temp Key ~= searchKey] whileTrue:
      [temp ← temp Next].
   ↑temp

internMutate: searchKey%read
   (self Find: searchKey) internMutate

      externMutate: searchKey%read
   mutator mutate: (self Find: searchKey)
```

```
class name                  Slot
instance variable names     next
                            key%unique
                            value%unique

instance methods
   next: newNext
      next ← newNext

   %read Next :%read
      ↑next

   %read Key :%read
      ↑key

   insertKey: newKey%free
         value: newValue%free
      key ← ↓newKey.
      value ← ↓newValue

   transfer: destination%unique
      destination insertKey: ↓key value: ↓value

   internMutate
      value mutate

   externMutate: mutator%unique
      mutator mutate: value
```

From the class definitions given above, DictionaryBuffer is an island, but Slot is not. (The unrealistic assumption is made here that DictionaryBuffer does not inherit from classes such as Object that provide methods for downright promiscuous access to private parts. An island-based reimplementation of the Smalltalk library would look very different from the present version.) An object inserted into the DictionaryBuffer is *unique*, and therefore any objects reachable from it form an island. This *unique*ness is preserved while the object is held in the container island, but the subsequent recipient of the held island structure may remove this protection.

4 Conclusion

An object-oriented version of functions that are statically guaranteed not to have side effects has been presented. Using this, islands have been introduced. Islands are of value both to the pragmatic software engineer and to those interested in formal proofs of object behaviour.

To the software engineer, islands allow a set of objects to be nicely encapsulated. An instance of a bridge class may be used as a true black box: clients can use objects within an island, but can never do so except through a single, controllable interface. Islands are not forced upon users; the tool is an aid, not a straitjacket. However, they are one more step towards the holy grail of truly reusable, black-box components.

A component can only be a black box if it comes with some sort of behaviour specification apart from the code. The most reliable sort of specification is one that implementations can be proven

to satisfy, such as a set of Hoare formulas or pre- and post-conditions on a method. Such a specification on a bridge class need not and cannot reveal the implementation of the island it protects. It is hoped that this will overcome scaling problems that make existing proof techniques for object systems inapplicable for non-trivial systems.

Only sequential languages have been treated in this paper. However, all of the ideas are directly applicable in some models of parallel object behaviour, and some of the ideas are applicable in all (reasonably mainstream) models.

Islands fit POOL [Ame89] particularly well. In that language, an object may only have one active thread of control at any point. If an object is executing one thread (either directly within its own context, or indirectly through a procedure call in the context of another object) then any messages that arrive are placed in a queue. As a result, an island will accept at most one external thread at a time. Inside an island, however, any number of threads can be active. They may be persistent, or they may be started by an external thread. In any case, the requirements and protections described in this paper all still hold.

An area for future work is the applicability of islands to models in which multiple threads can be active within an instance. The problem is that *unique* loses its meaning: the reference may be held by only one object, but two threads through that object can still clash with one another.

Bridges and islands were originally devised as a technique for making proof systems for object-oriented languages practical. However, other uses have also been suggested. Islands can greatly simplify some persistent-store and storage management problems. An island with a *unique* bridge can be moved to persistent store or freed with no possibility of dangling references.

This presentation of access modes and their applications has been informal. A formal treatment requires presenting a formal semantics of a language based on a formal model, which is beyond the scope of this forum. A complete treatment for a simplified object-oriented language is given in [Hog91].

References

[AdB90] Pierre America and Frank de Boer. A sound and complete proof system for SPOOL. Technical Report 505, Philips Research Laboratories, May 1990.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal ASpects of Computing*, 1(4):366–411, 1989.

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *OOPSLA '89 Proceedings*, pages 1–6, October 1989.

[Boe84] Hans-Juergen Boehm. A logic for the Russell programming language. Technical Report TR 84-593, Ph.D. thesis, Cornell University, February 1984.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, February 1978.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HMRC88] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Language: Design and Definition.* Prentice-Hall, 1988.

[Hog91] John S. Hogg. *Formal Semantics of Opaque Objects in the Presence of Aliasing.* PhD thesis, University of Toronto, 1991. (to appear).

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 47–57, January 1988.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, January 1978.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *OOPSLA '87 Proceedings*, October 1987.