Flexible Alias Protection

James Noble[1], Jan Vitek[2], and John Potter[1]

[1] Microsoft Research Institute, Macquarie University, Sydney
kjx,potter@mri.mq.edu.au
[2] Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are hard to understand, maintain, and analyse. *Flexible alias protection* is a conceptual model of inter-object relationships which limits the visibility of changes via aliases, allowing objects to be aliased but mitigating the undesirable effects of aliasing. Flexible alias protection can be checked statically using programmer supplied *aliasing modes* and imposes no runtime overhead. Using flexible alias protection, programs can incorporate mutable objects, immutable values, and updatable collections of shared objects, in a natural object oriented programming style, while avoiding the problems caused by aliasing.

1 Introduction

I am who I am; I will be who I will be.

Object identity is the foundation of object oriented programming. Objects are useful for modelling application domain abstractions precisely because an object's identity always remains the same during the execution of a program — even if an object's state or behaviour changes, the object is always the same object, so it always represents the same phenomenon in the application domain [30].

Object identity causes practical problems for object oriented programming. In general, these problems all reduce to the presence of *aliasing* — that a particular object can be referred to by any number of other objects [20]. Problems arise because objects' state can change, while their identity remains the same. A change to an object can therefore affect any number of other objects which refer to it, even though the changed object itself may have no information about the other objects.

Aliasing has a large impact on the process of developing object oriented software systems. In the presence of aliases, understanding what a program does becomes more complex, as runtime information about topology of the system is required to understand the effects of state changes. Debugging and maintaining programs with aliasing is even more difficult, because a change to one part of a program can affect a seemingly independent part via aliased objects.

In this paper, we present *flexible alias protection*, a novel conceptual model for enforcing alias encapsulation and managing the effects of aliasing. Flexible

alias protection rests on the observation that the problems caused by aliasing are not the result of either aliasing or mutable state in isolation; rather, problems result from the interaction between them, that is, when aliases make state changes visible. We propose a prescriptive technique for enforcing flexible alias protection based on programmer-supplied *aliasing mode declarations* which relies on static mode checking to verify the aliasing properties of an object's implementation. The mode checking is modular, allowing implementations to be checked separately, and is performed entirely at compile-time, with no additional run-time cost.

Flexible alias protection is closely related to the work of Hogg [19] and Almeida [2]. Our proposal differs from these in two main respects. Most importantly, flexible alias protection allows objects to play a number of different roles, which reflect the ways in which objects are used in common object oriented programming styles. For example, a container's *representation* objects may be read and written, but must not be exposed outside their enclosing container, while a container's *argument* objects may be aliased freely, but a container may not depend upon their mutable state. Flexible alias protection does not require the complex abstract interpretation of Almeida's Balloon types, and is thus more intuitive for programmers and less sensitive to small changes in the implementations of the protected objects.

This paper is organised as follows. Section 2 presents the problems created by aliasing in object oriented programs, and Section 3 discusses related work. Section 4 then introduces the concepts underlying flexible alias protection, and Section 5 presents a model for static mode checking. Section 6 discusses future work, and Section 7 concludes the paper. We begin by describing the problem caused by aliasing in object oriented programs.

2 Aliasing and Encapsulation

Aliases can cause problems for object oriented programs whenever a program abstraction is implemented by more than one object in the target program. That is, when there is one *aggregate object* representing the whole of an abstraction and providing an encapsulated interface to it, and encapsulating one or more other objects implementing the abstraction represented by the aggregate object. We say the objects implementing the aggregate objects are members of the aggregate object's *aliasing shadow*[1].

Aliasing can cause problems whenever references into an aggregate object's shadow exists from outside the shadow. Messages can be sent to that shadow object via the alias, bypassing the aggregate, and modify the state of the subsidiary objects, and thus of the whole abstraction implemented by the aggregate object, see Figure 1. References to an aggregate object's shadow can arise in two

[1] An aggregate object's shadow is similar to Wills' *demesnes* [41], or the objects in an Island [19] or Balloon [2]. In this paper, we use the term *shadow* to denote the intrinsic nature of this set of objects, and other terms to denote particular aliasing control mechanisms

ways: either an object which is referenced from outside can be added into the shadow, or a reference from within the shadow can be passed out.

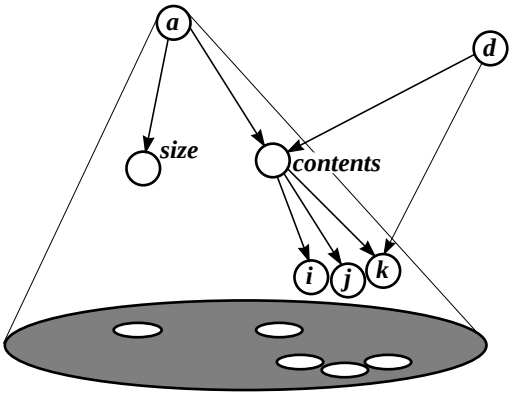

Fig. 1. *Unconstrained Aliasing.* The hash table a has a shadow composed of representation objects *size* and *contents*, and some argument objects contained in the table, i, j, and k. Both *contents* and k are seen from the outside by d. Thus, d is able to change the state of a's implementation without going through a.

The breaches of encapsulation caused by aliasing may affect correctness of the aggregate objects, causing the program to err, or, perhaps even more seriously, opening security holes in the application. We illustrate these problems with two examples.

Consider an object implementing a simple hash table (see Figure 2). The hash table object has two components: an array of table entries and an integer, stored as variables named contents and size. The hash table object is an aggregate, its shadow contains the integer, the array, and the table entry objects in the array. If a reference to the contents array exists from outside the hash table object (that is, if the array is aliased) the contents of the hash table can be modified by sending a message directly to the array object, without sending a message to the hash table object itself.

Aliases to the hash table's elements can arise in a number of ways. For example, if references to the key and item objects are retained outside the hashtable, the hash table elements will be aliased. Alternatively, a hash table operation (such as get) can directly return a reference to an object stored in a hash table, and this will immediately create an alias.

Aliases to the array object making up the hashtable's internal representation may also be created. Typically, representation objects are created within the aggregate object of which they are a part, and so preexisting references from outside the aggregate are unlikely. An operation upon the aggregate can, however, return a reference to one of the internal representation objects just as easily

```
class Hashtable<Hashable,Item> {

      private Array<HashtableEntry<Hashable,Item>> contents;
      private int size;

      public void put(Hashable key, Item val);
      public Item get(Hashable key);

      public Array<HashtableEntry<Hashable,Item>>
         expose() {
            return contents;
      };
}
```

Fig. 2. A Simple Hashtable

as it can return a reference to one of the elements — for example, Figure 2 shows how a hashtable could include an expose operation which would return the entire array.

Exposing internal representation may have security implications if objects are used as capabilities [13, 16]. Viewing an object's interface as a capability is appealing, because it leverages the safety property guaranteed by a strong type system to turn it into a protection mechanism for implementing access control. In effect, the type system prevents access to operations not explicitly listed in an object's interface. The danger with this model is that, as there are no strong protection domains between entities, it surprisingly easy to open an aggregate object to attacks [39]. Aliasing plays an important role here as it can be exploited to gain access to the trusted parts of an abstraction. A case in point is the recent defect in an implementation of SUN's digital signatures for Java applets which permitted any applet to become trusted, simply because an alias to the system's internal list of signatures was being returned, instead of a copy of that list.

This paper is concerned with a programming discipline which simultaneously prevents aliases to the internal representation of an abstraction from escaping the abstraction's scope, and protects an abstraction from existing aliases to objects it receives as argument, while preserving many common object oriented programming styles. We will start by reviewing known approaches to this problem.

3 Related Work

3.1 Aliasing and Programming Languages

The traditional solution adopted by programming languages to encapsulate references (and thus aliases) is to provide *access modes* which control how names can be used within programs. For example, consider the *private* and *protected*

modes of Java and C++ which restrict access to the names of variables and methods.

An aggregate object's shadow can be stored within the aggregate object's protected local state, but this is not enough to protect the shadow objects from aliasing [2, 19, 20]. As we have already seen, a method attached to an aggregate object can return a reference to a shadow object. An aggregate object can also store objects created outside itself into nominally private state, and these objects may have been aliased before they become members of the shadow. An object's encapsulation barrier protects only that individual object, and that object's private local state: the members of an aggregate object's shadow are not effectively encapsulated. That is to say, access modes protect local state by restricting access to the names of the local state, rather than to the objects to which the names refer.

In practice, many programming languages do not provide even this level of encapsulation. In languages such as C++ and Java, the language access modes provide protection on a *per-class* basis, so any object can retrieve a private reference from any other object of the same class, thus instantly creating an alias into another object's shadow. Eiffel includes *expanded types* which are always passed by value rather than reference. Unfortunately, subcomponents of expanded types can be passed by reference, and so can be aliased.

Rather than rely on access modes, it is sometimes suggested that aliasing can be controlled using private classes — that is, a private object should be an instance of a private class, rather than stored in a private variable. Private classes are not a general solution, however, since they also protects names rather than objects. For example, private classes are typically shared among all instances of the class where they are declared. More importantly, if a private class is to be used with existing libraries or frameworks, it will have to inherit from a well known public class, and so dynamic type casts can be used to access the private class as if it were its public superclass.

Garbage collection (or at least a restricted form of explicit memory management) is required to support all forms of aliasing control. If a program can delete an object while references to it are retained, and that object's memory is then reallocated to a new object, the new object will be aliased by the retained pointers to the nominally deleted object.

In practice, then, careful programming and eternal vigilance are the only defences against aliasing problems in current object oriented languages.

3.2 Full Alias Encapsulation

In recent years there have been a number of proposals to address aliasing in object oriented languages. For example, expressions can be analysed statically to determine their *effects*, described in terms of the memory regions they can change or depend upon [35, 28], whole programs can be analysed directly to detect possible aliasing [26, 10, 22], or hints may be given to the compiler as to probable aliasing invariants [18]. Objects can be referred to by tracing *paths* through programs, rather than by direct pointers, so that aliased objects will always have

the same name [3, 6, 5], or pointers can be restricted to point to a particular set of objects [38]. Copying, swapping, destructive reads, or destructive assignments can replace regular reference assignment in programs, so that each object is only referred to by one *unique* or *linear* pointer [4, 8, 32, 17, 27]. Finally, languages can provide an explicit notion of aggregation, object containment, or ownership [19, 2, 9, 24, 11, 15]. Unfortunately, these proposals forbid many common uses of aliasing in object oriented programs.

In this section, we review two of the most powerful proposals: John Hogg's Islands [19] and Paulo Sergio Almeida's Balloons [2]. Although they differ greatly in detail and mechanism — Islands use aliasing mode annotations attached solely to object's interfaces, while Balloons use sophisticated abstract interpretation — both these proposals have a common aim, which we term *full alias encapsulation*. Essentially, these proposals statically prevent external references into an object's shadow. This restriction ensures that Islands and Balloons can never suffer from problems caused by aliasing — their representations cannot be exposed, they cannot accept aliased objects from outside, and they cannot depend transitively upon other aliased objects. These restrictions apply only at the interface between Islands and Balloons and the rest of the system, so objects may be aliased freely inside or outside a Balloon or Island. Similarly, aliasing of normal objects is unrestricted within Islands and Balloons. This allows Islands and Balloons to encapsulate complex linked structures while still providing aliasing guarantees to the rest of the system.

Unfortunately, full encapsulation of aliasing is too restrictive for many common design idioms used in object oriented programming. In particular, an object cannot be a member of two collections simultaneously if either collection is fully encapsulated against aliases. A collection's member is part of the collection's shadow, and as such cannot be part of another fully encapsulated collection.

Islands and Balloons have mechanisms which mitigate against this restriction, generally by distinguishing between static and dynamic aliases — a static alias is an alias caused by reference from a long-lived variable (an object's instance variable or a global variable) while a dynamic alias is caused by a short-lived, stack allocated variable. Unfortunately, these distinctions also cause problems. Both Islands and Transparent Balloons allow dynamic aliases to any member of an aggregate object's shadow. This allows collection elements to be acted upon when they are within the collection, provided no static references are created. Unfortunately, this also allows objects which are part of an aggregate's private internal representation to be exposed.

Islands restrict dynamic aliases to be read only, that is, Islands enforce *encapsulation* but not *information hiding*. Transparent Balloons impose no such restriction, so in a transparent Balloon, an internal representation object can be dynamically exposed and modified externally. Almeida also describes Opaque Balloons which forbid any dynamic aliases. That is, transparent balloons control static aliasing, but provide neither information hiding nor encapsulation, while opaque balloons completely hide and encapsulate everything they contain.

4 Flexible alias protection

Although aliasing has the potential to cause a great many problems in object oriented programs, it is demonstrably the case that these problems do not manifest themselves in the vast majority of programs. That is, although paradigmatic object oriented programming style uses aliasing, it does so in ways which are benign in the majority of cases.

This situation parallels that of programming in untyped languages such as BCPL or assembler. Although untyped languages leave a wide field open for gratuitous type errors, programmers can (and generally do) successfully avoid type errors, in effect imposing a type discipline upon the language. Of course, it is almost certain that type problems will arise over time, especially as programs are maintained by programmers unaware of the uses and constraints of the types in the program. As a result, more formal static typing mechanisms have evolved to protect the programmer against type errors. The success and acceptance of a type system in practice depends on the extent to which it supports or constrains idiomatic programming style [25].

Our aim is to use techniques similar to type checking to provide guarantees about programs' aliasing properties, but without compromising typical object oriented programming styles. In particular, we aim to support many benign uses of aliasing, including objects being contained within multiple collections simultaneously, while still providing significant protection against aliasing problems. This requires that some form of aliasing be permitted, but that aliasing must be restricted to where it is appropriate.

Some objects can always be aliased freely without affecting the program's semantics. These objects are instances of *value types* which represent primitive values, such as machine level integers, characters or booleans. Since instances of value types are *immutable* (they never change, although variables holding them can change) they cause no problems when they are shared between various aggregate objects[2]. Functional languages have always used aliasing to implement immutable referentially transparent values — the great advantage being that precisely because these objects are immutable, any aliasing is completely invisible to the programmer.

The observation that value types can be aliased indiscriminately without compromising safety, because their state does not change, suggests an alternative formulation of the aliasing problem: the problem is not the presence of aliases, but the visibility of non-local changes caused by aliases. This suggests a different approach to dealing with aliasing: rather than trying to restrict aliases by constraining the references between objects, we should restrict the visibility of changes to objects. Aliasing can certainly be permitted, provided any changes within aliased objects are invisible.

This bears out the experience that many object oriented programs have been written in spite of aliasing — aliasing *per se* causes no problems for object

[2] Almeida describes how value types can be implemented as a specialisation of Balloons, and Hogg mentions immutable objects in passing.

oriented programming: the problem is the unexpected changes caused by aliasing. Object oriented programs which employ aliasing must do so in ways which avoid critical dependencies on mutable properties of objects.

To address these aliasing issues and to develop a programming discipline that may help preventing the problems described in previous section, we introduce the notion of an *alias-protected container* as a particular kind of aggregate object which is safe from the undesirable effects of aliasing. The remainder of this section is devoted to specifying the characteristics of alias-protected containers. The following section introduces *alias mode checking* which provides the means to enforce alias protection in object oriented languages by using aliasing modes and roles.

4.1 Alias-Protected Containers

We propose to protect containers from aliasing by dividing the elements of a container's shadow into two categories — the container's private *representation* and the container's public *arguments*. A container's representation objects are private and should not be accessible from outside. A container may freely operate upon (or depend upon) its representation objects — it may create new representation objects, change their state, and so on, but never expose them.

A container's arguments can be publicly accessed from elsewhere — in particular, an object can be an argument of more than one container. Because these objects are available and modifiable outside the container, the container may only depend upon argument objects inasmuch as they are immutable, that is, a container can never depend upon any argument object's mutable state. It is important to realise that the dependency is on the *interface* presented by the element objects to the collection. Provided all the operations in this interface do not rely upon mutable state, no changes in the element object can be visible to collection, and the element objects can be freely aliased and mutated outside the collection. This restriction protects the container's integrity against changes in elements caused via aliases.

For example, a hash table typically depends on element objects understanding a message which returns their hash code. If an element's hash code changes (presumably caused by another part of the program modifying the element via an alias) the integrity of the hash table will be compromised, but if the hash codes never change, the hash table will function correctly, even if other aspects of the elements change frequently.

Because a container's argument and representation objects have different aliasing and mutability restrictions — representation objects must remain inside the container, but can be read and written, while argument objects must be treated as immutable but can be referenced from outside the container — the implementation of the container needs to keep the two sets completely separate. If a representation object is accidently treated as an argument, it can be exposed outside the container, typically by being explicitly returned as the result of a method. If an argument object is treated as part of the representation, the

containers implementation can become susceptible to problems caused by pre-existing aliases to the argument. Figure 3 illustrates how the objects referred to by a hash table (from Figure 2) are either part of the hash table's representation or arguments.

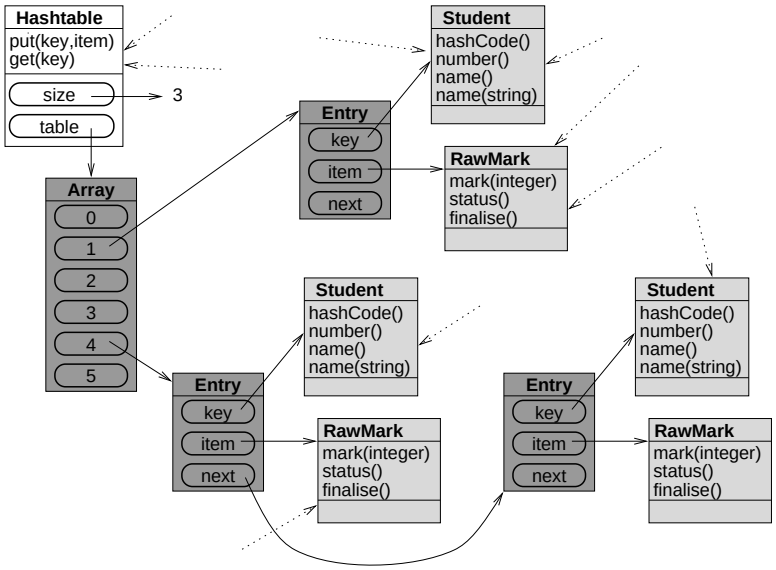

Fig. 3. A hashtable's internal Array and Entry objects are part of its representation (dark grey) while student and RawMark objects are stored as the hashtable's arguments (light grey). Representation objects can only be referenced from within the hashtable aggregate (solid arrows) while arguments objects can be referenced from outside (dotted arrows).

Alias protected containers themselves may be aliased, in fact they may even be mutually aliased. For instance, a container may be passed as argument to itself. This does not cause problems, however, as a container cannot depend upon the mutable state of its argument objects.

One very important aspect for the usability of any definition of alias-protected containers is their composability — that is whether alias-protected containers can be implemented out of simpler containers.

4.2 Composing Containers

Complex aggregate objects should be able to be composed from simpler objects — that is, containers need to be able to use other objects as part of their implementations. This is easily accommodated within our container model — a container can certainly have another object (which could be a container) as part

of its representation. Provided the internal object does not expose its own representation or depend upon its arguments, the composite container will provide the same level of aliasing protection as an individual container object.

Sometimes, however, a composite container may need to use an internal container to store some of its argument or representation objects. For example, a university student records system may need to record the students enrolled in each course and the raw marks each student has received. Each course object can use a hash table to keep track of its students and their marks, however students will be part of the course's arguments (since a single student could be enrolled in multiple courses) while each student's raw marks will be part of the course's representation, since only weighted final marks should actually be presented to students. As far as the internal hash table is concerned, both the student objects and mark objects are its arguments — the students being the hash table's keys and the raw marks the hash table's items. The hash table will also have its own representation objects, which must be completely encapsulated inside it.

To maintain flexible alias protection, a container's representation objects and argument objects must be kept completely separate. This requirement holds no matter how a container is implemented. When a container uses an internal collection, this requirement must be enforced on the internal collection also. If a representation object could be passed into the internal collection, then retrieved and treated as if it were an argument, then the composite container's representation could be exposed. Similarly, if an argument could be retrieved from an inner collection and treated as part of the composite container's representation, the composite container would become susceptible to its arguments aliasing.

To avoid breaching encapsulation, composite containers have to be restricted in how they can pass objects to internal objects. We consider that each object has one or more *argument roles*, which describe how the object uses its arguments. An object must keep its various argument roles separated — in particular, it may only return an argument as a particular role from some message if the argument was passed into the object as the same role. For example, a simple collection, such as a set, bag, or list, will have only one argument role, while an indexed collection, such as a hash table mapping keys to items, will have two roles, one for its keys and one for its items. A composite container may only store one kind of object (argument or representation) in any given inner object's role. Thus, an enclosing container can store part of its representation in an inner container and retrieve it again, sure that the inner container has not substituted an argument object or an object which is part of the inner container's representation.

Reconsidering the university course example, the hash table will have a key role and an item role. The course object stores its argument Student objects in the hash table's key role, and its representation objects representing the students' marks in the hash table's item role (see Figure 4).

4.3 Summary

We have introduced flexible alias protection to provide a model of aliasing which supports typical object oriented programming styles involving aggregate con-

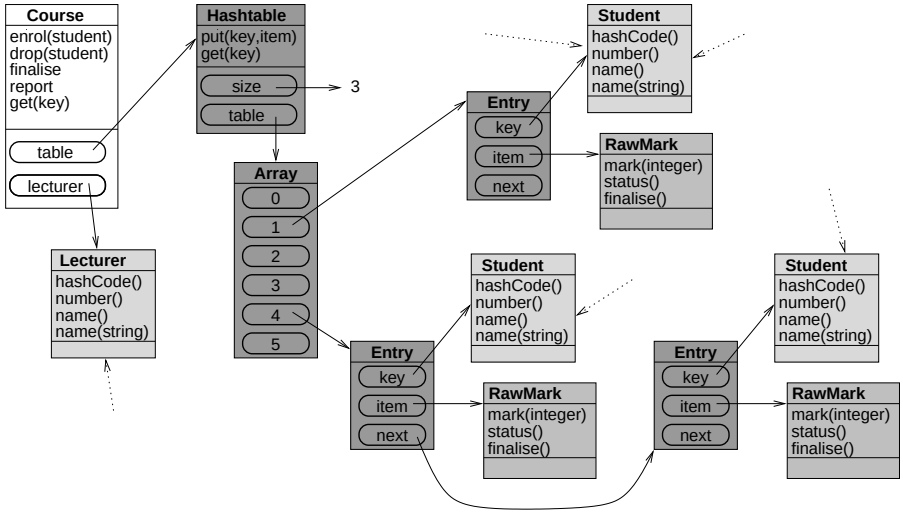

Fig. 4. A Course uses a hashtable as part of its representation (dark grey) while Student and Lecturer objects are the course's arguments (light grey). The hashtable also stores RawMark objects for each student, and these are arguments to the hashtable but part of the Course's representation (mid gray), so cannot be accessed from outside the Course (dotted arrows).

tainer objects. Flexible alias encapsulation separates the objects within an aggregate container into two categories — representation objects which can be modified within the container but not exported from it, and argument objects which can be exported from the container but which the container must treat as immutable. Argument objects can be further divided into subcategories, each representing a different argument *role*. Just as a container's representation objects must be kept separate from its argument objects, so each role must be kept independent.

These restrictions can be expressed in the following invariants:

- F_1 **No Representation Exposure** — A container's mutable representation objects should only be accessible via the container object's interface. No dynamic or static references to representation objects should exist outside the container.
- F_2 **No Argument Dependence** — A container should not depend upon its arguments' mutable state. That is, a container should use arguments only insofar as they are immutable.
- F_3 **No Role Confusion** — A container should not return an object in one role when it was passed into the container in another role.

In the next section we describe how *aliasing modes* ensure these three invariants can be checked statically for a variety of container types.

5 Aliasing Modes

We have developed a set of *aliasing modes* and a simple technique, *aliasing mode checking*, to statically ensure invariants $F_1 \ldots F_3$ hold, so that a container can defend itself against possible aliasing problems. Aliasing mode checking aims to preserve as much as possible of the paradigmatic object-oriented style, including the benign use of aliasing, while making program's aliasing properties explicit. Aliasing mode checking is based on declarations of *aliasing modes*, which are similar to the modes used in Islands [19] or the `const` mode used in C++ [36]. An aliasing mode is essentially a tag which annotates the definition of a local name, and restricts the operations which can be performed upon objects through that name. Modes are purely static entities, having no runtime representation. Like the C++ `const` attribute, and unlike Island's modes, our aliasing modes can decorate every type constructor in a type expression, and are propagated through the expressions in the program, just as types are. Also like C++ or Islands' modes, our modes are relational in that they restrict access only through the name they annotate — if an object is aliased by another name, the aliases may have different modes and allow different operations to be performed on the object. Unlike C++'s `const`, modes may not be cast away.

Aliasing mode checking verifies an object's aliasing properties to a similar extent that a static type checker verifies an object's typing properties. Working from declarations supplied by the programmer, an aliasing mode checker propagates aliasing modes through expressions. The resulting modes are then checked for consistency within the defining context. Like most type checking, aliasing mode checking is conservative, so it should not accept programs which do not have the required aliasing properties, but it may reject programs which actually have the required properties if it cannot verify them statically. Also like type checking, aliasing mode checking is enforced by a set of simple, local rules, designed to be easy for programmers to understand and to debug. Note that although they are similar, aliasing mode checking and type checking are completely orthogonal. An expression's aliasing mode correctness implies nothing about its type correctness, and vice versa.

The aliasing mode system comprises the following modes: *arg*, *rep*, *free*, *var*, and *val*. These modes decorate the type constructors in a language's type expressions, resulting in *moded type expressions*. The *arg* and *var* modes optionally also have a *role tag* $\mathcal{R}$, which is used to distinguish between similar modes which play different roles. The first two modes, *rep* and *arg*, are the most important, and identify expressions referring to the representation and argument objects of containers. The *free* mode is used to handle object creation, the *val* mode is syntactic sugar for value types, and the *var* mode provides a loophole for auxiliary objects which provide weaker aliasing guarantees. To separate argument objects from representation objects and argument objects in other roles, different modes are not assignment compatible, except that expressions of *free* mode can be assigned to variables of any other mode (assuming type compatibility).

The modes attached to the parameters of the messages in an object's interface (including the receiver, self) determine the aliasing properties for the object as

a whole. For example, if an object uses only modes *arg*, *free*, and *val*, it will be a "*clean*" immutable object, that is, it will implement a referentially transparent value type. If all of an object's method's parameters and return values (except the implicit self parameter) are restricted to *arg*, *free*, or *val*, the object will be an alias-protected container with flexible alias protection, and if in addition it has no variables of *arg* mode, the object will provide full alias encapsulation. The modes of an object's internal variables are used to check that the aliasing properties of the object's implementation match those of the declarations, as follows:

rep A *rep* expression refers to an object which is part of another object's representation. Objects referred to by *rep* expressions can change and be changed, can be stored and retrieved from internal containers, but can never be exported from the object to which they belong.

arg $\mathcal{R}$ A *arg* expression refers to an object which is an argument of an aggregate object. Objects referred to by *arg* expressions can never change in a way which is visible via that expression — that is, *arg* expressions only provide access to the immutable interface of the objects to which they refer. There are no restrictions upon the transfer or use of *arg* expressions around a program.

free A *free* expression holds the only reference to an object in the system, so objects referred to by a *free* expression cannot be aliased. In particular, the mode of the return values of constructors is *free*. Expressions of mode *free* can be assigned to variables of any other mode, provided that any given *free* expression is always assigned to variables of the same other mode.

val A *val* expression refers to an instance of a value type. The *val* mode has the same semantics as the *arg* mode, however, we have introduced a separate *val* mode so that explicit *arg* roles are not required for value types. The *val* mode is the only mode which implies a constraint upon the *type* of the expression to which it is bound, and can be automatically attached to all expressions of value types where no other mode is supplied.

var $\mathcal{R}$ The *var* mode refers to a mutable object which may be aliased. Expressions with mode *var* may be changed freely, may change asynchronously, and can be passed into or returned from messages sent to objects. This mode is basically the same as the reference semantics of most object oriented programming languages, or the *var* mode in Pascal, except that it obeys the assignment (in)compatibility rules of the other modes.

Note that modes and roles are not specific to a particular object oriented language, but do require a strong static type system. Our examples use an idealisation of Java with parametric polymorphism, based on Pizza [34], For pedagogical reasons, in places we use more explicit role annotations on variables and parameters than is strictly necessary.

5.1 Modes and Invariants

The purpose of aliasing modes is to enforce the flexible alias encapsulation invariants F_1 to F_3. The invariants are enforced by recasting them in terms of the

modes of expressions, rather than sets of objects, and then restricting the operations permissible on expressions of various modes. This results in three mode invariants, each corresponding to one flexible alias encapsulation invariant. The semantics of these invariants are implicit in the semantics of the modes described above, but we will consider each separately, as follows:

- M_1 **No Representation Exposure** — No expression containing mode *rep* may appear in an object's interface. An aggregate object's representation should remain encapsulated within that object. In the mode system, component objects which make up an object's representation will have mode *rep*, so they should not be returned from that object. Expressions including mode *rep* should not be accepted as arguments, due to the possibility of preexisting aliases. We take an object's interface to include all external variables or functions visible within an object, so this restriction (together with the composition rules below, §5.4) also stops objects exposing their representation through a "back door".
- M_2 **No Argument Dependence** — No expression of mode *arg* may be sent a message which visibly changes or depends upon any mutable state. Objects referred to by expressions of mode *arg* may be freely aliased throughout the system, so containers may not depend upon their mutable state. To enforce this restriction, we forbid messages sent to *arg* expressions which access any mutable state. The only messages which may be sent to *arg* expressions are those which are purely functional — we call them *clean* expressions. For the same reason, *arg* expressions may only be passed to other functions as mode *arg*.
- M_3 **No Role Confusion** — No expression of any mode except *free* may be assigned to a variable of any other mode. Objects subject to mode checking must keep objects of different roles separate. This can be implemented fairly simply by forbidding assignment between expression of different modes.

Immutable Interfaces Rule M_2 for avoiding argument dependencies requires that messages sent to expressions of mode *arg* should not depend upon mutable state, or cause any side effects. We call these types of messages *clean* messages, and they should be identified by an annotation on method declarations. One simple definition of a *clean* message is that it is made up only of *clean* expressions, where a *clean* expression either reads a variable of mode *arg* or *val*, or sends a *clean* message — the only modes which may appear in a *clean* method definition are *arg*, *val*, or *free*, and a clean method cannot modify variables. More complex definitions of *clean* could be formulated to have the same effect, but with fewer practical restrictions.

However it is defined, *clean* will impose restrictions on the way a container can use its arguments, but these restrictions are not as severe as they may seem. This is because aliasing mode checking distinguishes between *clean interfaces* and *clean objects*. A *clean* interface provides access to the immutable properties of an otherwise mutable object, while a *clean* object implements an immutable value type. A mode *arg* reference to a mutable object restricts the use of that

object to the *clean* portion of its interface — if the object is aliased elsewhere via *rep* or *var*mode, those references can make full use of the object. Completely *clean* objects are only required when value semantics (mode *val* expressions) are to be used, and should be identified by annotations on objects' definitions.

5.2 Example: A Simple Hashtable

To illustrate the use of modes and roles Figure 5 shows a simple example of a naïve hashtable class completely annotated with mode declarations — compare with Figure 2. The hashtable is represented using an array of hashtable entries which hold the keys and items stored in the table.

This example uses three modes — *arg*, *val*, and *rep*. Argument items to be contained within the hashtable are declared as mode "*arg k*" or mode "*arg i*" — that is, mode *arg* with role tag *k* for keys or *i* for items. The modes are identified both in the declarations of method parameters and return values, and within the definition of the `table` representation array. The representation array object holding the hashtable entries is mode *rep*, because it needs to be changed by the hashtable (to store and retrieve entries). The entries themselves are similarly mode *rep*. Because the table contains argument objects (which are mode *arg*), the table's full moded type is **rep** `Array<HashtableEntry<`**arg k** `Hashable,` **arg i** `Item>>`. Finally, integers are used to return `size` of the hashtable. Since integers are value types, these are mode *val*.

```
class Hashtable<arg k Hashable, arg i Item> {
   private rep
      Array<rep HashtableEntry<arg k Hashable, arg i Item>>
         contents;
   private val int size;

   public void put(arg k Hashable key, arg i Item value);
   public arg i Item get(arg k Hashable key);
}
```

Fig. 5. A Hashtable with Aliasing Mode Declarations

5.3 Mode Checking

In this section we give an intuitive overview of mode checking, as a formal definition is beyond the scope of this paper. A method is mode checked by first determining the modes of its constituent expression's terms, then propagating modes through expressions. Determining the mode of the terms in an expression is generally quite simple — aliasing modes are attached to terms in the environment. Propagating modes through compound subexpressions is more complicated, but

assuming moded type information for operators is available in the environment, parameter modes can be checked against the environment definitions, and then the operator's result mode from the environment can be taken as the mode of the whole subexpression.

Figure 6 shows the definition of the hashtable's `get` method. The figure includes an *arg k* mode declaration on the method's `key` parameter, and is annotated with the modes of the most crucial terms in the method.

```
arg i Item get(arg k Hashable key ) {
   val int hash = key.hashCode();
   val int index =
      (hash & 0x7FFFFFFF) % contents.length;
   rep HashtableEntry<arg k Hashable, arg i Item> e;
   for (e = contents[index]; e != null ; e = e.next) {
            // all rep HashtableEntry<arg k Hashable, arg i Item>
      if ((e.key.hashCode() == hash) && e.key.equals(key)) {
            // hashCode and equals must be clean
         return e.item;
      //...
```

Fig. 6. The Hashtable Get Method

The most important message send in the `get` method occurs on the first line, in `int hash = key.hashCode()`. The mode of `key` is *arg*, and only *clean* messages may be sent to expressions of mode *arg*. Provided the `hashCode` method is *clean*, it may be sent to the `key` parameter object. Since `hashCode` returns an integer, its return value has mode *val* and so can be assigned to a mode *val* variable. The arithmetic on the second line is simple: arithmetic operators on mode *val* expressions return mode *val*.

The expressions within the `for` loop are more complex to check, as these involve a number of propagations. First, the mode of `contents[index]` must be determined. This is a index operation on the `contents` array. Since the array has the mode **rep** `Array<`**rep** `HashtableEntry<`**arg h** `Hashable,` **arg i** `Item>>`, the result of an index operation will be the mode of the hash table entries — **rep** `HashtableEntry<`**arg h** `Hashable,` **arg i** `Item>`. This is the same as the mode of the `e` variable, so the assignment can proceed. Since this is a *rep* mode, the fields of `e` (and other `HashtableEntries` which also have mode *rep*) can be read and assigned to. The `HashtableEntry` objects have two fields, `key` and `item` with modes *arg k* and *arg i* respectively. Since *arg k* objects support *clean* `hashCode` and `equals` methods, these two sends can proceed even though they are sent to the mode `arg` expression `e.key`. Finally, `e.item` can be returned since it has mode *arg i*.

5.4 Composing Moded Types

Mode checking in the context of a single method is quite simple, and is adequately covered by the rules and invariants described above. In fact, the above rules apply within a single static type environment, such as a module or package, even if this involves more than one class. For example, the hash table above actually involves a number of objects (the hash table itself, its component array, and the hash table entries) but does so solely from the perspective of the hash table.

Mode checking which crosses scope boundaries is somewhat more complex. When an aggregate object is composed from a number of other objects, the modes in the aggregate object must be unified with the externally visible modes of its subsidiary objects — that is, the subsidiary objects' *arg* and *var* roles. We call this process *aliasing mode parameter binding*, and it is analogous to the binding of type parameters when instantiating generic types. The complexity arises when containers are composed inside other objects, as any mode must be able to be bound to mode *arg* moded type parameters of encapsulated containers.

For example, imagine the hash table being used to represent the relationship between `Student` objects and `RawMark` objects representing the enrolment in a university course (see Figure 7). The `Student` objects have mode *arg*, because they do not belong to the `Course` object — in particular, one student can be enrolled in a number of courses. The `RawMark` objects are part of the `Course` object's representation (i.e. mode *rep*), to ensure that they are encapsulated within the `Course`, and also so that they can be sent messages which change their state to record each `Student`'s raw marks. These *rep* `RawMark` objects need to be stored within the `Hashtable`, that is, passed to variables and retrieved as results which were declared as mode *arg* (see Figure 5).

```
class  Course<arg s Student> {
   private rep Hashtable<arg s Student, rep RawMark> marks =
         new Hashtable();
      ...
   public void enrol (arg s Student s) {
      rep RawMark r = new RawMark();
      marks.put(s, r);
   }
   public void
      recordMarkFor(arg s Student s, val String workUnit, val int mark) {
      marks.get(s).recordMarkFor(workUnit, mark);
   }
   public void finalReport (arg s Student s) {
      marks.get(s).finalReport();
   }
}
```

Fig. 7. A Course represented by a Hashtable

Aliasing mode parameter binding occurs when generic types are instantiated and their parameters bound. A sequence of actual moded types (containing the client's roles) must be bound to a sequence of formal moded types (containing the server's roles), resulting in a mapping from formal to actual moded type parameters. Each moded type parameter is considered individually, in two stages. First, the parameter roles are bound, and then the aliasing modes in the bindings are checked.

Each formal role in the server must map to one actual role in the client. One actual role may be mapped by more than one formal role, however. The most important feature of these mappings is that they must be consistent, that is there must be only one mapping for each object within any given scope, and whenever parameters are passed to or results retrieved from a particular object, the *same* mappings must be used.

The aliasing mode bindings in these mappings are then checked depending upon the modes within the server's formal parameters. Formal parameters may have either mode *arg* or mode *var* (since mode *rep* is encapsulated within components, and modes *free* and *val* are global so do not need to be bound). Formal mode parameters of mode *arg* can be bound to any actual mode, and formal parameters of mode *var* can be bound to any actual mode except *arg*.

These binding rules are designed to ensure that the flexible alias encapsulation invariants of an outer container are maintained, assuming they are maintained by an inner container. The interesting cases occur when objects which are parts of an outer container are passed to an inner encapsulated container, since the basic rules encapsulate aliasing in each individual container.

The outer container's representation is protected against exposure (F_1 is maintained) because the inner container can only return the outer container's representation objects back to the outer container, as the inner container is part of the outer container's representation. An inner container cannot depend upon any of the outer container's arguments, because the outer container's arguments can only be bound to mode *arg* in an inner container. Thus F_2 for the whole container is supported by M_2 in the inner container.

Similarly, the role binding rules and M_3 in an inner container ensure that the enclosing container's roles are not confused, maintaining F_3. Each formal role in the inner container can only be bound to at most one of the outer container's roles, so objects cannot be inserted into the inner container under one outer container role and retrieved as another. Several inner container roles may be bound to one outer container role, but this simply means the inner container makes a finer distinction within the outer container's roles.

5.5 Choice of Modes

Our choice of aliasing modes may seem somewhat idiosyncratic. While some of the modes (*var*, *rep*, *val*) are hopefully noncontroversial, and others taken directly from previous work (*free* from Islands [19]), the *arg* mode is novel. We have also omitted several modes from other work, including *read* and *unique* modes [19, 2]. This section presents some of the rationale for our choice of modes.

Mode *val* The *val* mode is in a strict sense redundant, as its semantics are essentially the same as *arg* mode, and could be replaced by *arg* mode without weakening the system. We have retained *val* mode for a number of reasons, foremost of which is that we share a sense of the overall importance of value types [2, 29, 23].

A separate *val* mode provides an additional cue to the programmer when used to describe a component of a container's representation. Reading the moded type declaration **rep** `Foo<`**arg a** `Shape>` a programmer can conclude that the *arg a* components of `Foo` are "real" arguments which may be aliased elsewhere. In contrast, the similar declaration **rep** `Foo<`**val** `Shape>` makes clear that the `Shape` components are pure value types. More practically, an explicit *val* mode greatly reduces the number of roles programmers must consider when designing objects, because all expressions which handle *clean* objects can have mode *val*.

The Restrictions on Mode *arg* The restrictions we have placed on mode *arg* are particularly tight. Mode *arg* combines the restrictions of modes like C++'s `const`, which prevents modifications to objects, and a strong transitive sense of referential transparency, so no changes are visible through *arg* mode references. The second part of this restriction is certainly necessary to guarantee the flexible aliasing encapsulation invariants, in particular F_2. The first part of this restriction is less necessary, because the aliasing invariants implicitly assume that a container's arguments may be changed asynchronously via aliases at any time, so no mode safety would be lost by allowing changes via an *arg* reference (at least in sequential systems). We have imposed this restriction as a matter of taste, to keep the mode system as simple as possible, and because widespread use of a `writeonly` mode seems quite counterintuitive [37].

***read* and *unique* Modes** Islands [19] make great use of a *read* mode, which can be seen as a transitive version of C++'s `const`. These *read* mode expressions cannot be used to change mutable state, and cannot be stored in object's instance variables, but are not referentially transparent, so the objects upon which they depend may be changed "underfoot" via aliases.

We have omitted a *read* mode for three reasons. First, *read* expressions are used to dynamically expose objects which are part of Islands. Since argument objects within flexibly encapsulated containers can be statically or dynamically aliased outside, a *read* mode is much less necessary. Second, to be useful, a *read* mode must constrain Islands' *clients*, and we have tried to avoid modes which propagate upwards, out of containers into their context. Third, especially because of the restrictions on storing *read* expressions into objects' variables, *read* does not fit well with typical object oriented programming styles.

Islands also introduced a *unique* mode [19], and similar ideas are used in Balloons [2] and have been proposed by others [32, 8, 17]. A *unique* variable is *linear* — it holds the only reference to an object [4]. We have not introduced a *unique* mode for much the same reasons we have omitted *read*. Like a *read* mode, a *unique* mode is useful in some cases, for example, a *unique* mode allows

objects to be inserted to and removed from encapsulated containers without copying or aliasing. Like a *read* mode, a *unique* mode extends its protection "upwards", requiring respect from containers' clients. Also, a *unique* mode may not provide as much protection as might be imagined, since *unique* objects can be shared via non-*unique* "handle" objects [32]. Unfortunately, making effective use of *unique* objects seems to require programming language support, such as a destructive read [19], copy assignment [2], or swapping [17]. Finally, our *free* mode reduces the need for a *unique* mode, although at the cost of requiring extra object copying in some circumstances.

Upwards and Downwards Mode Restrictions Most of our aliasing modes are anchored at a particular object, and propagate *downwards* into the implementation of that object, restricting the ways it can use other objects. This is in contrast to modes like *read* and *unique* which work *upwards*, giving rise to restrictions on objects' clients, Mode *arg* is a downward mode *par excellence* — *arg* imposes a great many restrictions on a container's implementations, but none on a container's client. We prefer downward modes to upward modes for several reasons. We assume objects with flexible alias encapsulation will form part of a traditional, alias-intensive object oriented system, and we aim to support a paradigmatic object oriented programming style, so we cannot make assumptions about programs' global aliasing behaviour. We don't want programmers to have to rewrite code to conform to mode restrictions. We imagine aliasing modes infiltrating the systems bottom up — our flexible alias encapsulation is particularly suitable for describing properties of existing collection libraries, for example. Containers with flexible alias encapsulation must defend *themselves* against aliasing problems: they cannot rely on the rest of the program "doing it for them" by obeying mode restrictions.

The only assumption we do accept about the "rest of the program" is that any methods or expressions claiming to be *clean* or *free* are in fact *clean* or *free*. In a way, this constraint also flows downwards, from the interface of the external objects to their implementation, rather than upwards out of a container to its elements. We view *clean* and *free* as *descriptions* of the properties of external objects in the program, which restricts the operations which flexible alias encapsulated containers can do with those external objects, rather than restrictions on the external objects.

5.6 Object Oriented Idioms

We conclude the presentation of aliasing modes by showing how they can be used to capture the aliasing properties of a number of common object oriented programming idioms.

Flyweights as *clean* objects Flyweight objects [12] contain no mutable intrinsic state, that is, a Flyweight object is an instance of a value type. A Flyweight can be described using the mode system as a *clean* object, that is, an object

which provides only a *clean* interface. A *clean* object is restricted to expressions of modes *arg*, *free*, and *val* — in particular, the mode of self is *arg*, which prevents assignment to any instance variables. For example, a simple Glyph flyweight could be implemented as a clean object:

```
clean class Glyph {
   private val Font font;
   private val int size;
   public free Glyph( val Font _font, val int _size) {
      font = _font; size = _size;
      //...
```

Although *clean* objects cannot normally access mutable state, they must still be constructed and initialised. Aliasing modes model construction explicitly, by treating constructors as special methods which return mode *free*. This allows objects' instance variables to be initialised within constructors, because *free* does not have the *clean*-message only restriction of mode *arg*. A *clean* object's variables could even be initialised after construction, for example to cache the results of a *clean* method, modelling language constructs such as Java's blank finals [14] or Cecil's per-object field initialisers [7].

Collections and Facades with Full Alias Encapsulation Collections and Facades are usually modelled as containers with flexible alias protection, that is, as objects where only *arg*, *val*, and *free* modes may appear in their method interfaces (including constructors) and any variables with scope larger than an object may only be read as mode *arg*. Aliasing modes can enforce the kind of full alias encapsulation provided by Islands [19] or Balloons [2]. In addition to the restrictions for flexible alias protection, instance variables of fully encapsulated containers may not have mode *arg* subcomponents. This allows aliased objects to be passed into a container, but not stored directly within it — to store an object it must first be copied, producing a *free* object which can then be passed to a mode *free* parameter and assigned to a mode *rep* variable. For example, a fully encapsulated `TupperwareSet` could be implemented using a `Set` with flexible alias protection as follows:

```
class TupperwareSet {
   private
      rep Set<rep Object> storage; // no arg subcomponents.
   public
      void add(free Object _o) {
         rep Object o; // for clarity
         o = _o.; // assign free copy to rep
         storage.add(o);
      }
```

If the `_o` argument was mode *arg* rather than mode *free*, it could not be added to the *rep* `storage` set.

Iterators for Collections Iterators [12] are commonly used to provide sequential access to collections. Unfortunately, by their very nature, iterators must alias the collections they iterate over, indeed, iterators often need direct access to containers' private implementations for efficiency reasons. This aliasing is made explicit in the moded type declarations, where an extra *var* role is required to indicate that implementations and iterators are aliased.

```
class FastVector<var a Array<arg i Item>> {
   private
      var a Array<arg i Item> table; // note var
   public
      free FastVectorIterator<var a Array<arg i Item >>
         newIterator() {
            FastVectorIterator(this.table);
               // hand in internal table
         }
   }

class FastVectorIterator<var a Array<arg i Item>> {
   public
      free FastVectorIterator<var a Array<arg i Item>>
         FastVectorIterator(var a Array<arg i Item >>) {
         // direct access to Array implementation
         // via var  parameter inside constructor ...


class IteratorClient {
   private
      var FastVector<var a Array<rep Elem e>> arr;
   public
      void iterate() {
         var FastVectorIterator<var a Array<rep Elem e>> it;
         rep Elem e;
         for (it=arr.newIterator(); it.hasNext; e = it.next) {
            e.use();
            //...
```

In the example, the iterator and vector have mode *var* in their moded types, so they cannot be exported from the `IteratorClient` object, if the `IteratorClient` is to be an alias-protected container.

6 Discussion

In this section we discuss further aspects of flexible alias protection and aliasing mode checking, and describe the current status of our work.

6.1 Usability

Because object identity is such a fundamental part of the object orientation paradigm, problems with aliasing cannot really be "solved". Any attempt to address the aliasing program for practical object oriented programming must be evaluated as an engineering compromise: how much safety does it provide, at what cost, and, most importantly, how usable are the mechanisms by typical programmers doing general purpose programming.

The crucial question is how natural (or how contrived) a programming style is required by the proposed aliasing mode checking. Obviously aliasing mode declarations impose a syntactic overhead, but this at most doubles the cost of the kind of static type declarations used in Eiffel or C++, even if all type declarations must be annotated with modes. In return for the extra syntax, flexible alias encapsulation imposes significantly weaker restrictions on program design than other types of alias encapsulation [2, 19, 24], while still providing protection against common aliasing problems. In particular, flexible encapsulation allows container arguments to be aliased, permitting many programming idioms which cannot be used when aliases are fully encapsulated.

Making aliasing modes explicit has advantages when checking aliasing modes and reporting aliasing errors. Like type checking, aliasing mode checking only needs information which is in the scope of the expression to be checked. Methods can be checked individually and incrementally, and because alias modes are visible to the programmer in the program's text, errors can be reported in terms which programmers should be able to understand.

This is in contrast to the sophisticated static analysis required to check Balloon types, which may need to check the implementation of a number of different classes as a unit, and which reports errors in terms of possible runtime aliasing states, rather than syntactic properties of the program [2].

The ability to present comprehensible error messages points towards an important secondary benefit of programmer-supplied aliasing declarations. Making alias modes explicit should help provide a conceptual language within which programmers can think about the aliasing properties of their programs and designs, in the same way that type systems promote awareness of program's type properties.

6.2 Inheritance and Subtyping

Aliasing issues are generally considered to be orthogonal to subtyping and inheritance [19, 2], so alias mode checking should be orthogonal to type checking and subtyping. In practice, there can be interplay between objects' aliasing properties, subtyping, and inheritance.

Subtyping is defined by the substitution principle — that an instance of a subtype can be used wherever an instance of a supertype is acceptable [1]. Considering aliasing, substitution requires that a subtype's aliasing guarantees cannot be weaker than its supertype's. The precise rules can be derived from the aliasing mode invariants (particularly M_3), expressed in the mode binding rules

in section 5.4. A subtype's aliasing modes must be able to be bound wherever its supertype's modes can be bound. The main consequence of this rule is that a type's *clean* interface must be a subtype of its supertype's *clean* interface, for *all* types in the program.

Alias mode checking depends upon inheritance, at least, it treats objects as if inheritance had been flattened. Modes introduce dependencies which make subclasses more dependent upon details of their superclasses, exacerbating the fragile base class problem. As with typing, visibility declarations can offer subclasses some protection against changes to superclasses mode definitions, by restricting the scope of the changes. If inheritance is used for code reuse, a subclass may require different modes to its superclass, giving rise to inheritance anomalies similar to those found in concurrent systems [31].

6.3 Concurrency

Flexible alias protection and aliasing modes provide a good foundation within which object oriented languages can support concurrent execution. Flexibly encapsulated objects can be units of concurrency control, that is, a container can manage concurrent access to itself and its representation objects. The M_1 invariant guarantees that no process is able to access a *rep* object without first passing through its enclosing container, and thus being subject to the container's concurrency control regime. Containers and their *rep* objects can be internally multi-threaded (providing intra-object concurrency) and they must manage this concurrency internally.

The accessing modes map particularly well onto the *Aspects of Synchronisation* model of concurrency control [21]. This model divides concurrency constraints into three *aspects* — *exclusion* constraints which protect objects against conflicting threads, *state* constraints which allow access to an object only when is it in a particular state, and *coordination* constraints which can depend upon multiple unrelated objects. Exclusion and state constraints are local to individual objects, that is, a container and any *rep* subcomponents. Transaction constraints involve multiple independent objects, so apply to objects which use mode *var* expressions.

Finally, *clean* objects and interfaces do not require any form of concurrency control, because they do not involve mutable state. This is particularly useful in conjunction with flexible alias encapsulation, because multiple concurrent processes and multiple concurrent containers can safely store and access shared elements via mode *arg* or mode *val* without any concurrency control, because *arg* and *val* mode references only provide access to *clean* interfaces.

6.4 Mode Polymorphism and Inference

Our system of aliasing modes is more restrictive than it needs to be. This is because the programmer is forced to choose a specific mode declaration for every argument and variable, even though more than one declaration may be consistent within the context of the whole program. Unfortunately, once a mode has been

chosen for a particular method or variable, other uses of that method which would require different modes are rejected by aliasing mode checking.

What is required here is some form of *mode polymorphism* — a single definition of a method, variable, object, or interface needs to be interpreted with different generic bindings for modes, in the same way a type-generic module can be instantiated with different concrete types. In our development of aliasing mode checking to date, we have not investigated mode polymorphism deeply.

Aliasing mode *inference* could also reduce the need for programmers to be overly specific about their program's aliasing modes. By analogy with type inference, aliasing mode inference would infer possible aliasing modes by analysing the source text of the program, automatically adding mode declarations to programs without them. We have only addressed inference in as much as mode checking's propagation of modes through expressions is the basis for inference.

6.5 Immutability and Change Detection

Our aliasing mode system is also restrictive because it is based around immutable properties of objects — properties which are set when objects are created (or initialised lazily) but do not subsequently change. Some objects' otherwise "immutable" state may remain unchanged for long periods of time, but then change on rare occasions. For example, a student's name is generally immutable, but may be changed by deed poll or marriage. If names may possibly change, they cannot be part of the student objects' clean interface, so student objects cannot be sorted or indexed based upon their names. Rather, some other attribute of students must be used to access them. Most academic institutions introduce student numbers for just this purpose, of course, and these typically meet the all the requirements for being part of a clean interface. The use of aliasing modes supports the practice of assigning these kind of "account numbers" during program analysis and design [40].

Alternatively, dynamic change detection techniques could be employed to handle changes in objects which would otherwise be treated as immutable. In this approach, the programming language or runtime system is extended to detect when a container depends upon the properties of one of its arguments, that is, when a container sends a message to another object through a mode *arg* reference. When such a dependency is detected, it can be recorded by the change detection system, which can then monitor the state of the "*subject*" object which is being depended upon. Using a mechanism such as the Observer pattern, when the subject's state changes, the dependent container can be notified of the change and can update its internal state [12]. We plan to extend our previous work on dynamic change detection to incorporate flexible alias protection and aliasing modes [33].

6.6 Current Status

In this paper, we have presented a conceptual model of flexible alias protection. We have also developed formal models of flexible alias protection which are

not presented here due to space restrictions. We are currently working on an extension of the Pizza compiler [34] to extend Java with aliasing modes and mode checking — indeed, all the examples in this paper are written using our moded Pizza (mmmPizza) syntax. Because aliasing mode checking is carried out purely at compile time, mmmPizza generates exactly the same code as the original Pizza compiler. We considered building a preprocessor to implement aliasing mode checking, however we believed it would be easier to modify an existing compiler than to build a mode checker from scratch. We are also working on the implementation of alias protected class libraries which we shall use in real applications. At that point, we will be able to asses more precisely the impact of aliasing modes on programming style.

7 Conclusion

One man's constant is another man's variable.
Alan Perlis, Epigrams on Programming.

Aliasing is endemic in object oriented programming. Indeed, given that object oriented programming is based strongly on object identity, perhaps *alias oriented* programming would be a better term than object oriented programming! We have presented flexible alias encapsulation, a conceptual model for managing the effects of aliasing, based on the observation that aliasing *per se* is not the major problem — rather, the problem is the visibility of changes caused via aliases. This model uses explicit aliasing modes attached to types to provide static guarantees about the creation and use of object aliases. As a result, the model prevents exposure of object's representations, limits the dependence of containers upon their arguments, and separates different argument roles.

Acknowledgements

We would like to thank Doug Lea for his pertinent comments on various drafts, Eydun Eli Jacobsen for his observation on protecting names versus protecting objects, David Holmes for his comments on aliasing and concurrent object systems, David Clarke for his perspectives from the evolving formal theory and implementation, John Boyland for his discussions about modes and promises, and Martin Odersky for the Pizza compiler. We also thank Bjorn Freeman-Benson and the anonymous reviewers for their careful consideration. This work was supported by Microsoft Pty. Ltd., Australia.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.
3. Pierre America and Frank de Boer. A sound and complete proof system for SPOOL. Technical Report Technical Report 505, Philips Research Laboratories, 1990.

4. Henry G. Baker. 'Use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1), January 1995.
5. Edwin Blake and Steve Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *ECOOP Proceedings*, 1987.
6. Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), October 1981.
7. Craig Chambers. The Cecil language: Specification & Rationale. Technical Report Version 2.7, University of Washington, March 1997.
8. Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limitied specifications for analysis and manipulation. In *IEEE International Conference on Software Engineering (ICSE)*, 1998.
9. Franco Civello. Roles for composite objects in object-oriented analysis and design. In *OOPSLA Proceedings*, 1993.
10. Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedigns of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.
11. Jin Song Dong and Roger Duke. Exclusive control within object oriented systems. In *TOOLS Pacific 18*, 1995.
12. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
13. T. Goldstein. The gateway security model in the Java electronic commerce framework. Technical report, Sun Microsystems Laboratories – Javasoft, December 1996.
14. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
15. Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, Montreal, Quebec, May 1994.
16. Daniel Hagimont, J. Mossière, Xavier Rousset de Pina, and F. Saunier. Hidden software capabilities. In *16th International Conference on Distributed Computing System*, Hong Kong, May 1996. IEEE CS Press.
17. Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5), May 1991.
18. Laurie J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the IEEE 1992 International Conference on Programming Languages*, April 1992.
19. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.
20. John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.
21. David Holmes, James Noble, and John Potter. Aspects of synchronisation. In *TOOLS Pacific 25*, 1997.
22. Neil D. Jones and Steven Muchnick. Flow analysis and optimization of LISP-like structures. In Steven Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.
23. Stuart Kent and John Howse. Value types in Eiffel. In *TOOLS 19*, Paris, 1996.
24. Stuart Kent and Ian Maung. Encapsulation and aggregation. In *TOOLS Pacific 18*, 1995.

25. Brian Kernighan. Why Pascal is not my favourite programming language. Technical Report 100, Bell Labs, 1983.
26. William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.
27. K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. Technical Report SRC-TN-97-001, Digital Systems Research Center, April 1997.
28. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.
29. B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12), December 1982.
30. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kirsten Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
31. S. Matsuoka, K. Wakita, and A. Yonezawa. Sychronisation constraints with inheritance: What is not possible? — so what is? Technical report, Dept. of Information Science, University of Tokyo, 1990.
32. Naftaly Minsky. Towards alias-free pointers. In *ECOOP Proceedings*, July 1996.
33. James Noble and John Potter. Change detection for aggregate objects with aliasing. In *Australian Software Engineering Conference*, Sydney, Australia, 1997. IEEE Press.
34. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.
35. John C. Reynolds. Syntatic control of interference. In *5th ACM Symposium on Principles of Programming Languages*, January 1978.
36. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
37. Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
38. Mark Utting. Reasoning about aliasing. In *The Fourth Australasian Refinement Workshop*, 1995.
39. Jan Vitek, Manuel Serrano, and Dimitri Thanos. Security and communication in mobile object systems. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet.*, LNCS 1222. Springer-Verlag, April 1997.
40. William C. Wake. Account number: A pattern. In *Pattern Languages of Program Design*, volume 1. Addison-Wesley, 1995.
41. Alan Cameron Wills. *Formal Methods applied to Object-Oriented Programming*. PhD thesis, University of Manchester, 1992.