The Objects of Aliasing

James Noble
Microsoft Research Institute
Macquarie University
Sydney, Australia
kjx@mri.mq.edu.au

April 7, 1999

Abstract

Aliasing is fundamental to object-oriented systems: every object has an identity, and since by that identity an object can be referenced from anywhere within the program, then the object could be aliased from anywhere within the program. By complicating the relationship between the static program text and its dynamic execution, aliasing makes programs harder to understand, to modify, and especially to debug. In this position paper, we describe some of the objectives of studying aliasing in object-oriented programs, and attempt to outline the major options for managing aliasing.

The aim of this position paper is to initiate discussion during the workshop, rather than to present a comprehensive survey of finished work.

1 Introduction

In recent years, there have been a number of valiant attempts to address the problems caused by aliasing in object-oriented languages. In some sense, the problems caused by aliasing are endemic to object-orientation, in that aliasing problems stem directly from two of the foundations of object-orientation: object identity and mutable object state [17]. In this position paper we outline (very briefly) a number of the issues involved in managing aliases in object-oriented systems, and attempt to illustrate how the various approaches fit together.

2 The Aim of Alias Management

The *Geneva Convention on the Treatment of Object Aliasing* [13] outlined four main strategies for managing aliasing: detection, prevention, control, and advertisement. Alias prevention prevents aliasing from ever occurring in programs, detection determines when aliasing is present in programs, advertisement allows the programmer to declare when aliasing may occur, and control ensures aliasing is restricted to only certain parts of the program.

In this position paper, we use the term "alias management" to cover any and all of these strategies. This is because many alias management techniques combine several of these strategies — perhaps preventing aliases in some regions of the program, requiring the programmer to advertise when aliases may occur in other regions, and then controlling any aliases which do occur so they do not grant access to those parts of the program where aliases are forbidden.

The aim of most alias management schemes is to prevent the problems caused by aliasing — and it is important to note that preventing the problems caused by aliasing is a different matter to preventing aliasing *per se*. Generally, problems arise when an object is examined or modified via an alias, while the existence of an alias merely creates the potential for such a problem to occur.

The classic problem caused by aliasing is *representation exposure* [14] when details of the representation of an abstraction are leaked out across the abstraction's encapsulation boundary. Many alias management schemes aim to prevent representation exposure, by ensuring an abstraction's representation can never leak across its boundary. A related problem is *argument dependence* [17], when the implementation of an abstraction depends upon an aliased object which is imported across the boundary from outside. Again, many alias management schemes also address argument dependence, often by tightly restricting the import operation (as in Islands [12]), or by limiting the visibility out across the boundary (as in Flexible Alias Protection [17]).

Aliasing management schemes can also address more general problems. For example, representation exposure is a specific case of privileged information (the representation) reaching an unprivileged part of the program, that is, of a security problem. Some alias management schemes, such as Confined Types [3] address these security problems directly, while others address optimisation [9]. Finally, some schemes may aim primarily to describe the general invariants about program's data structures [11].

3 The Objects of Aliasing

Alias management schemes can deal with any language construct where aliasing can appear. We make a distinction between *object* aliasing and *name* aliasing — object aliasing means that one object can be reached by more than one path though other objects in memory, while name aliasing means that one object can be referenced by more than one name in the program's text. Name aliasing is the only kind of aliasing which can occur in languages without pointers, and techniques for managing name aliases are quite well understood, so in this paper we address only object aliasing.

3.1 Drawing Abstraction Boundaries

Considering object aliasing, then, there are several different kinds of aliases which can be managed. For example, managing aliases within a static program entity (such as a class or package, as in Confined Types [3] or Sandwich Types [9]) is generally easier than managing aliases within dynamically allocated objects. Alias management schemes can also differ about scope of an object, that is, where an object's encapsulation boundary should be drawn. Many schemes, including Islands and Balloons, provide *full alias encapsulation*, that is, an encapsulation boundary is drawn around an object and all objects reachable from that object. More flexible alias protection schemes relax this restriction, using some notion of object ownership to make a distinction between an object's *representation* which is within the boundary and its *arguments* to which it can refer but which lie outside its boundary [17, 6]. COM-style components have yet another style of encapsulation, where several objects provide interfaces to a component, and share many other components as their representation.

3.2 Static and Dynamic Aliases

Many alias management schemes make distinctions between *static* and *dynamic* aliases, that is, between aliases which originate in variables stored in the heap (static aliases, such as object's variables) and variables stored on the stack (dynamic aliases, such as method parameters and temporary variables) [13]. Generally, static aliases are considered worse than dynamic aliases, because dynamic aliases cease to exist when their stack frame is released, while static aliases may persist arbitrarily in the heap.

Some full alias protection schemes allow dynamic aliases across encapsulation boundaries to provide temporary access to objects' representations, and enforce restrictions so that the dynamic aliases cannot be stored in to objects' variables and become static aliases [12, 1]. Unfortunately, dynamic aliases can still be used to corrupt objects' representations from outside the abstraction boundary unless they are carefully managed.

4 Mutability

Aliasing problems arise when state is modified, so some aliasing management schemes address the mutability of state. One technique is to distinguish between *reference* and *value* semantics, especially for arguments passed into procedures: reference semantics means objects' identities are passed (which can create aliases), while value semantics means than the identity of a fresh copy of objects is passed (which cannot create aliases). Given that object-orientation is based around object identity, objects in most languages are always passed by reference, but some languages also support value semantics. For example, expanded types in Eiffel are always passed by value to avoid these kinds of aliases [15].

Pure functional programming languages lie at the other extreme to object oriented languages: in a functional language, every object is immutable — or alternatively, every object is passed by copying — and since pure functional languages do not provide direct tests for object identity, programs cannot distinguish between copies or shared immutable objects.

Several alias protection schemes employ "functional sublanguages" which enforce restrictions on the mutability or access to mutable state. Often this appears as a transitive "readonly" or "const" mode (as in Islands [12]) which permit objects to read state but not to change it. Reading state does allow information to flow out of abstractions, however, and so in some cases a "writeonly" mode could be appropriate — for example, allowing information to flow into an abstraction while preventing the abstractions representation against being exposed. Flexible Alias Protection uses an even more strict functional sublanguage which forbids both reading and writing mutable state, only allowing immutable state (such as constant or write-once variables) to be read [17]

5 Uniqueness

A number of alias-management schemes employ some notion of uniqueness — that some object is accessible through precisely one "unique" pointer from another object, or perhaps on the stack. The great advantage of a unique object it that it can be created, modified, and passed quite freely around the program — its ownership or access paths can change dynamically — and, provided its uniqueness conditions are maintained, no aliases are introduced.

Given a choice to include uniqueness, there are a number of different programming language mechanisms which can be used to support programming with unique variables or objects. Most proposed schemes involving uniqueness choose one of the following approaches: the first three (destructive reads, copy assignment, swapping) seem roughly equal in power, with destructive reads being the most popular and probably the most efficient. Strict and non-strict linearity are rather more restrictive conditions.

As with other aspects of alias management, an object may have a static uniqueness condition (only one heap reference, but any number of stack references), or a dynamic uniqueness condition (only one reference permitted from either the stack or the heap) or some combination.

5.1 Destructive Reads

Reading a unique variable can "nullify"the variable as a side effect of the read, ensuring only a single pointer remains to the object [12, 16, 5, 4]. Destructive reads can be implemented efficiently (if the destruction can be delayed until the variable is dead, it need never be physically performed), but their semantics are generally at odds with the expected behaviour of assignment.

5.2 Swapping

The value of a unique variable can be atomically exchanged with the value of another unique variable — preserving the uniqueness of both variables. A destructive read swaps the value of a unique variable with null, so swapping is basically a more flexible version of a destructive read, with similar costs and benefits [10].

5.3 Copy Assignment

Reading a unique variable results in a *copy* of the unique object being created, again preserving the uniqueness of the object stored in the variable [1]. Eiffel's expanded types are effectively passed and updated by copy assignment, as are parameters passed by value in Pascal, so their semantics are reasonably conventional — if not necessarily object oriented, because frequent creation of new objects devalues object identity. Of course, copy assignments are computationally expensive thanks to all the copying.

5.4 Strict Linearity

A unique (or linear) variable *must* be used (consumed) exactly once on each path through the function in which it is defined. Baker's Linear Lisp uses strict linearity for its variables [2]. As a result, linear variables which are unused along some execution paths must be explicitly "killed" before the end of the function. This condition is similar to destructive reads, with the added requirement that every linear variable *must* be destructively read within the scope of their definition. Baker compares the resulting programming style to FORTH programming with an explicit argument stack: every argument a function could consume must be consumed by the function (popped of the FORTH argument stack) before a function returns, to ensure consistent function calling conventions. Strict linearity can lead to efficient code generation, but the programming style required is quite unorthodox.

5.5 Non-strict Linearity

The simplest non-strict linearity condition is that a variable may appear at most once within the scope of its definition — more complex non-strict conditions allow variables to appear more than once but somehow restrict assignments [5, 8]. This can only be applied to dynamic variables allocated on the stack, because variables with longer lifetimes (heap variables in objects, or global variables) can be accessed many times from different dynamic contexts (stack frames).

5.6 Alternatives to Uniqueness

It is also important to note in passing that several alias management schemes do not depend on full uniqueness — they typically limit the extent of a protected object (those objects which may refer to the protected object), but do not require that only one reference exist to the protected object. Schemes based on object ownership [6, 17] of one kind or another fall in to this category (semi-unique?), as do schemes which restrict objects based on their types [9, 3].

6 Phase and Granularity

As with typing, array bounds checks, or other kinds of program condition checking, alias conditions can be checked either statically, before the program is executed, or dynamically, while the program is running.

Static checks are typically more difficult to perform (because the checker must somehow consider conditions across all possible program executions), but can be more effective, catching problems before a program is ever executed (indeed, before it is even completely written) and more efficient, because no performance penalty needs to be paid for checks at runtime.

Dynamic checks can be easier to perform, although they can greatly reduce the performance of the resulting program, and only catch problems which actually arise at runtime. Because of this dynamic checkers can be more "forgiving" — where a static check may raise an error due to a potential aliasing problem, a dynamic check will only raise an error when a problem actually occurs.

Again, this is similar to type checking — static checkers are more reliable, while dynamic checkers are more accommodating. For example, a static checker (say based on a static ownership type system such as [6]) will typically disallow the construction of heterogeneous lists (lists where each member has a different owner or uniqueness status) while a dynamic checker could allow such a list to be constructed, but may signal an error at runtime if the list elements are not accessed appropriately.

Related to the issue of when checks are performed is the modularity or granularity of the checking. Some systems (such as Confined Types) require just the source code of a single method or class to test, while other systems may require whole packages [3], whole inheritance hierarchies [17], or the whole program [1] to be checked in one pass.

7 New Wine in Old Bottles?

Finally, to be useful, alias management must be integrated into programming languages. There seem to be three major approaches to integrating aliasing management into programming languages: supplementing the language via extratextual comments, extending programming languages, and using external assertion languages.

7.1 Language Supplementation

The simplest way to provide some form of aliasing management is to supplement the syntax of the programming language. This approach can be traced as far back at least as far as the Lint programmer checker for C. More recently, the

Extended Static Checker [7] generates and checks proof conditions about aliasing to catch common programming errors, based in part on declarations supplied by the programmer as special comments which ESC interprets. Other, more modest proposals such as Confined Types [3], have been implemented by comments in the program text which carry meaning to an external program checker.

7.2 Language Extensions or Changes

The most common technique for managing aliasing in programming languages is to extend the language. Generally, these approaches add keywords to variable or type declarations to indicate that the variable or type they declare must obey some extra restrictions related to aliasing — often to do with uniqueness. This approach was pioneered by Islands [12] and also adopted by Balloons [1] and Flexible Alias Protection [17].

Languages generally need to be extended for one of two reasons. First, some alias management schemes need extra language-level operations (such as destructive reads or copy assignments) which are not provided in most programming languages, as in Islands and Balloons. Second, some schemes are so intrinsically complicated that support in a preprocessor is too difficult to provide — for example, Flexible Alias Protection requires the full complexity of parametric types, and this is difficult to provide in a preprocessor outside the language, especially if the language in question does not support parametric types.

7.3 External Assertion Languages

Rather than supplementing existing programming languages, aliases can be managed by extra-programmatic reasoning, that is, by forming statements about objects and references in the program which are not incorporated into the text of the program, and thus are not bound by any particular programming language. This approach has mostly been adopted when handling aliases as part of larger efforts to prove statements about program properties. For example, Wills' Fresco system [18] supports assertions about aliasing within the framework of a system for refinement of specifications into Smalltalk programs, and Hoare's Trace Model supports assertions about the general topology of pointer-based data structures [11].

7.4 Analysis of Existing Languages

Finally, alias management techniques could require no programmer supplied information about the program's aliasing properties, rather, they could compute all the required information directly from the program's source text. This would be the most difficult technique to implement, because no specialised information is provided to the aliasing analysis, but, precisely because the programmer has to make no effort to supply information, it could be the easiest to use.

The difficultly with this approach is that it must rely on complex analyses which may need to be performed at large granularities (ultimately requiring whole program analysis) and which may be brittle and fail (or detect errors) in ways which most programmers would find unpredictable.

8 Conclusion

Aliasing is endemic in object-oriented programming [17] and quite a large number number of programming language mechanisms have been proposed to manage, control, serve, and protect objects against the pernicious effects of aliasing. A successful alias management scheme has to balance flexibility and expressibility against protection and paranoia, while somehow integrating within existing programming languages and supporting common programming styles. This is not a easy task, and many open questions remain unanswered.

Acknowlegements

Thanks to David Clarke for his comments on this position paper, to John Potter for his email, and to the anonymous reviewers. This work *was* supported by Microsoft Australia Pty Ltd

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP Proceedings*, 1997.

[2] Henry G. Baker. 'use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1), January 1995.

[3] Boris Bokowski and Jan Vitek. Confined types. in preparation, 1999.

[4] John Boyland. Deferring destruction when reading unique variables. in preparation.

[5] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *ICSE Proceedings*, 1998.

[6] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.

[7] Dave Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extened static checking. Technical Report 159, COMPAQ SRC, 1998.

[8] Dave Detlefs, Greg Nelson, and K. Rustan M. Leino. Virginity: A contribution to the specification of object-oriented software. Technical Report Technical Note 1997-001, COMPAQ SRC, 1998.

[9] D. Genius, M. Trapp, and W. Zimmermann. An approach to improve locality using sandwich types. In *2nd Types in Compilation Workshop*, volume LNCS 1473, 1998.

[10] D. E. Harms and B. W. Weide. Copying and swaping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5), May 1991.

[11] C.A.R. Hoare and He Jifeng. A trace model of pointers and objects. in preparation, 1999.

[12] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.

[13] 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.

[14] Barbara Liskov and John V. Guttag. *Abstraction and Specification in Program Development.* MIT Press/McGraw-Hill, 1986.

[15] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[16] Naftaly Minsky. Towards alias-free pointers. In *ECOOP Proceedings*, July 1996.

[17] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP Proceedings*, 1998.

[18] Alan Wills. Capsules and types in Fresco. In *Proceedings ECOOP '91.* Springer-Verlag, July 15-19 1991.