The Geneva Convention On The Treatment of Object Aliasing

John Hogg
Bell-Northern Research

Doug Lea
SUNY Oswego

Alan Wills
University of Manchester

Dennis deChampeaux
Hewlett-Packard

Richard Holt
University of Toronto

1 Introduction

Aliasing has been a problem in both formal verification and practical programming for a number of years. To the formalist, it can be annoyingly difficult to prove the simple Hoare formula {x = true} y := **false** {x = true}. If x and y refer to the same boolean variable, i.e., x and y are *aliased*, then the formula will not be valid, and proving that aliasing cannot occur is not always straightforward. To the practicing programmer, aliases can result in mysterious bugs as variables change their values seemingly on their own. A classic example is the matrix multiply routine mult(left, right, result) which puts the product of its first two parameters into the third. This works perfectly well until the day some unsuspecting programmer writes the very reasonable statement mult(a, b, a). If the implementor of the routine did not consider the possibility that an argument may be aliased with the result, disaster is inevitable.

Over the years, solutions or workarounds have been found for aliasing problems in traditional languages, and the matter is seemingly under control. Unfortunately, as described below these solutions tend to be too conservative to be useful in object-oriented programs.

The object paradigm has been sold partly on the basis of the strong encapsulation that it provides. This is a misleading claim. A single object may be encapsulated, but single objects are not interesting. An object must

be part of a system to be useful, and a system of objects is not necessarily encapsulated.

However, the picture is not entirely bleak. Some partial solutions to the object-oriented aliasing problem have been put forward. More research is needed to fill in the rest of the puzzle, but the future looks bright if researchers treat the problem from a truly object-oriented perspective.

This Convention defines and explains aliasing in the object-oriented context. Various approaches are described, and proposals are made as to areas for future research.

2 Definitions

We will start by defining a few terms informally. The model of object-oriented systems used is a very simple one with a Smalltalk flavour, and terms that are not further defined can be assumed to have their Smalltalk meanings. Every object is either *primitive* (such as an integer or boolean) or *constructed*. A constructed object has a set of *instance variables* which hold references to other objects. (Every variable is thus implicitly a pointer variable.) Every object also has a set of *methods* that make use of instance variables, *local variables* and *parameters*. A method is invoked by sending a *message* to an object, and may in turn send other messages and set variables.

The set of (object address) values associated with variables during the execution of a method is a *context*. It is only meaningful to speak of aliasing occurring within some context; if two instance variables refer to a single object, but one of them belongs to an object that cannot be reached from anywhere in the system, then the aliasing is irrelevant.

In accord with Smalltalk, and with Snyder's Abstract Object Model [Sny91], we assume that *all* method argument and result passing is performed in a manner classically described as "by-reference", but perhaps better labelled as "by-identity".

Objects may be accessed "directly" through a bound variable or pseudovariable, or "indirectly" through the instance variables of other objects. More generally an access *path* is a sequence of variable names. The evaluation of the first variable within the current context yields a new context that is an object with a set of instance variable, and the evaluation of each successive variable in the path yields a further object and context. In particular, access paths may include traversals through the objects held in "collections".

Within any method, objects may be accessed through paths rooted at

any of:

1. self.
2. An anonymous locally constructed object.
3. A method argument.
4. A result returned by another method.
5. A (global) variable accessible from the method scope.
6. A local method variable bound to any of the above.

An object is *aliased* with respect to some context if two or more such paths to it exist.

¿From a conceptual, rather than operational stance, aliasing occurs when one object is accessed through more than one if its possible "roles" in a given program (where the different roles are indicated by multiple names or access paths), and aliasing is a problem whenever these roles conflict. Such conflicts can be as simple as trying to simultaneously serve as a source and destination in a matrix multiplication, or as intricate as an account crediting a payment from itself through various side channels in a complex financial system.

Determination that an object referred to in two different roles is actually the same can be just as surprising conceptually as it is difficult analytically; discovering and dealing with alias conditions in a program can be a semantically meaningful challenge, not merely a technical exercise in determining correctness and safety properties. To illustrate with an ancient example, for many centuries astronomers used the distinct terms "evening star" and "morning star" without realizing that both referred to one object, the planet Venus. While this is disanalogous to programming situations in that detection of aliasing is a matter of analysis rather than scientific discovery, in large enough programs the two are difficult to distinguish.

While some aliasing problems may thus result from insufficient problem analysis, leading to situations in which roles accidentally conflict, perhaps they more typically arise out of class and method design and implementation decisions that either ignore the possibility of aliasing or intentionally disallow it without making this fact visible to clients.

In any case, the fact that objects are referred to by variables describing their roles, but are actually manipulated in terms of their identities means

aliasing is essentially always present, and aliasing problems are always possible within the object-based ([Weg87]) paradigm, whether or not constructs like inheritance, concurrency and persistence are supported. However, such features do accentuate and complicate problems. Even simple subclass polymorphism can make aliasing opportunities even more difficult to notice and appreciate, especially in statically typed languages. For example, in a C++ function f(A& a, B& b), a and b may indeed be aliased if B is a subclass of A or vice versa.

3 Aliasing and Objects

The aspect of an object-based system that *does* set it apart from traditional procedural languages is the presence of persistent local state. An object encapsulates a set of variables which are not externally visible, yet which retain their values between method invocations. In traditional languages, all aliasing is *dynamic* in the sense that it only exists for the duration of a particular scope entry. (Global variables exist for the duration of a global scope.) Upon scope exit, any aliases that were created within that scope disappear. By contrast, when a method scope is left, only the dynamic aliases involving its parameters and temporary variables go away. Instance variables retain their values, and this *static* aliasing will still be present when the object scope is reentered.

We worry about aliasing because the execution of a method may change the behaviour of a seemingly uninvolved object, and this may happen even without the affected object being accessed. This is because the real state of an object is not fully specified by just its variables, but also upon the states of the objects to which these variables refer. The value of a predicate (in denotational terms) or the result of a method (in operational terms) may depend upon the states of any object that can be reached from its context, and thus the state of an object is the state of the transitive closure of objects that can be reached from it. Therefore, two objects are *effectively aliased* if the transitive closures of the objects reachable from them have a non-empty intersection.

For example, consider a bank object containing a number of Portfolios, among them port1 and port2. Each Portfolio has (among other attributes) a chequingAccount of class Account. An Account instance understands the methods debit: and credit: which decrease and increase its balance respectively. These methods are also understood by a Portfolio, which will apply

them in turn to its chequingAccount. A Portfolio instance also understands a method transferTo:amount: which will debit itself and credit its first parameter by the amount of the second parameter. Now, will port1 transferTo: port2 amount: $100.00 really decrease the amount of money in port1?

There are two ways in which this can fail to happen. First, port1 and port2 may be the same portfolio, i.e., port1 and port2 are aliases. This can at least be recognized from within the bank object: we can require that port1 $\not\equiv$ port2, where $\not\equiv$ is an object identity comparator.

Unfortunately, port1 will also have an unaltered final balance if port1 and port2 are different, but they share a common chequingAccount. That is, if the chequing account is aliased with respect to the bank context, then the two portfolios are effectively aliased. This is more difficult to deal with because we cannot even express the idea in a programming language that claims to provide encapsulation. When the transferTo:amount: method of port1 is entered, there is no way to refer to the Account object held by the Portfolio first parameter.

4 The Treatment of Aliasing

While the object aliasing problem has been known for some time [Mey88], few discussions have reached print. We broadly categorize approaches that we are aware of in terms of:

Detection. Static or dynamic (run-time) diagnosis of potential or actual aliasing.

Advertisement. Annotations that help modularize detection by declaring aliasing properties of methods.

Prevention. Constructs that disallow aliasing in a statically checkable fashion.

Control. Methods that isolate the effects of aliasing.

4.1 Alias Detection

Alias detection is a *post hoc* activity. Rather than determining beforehand whether variables could reasonably be expected to alias the same object, it determines those alias patterns potentially or actually present in a given program through static (compile-time) or dynamic (run-time) techniques.

Especially in the absence of *a priori* information, it is useful for compilers, static analysis tools, and programmers to detect aliasing conflicts present in programs. Compilers can then generate more efficient code, static analyzers can assist formalists in discovering cases where aliasing may invalidate predicates, and programmers can specially deal with troublesome conflicts.

Because aliasing is usually a non-local phenomenon, static detection requires NP-hard "interprocedural" analysis ([LR91]), resulting in information about whether any two variables *never*, *sometimes*, or *always* alias the same object, where *sometimes-aliased* refers to situations in which variables are aliased during some invocations but not others, including paths that are not necessarily taken during actual execution.

The *never-aliased* and *always-aliased* cases can be very useful for optimization purposes. For example, two arrays that are never aliased can be independently manipulated in vector processors, while two variables that are always aliased can be represented by a single pointer.

Unfortunately, given the ubiquity of aliasing opportunities in object-oriented programs, full analyses are likely to be too slow to be practical, and to result in the *sometimes-aliased* case more often than not. However, techniques like message-splitting and customization pioneered in **self** ([CU91]) show some promise for improving matters. For example, automatically splitting off and specially generating code for aliased versus non-aliased versions of a method may both simplify further analysis and allow further optimizations.

Programmers themselves should write code to detect aliasing conflicts at run-time, and take evasive action. However, this is not always possible: Run-time alias detection via object identity comparison ($\not\equiv$) is not fully supported in most object oriented languages. While "pointer identity" operations can sometimes be used for such purposes, they may not always work in all cases. Notable examples include C++ variables where the same object is referred to in terms of more than one of its multiply-inherited base classes, and, in languages without full support for object persistence, identity preservation for objects recovered from secondary storage. Also, as described above, access protection may impede a programmer's ability to check identities.

4.2 Alias Advertisement

Because global detection is impractical, it is important to develop methods and constructs that can lead to more modular analysis. Both programmers and formalists could benefit from constructs that enhance the locality of

analysis by annotating methods in terms of their resulting aliasing properties.

Without evidence to the contrary, people tend to make optimistic assumptions about the aliasing properties of methods. For example, most programmers would find it very surprising if the or(arg) method of a Boolean object were programmed to return self (if self held True) else arg. Even though this is "correct" behaviour, programmers expect or to return a new object that would not be aliased to either of the operands in later expressions.

Yet, popular object oriented languages have no means of indicating whether methods "capture" objects by creating access paths (instance variables, return variables, globals) that persist beyond their invocation.

Constructs or annotations indicating which object bindings are captured by a method and/or which aliases a method is able to cope with could play a role similar to, but independent of, qualifiers like const in C++ and related constructs that integrate useful subsets of full behavioural specifications. As with annotations describing mutability, "negatively" expressed qualifiers are likely to be more useful.

Thus, in the same way qualifying a parameter with const advertises that the argument is not modified, an uncaptured qualifier could indicate that an object is never bound to a variable that could cause it to be *further* modified via side channels after the method returns.

For example, a typical constructive implementation of *or* may then declare that both self and arg are both const and uncaptured and that the return variable is also uncaptured.

In addition to indicating (as a postcondition of sorts) that aliases not be propagated, a method may similarly be advertised with the restriction (precondition) that actual arguments *never* be aliased, via a construct like noalias. This is the default restriction in Turing ([HMRC88]).

The pattern of const, noalias, uncaptured operands and uncaptured results is an object-oriented analog of "pure" functions (as opposed to procedures). As discussed in [Hog91], languages that specially mark such operations in a special category (as in Turing) thereby enhance informal and formal reasoning about program behaviour.

Actual enforcement of qualifiers like uncaptured and noalias leads to the notion of alias prevention.

4.3 Alias Prevention

Alias prevention techniques introduce constructs that promise that aliasing will not occur in particular contexts in ways that guarantee static checkability by compilers and program analyzers.

Static checkability requires conservative definitions of constructs. For example, a checkable version of uncaptured might prohibit *all* bindings of a variable within a method except in calls to other methods with uncaptured attributes. This would prohibit uses that programmers happen to know do not propagate aliases, but cannot be syntactically determined to be safe.

A statically checkable form of noalias would be even more draconian. For example, the rules used for aliasing prevention in Turing [HMRC88] assume that any change to a single entity in a collection is assumed to have affected the entire collection. As a result, the strong aliasing protection of Turing cannot be applied to Object Turing without losing the ability to express common object-oriented idioms.

Conservatism is useful in that it ensures validity: the formalist will not be able to prove formulas that could be invalid due to aliasing, and the programmer will not be able to compile code in which aliasing could produce surprises such as in the example above. However, valid formulas may not be provable (i.e., the proof system is not even complete in the sense of [Coo78]), and perfectly safe code may not compile without errors or warnings.

Thus, fine grained alias prevention constructs have limited utility. Higher level constructs are required in order to overcome these problems.

Islands [Hog91] provide a mechanism for isolating a group of closely-related objects. A set of syntactic mechanisms are used to ensure that no static references can exist across the boundary of an island. An atomic assignment operation that sets a previous reference to null allows objects to be passed in and out across this boundary. Within an island, any system of aliasing control can be used, but a nested island is a completely encapsulated unit. This means that the prevention mechanism scales, and the control strategy can therefore be confined to small groups of objects and need not scale.

A more radical approach is that of [HW91], in which the traditional assignment operator that copies its right side to its left is replaced by a swapping operator that exchanges the bindings of its two sides. By avoiding reference copying, aliasing is also avoided, and its problems disappear. Naturally, the programmer must learn a different paradigm. It is unclear whether this paradigm can mesh well with mainstream object oriented pro-

gramming techniques.

4.4 Alias Control

Aliasing prevention is not sufficient in itself because aliasing is not avoidable under the conventional object-oriented paradigm. There will remain cases in which the effects of aliasing cannot be determined without taking into account the runtime state of a system. Under these circumstances, aliasing *control* must be applied. The programmer must determine that the system will never reach a state in which there will be unexpected aliasing, even though this is not precluded by an examination of the code components in isolation. The formalist must show that no predicate is affected by being effectively aliased with the left side of any assignment statement. Control is thus based on some analysis of state reachability.

A proof system for an object-oriented language (SPOOL) is given in [AdB90]. This uses aliasing control exclusively; there is no prevention component to the management strategy. The predicate language is an extension to the programming language in which encapsulation is removed. Within some context, variables in other objects can be referred to using variable paths as defined earlier. In the example above, we could assert that port1:chequingAccount $\not\equiv$ port2:chequingAccount from the context of the bank. This approach is impractical in anything but the smallest application, but it forms the foundation for future work.

In [Wil91] an approach to aliasing control based on *demesnes* is proposed. This concept is related to the reaches of [LG88]. A demesne is a set of objects which participate in the representation of a given value. The programmer defines for each class a demesne-function, which yields a union of the singleton set containing self, and the demesnes of some or all of the instance variables. 'Backward' pointers and cache variables would be amongst those omitted. Several named demesnes may be provided for one class. The functions need not be implemented, but are used to reason about a program. For example, statements can be made about whether the demesnes of two parameters are allowed to intersect. Statements about framing (what objects may be changed by a method) made in terms of demesnes preserve encapsulation, since the detailed definition of the demesnes is made within their own classes.

5 Conclusion

The aliasing problem is attracting an increasing amount of attention. Component reuse requires adequate description of component behaviour, and this can only be given if components are sufficiently encapsulated for their behaviours to be predictable. To ensure this, aliasing must be detected when it occurs, advertised when it is possible, prevented where it is not wanted, and controlled where it is needed.

References

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

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *OOPSLA '91 Proceedings*, October 1991.

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

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

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA '91 Proceedings*, October 1991.

[HW91] D.E. Harms and B.W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[LR91] William Landi and Barbara Ryder. Pointer-induced aliasing: A Problem taxonomy. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

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

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

[Sny91] Alan Snyder. Modelling the C++ object model: An Application of an abstract object Model In *ECOOP '91 Proceedings (Springer-Verlag LNCS 512)*, pages 1–20, July 1991.

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

[Wil91] Alan Wills. Capsules and types in Fresco: Program verification in Smalltalk. In *ECOOP '91 Proceedings (Springer-Verlag LNCS 512)*, pages 59–76, July 1991.