Implementing "Object Ownership to Order"

IWAOOS'99 Position Paper
Boris Bokowski
bokowski@acm.org
GMD-FIRST, Berlin, Germany

So far, no practical way of controlling aliasing in object-oriented programming has been found. On the one hand, approaches like Eiffel's `expanded` [1], Hogg's Islands [2], Almeida's Balloons [3], and Confined Types [4] are too restrictive because they rule out common programming idioms and patterns. On the other hand, Flexible Alias Protection [5, 6] claims to be a more flexible way of controlling aliasing that still allows common programming styles. However, the proposed scheme is rather complex, so that it is difficult to make such a claim without having experience with larger alias-protected example programs. Moreover, it is not yet clear whether the scheme can be integrated into a non-toy programming language.

Using CoffeeStrainer [7], a framework for checking programmer-defined constraints on Java programs, we have implemented "Object Ownership to Order" [8], a refinement of the original Flexible Alias protection proposal. The implementation serves two purposes: First, it shows that the CoffeeStrainer framework is useful for implementing language extensions that can be expressed as static constraints. Second, it helps validating the theoretical results for a real-world programming language, it allows to experiment with variations of the alias protection scheme, and it enables automated checking of larger examples.

Usually, implementing language extensions is difficult. Either a new compiler has to be written, or an existing compiler has to be modified. Unfortunately, the latter approach does not work well because most compilers are monolithic programs that are not open to modifications and extensions. With the implementation of "Object Ownership to Order", we have shown that CoffeeStrainer is a good platform for implementing language extensions that can be expressed as static constraints. Originally, CoffeeStrainer has been written for framework programmers who want to express constraints on the way their framework should be used. The constraints, which are embedded within special comments in classes or interfaces, may apply to the *definition* of the classes and interfaces in which they appear, and to all places in a program where the enclosing classes or interfaces are *used*.

The experiences while implementing Flexible Alias are threefold: experiences with automated alias-checking of larger example programs, Java-specific issues, and repercussions on the original proposal.

Experiences with alias-checking: Using the implementation, we have alias-checked a number of example programs. As expected, programs that are designed with the program's aliasing properties in mind can be annotated as is. However, adding ownership tags in a way that tries to control aliasing as much as possible leads to a large number of ownership parameters for each class. Other programs that have been designed with less care (or more design patterns) seem to prevent alias protection to a certain degree. For example, annotating and checking the original implementation of `java.lang.Vector` showed that it is not yet possible to define binary methods such as `equals()` when using Flexible Alias Protection. Another problem is the implementation of the corresponding `Enumeration` class which directly accesses internals of the vector object.

Java-specific issues: While it was easy to handle some features not present in the original proposal, such as `null`, arrays, and primitive types, others were problematic. Unlike instance fields and methods, which have a context in which the most important rule – no role confusion – can be checked, there is no such context for static methods and fields. Consequently, no objects may be manipulated by static methods or stored in static fields. Of course, this is not satisfactory; a solution seems to be to parameterize static methods with ownership tags similar to the parameterization of classes. Inner classes are not supported, because this would require an extension of the proposal since inner objects have access to their enclosing instances.

Repercussions on the original proposal: In addition to the extensions of the original proposal that have been hinted at in the previous paragraph, we found that the flat parameterization with ownership tags is not sufficient: Simple objects can be stored in alias-protected containers, but if alias-protected containers themselves are to be stored in other alias-protected containers, nested parameterization is needed. Additionally, it could be useful to allow parameterization not only of static methods, but of methods in general.

[1] B. Meyer, *Eiffel: The Language*, Prentice-Hall, 1992
[2] J. Hogg, *Islands: Aliasing Protection in Object-Oriented Languages*, Proceedings of OOPSLA'91
[3] P.S. Almeida, *Balloon types: Controlling Sharing of State in Data Types*, Proceedings of ECOOP'97
[4] B. Bokowski, J. Vitek, *Confined Types*, manuscript
[5] J. Noble, J. Potter, J. Vitek, Flexible Alias Protection, Proceedings of ECOOP'98
[6] D. Clarke, J. Potter, J. Noble, *Ownership Types for Flexible Alias Protection*, Proceedings of OOPSLA'98
[7] D. Clarke, R. Shelswell, J. Potter, J. Noble, *Object Ownership to Order*, manuscript
[8] B. Bokowski, *CoffeeStrainer – Statically Checking Structural Constraints on Java Programs*, Technical Report B-98-14, FU Berlin, Inst. f. Informatik, http://www.inf.fu-berlin.de/~bokowski/CoffeeStrainer