Dynamic Alias Protection with Aliasing Contracts

Janina Voigt

Computer Laboratory, University of Cambridge
JJ Thomson Avenue, Cambridge CB3 0FD, UK
janina.voigt@cl.cam.ac.uk

1 Problem Description

In typical object-oriented (OO) programming languages, objects have reference types; an object variable does not contain the object itself, but the address of the object on the heap. Therefore, one object can be referenced by multiple variables at the same time. This situation is known as *aliasing*.

Aliasing is a central feature of OO; it allows objects to be shared by different parts of a program, enabling the efficient implementation of important idioms, such as iterators. However, it also causes serious problems in software development since it reduces modularity and encapsulation, making programs difficult to understand and bugs hard to trace. To avoid these issues and reduce the risk of alias-related bugs, it is therefore important to more tightly control aliasing.

Much research has been done aiming for such alias control. Important alias protection schemes include Hogg's islands [1], Clarke et al.'s ownership types [2] and Müller et al.'s universe types [3]. These schemes allow the specification that one object is encapsulated inside another object, disallowing any external references to the encapsulated object. These aliasing specifications can usually be checked *statically*.

With Clarke et al.'s ownership types, for example, each object is *owned* by another object, producing a tree-shaped ownership hierarchy called the ownership tree. All paths to an object must pass through its owner, thus disallowing aliases from the outside.

The approach taken by many existing alias protection schemes is restrictive because some common idioms, such as iterators, require sharing of objects. This is not possible if an object is fully encapsulated by another object; in addition, an object cannot usually be transferred from one owner to another at runtime.

Although some extensions (for example [4–6]) have been proposed to add more flexibility to existing schemes, for example by accommodating so-called *multiple ownership* and *ownership transfer*, most schemes still support only single, non-transferable ownership; statically checking this is straightforward.

2 Proposed Solution: Aliasing Contracts

We believe that the lack of flexibility in existing alias protection schemes has so far prevented their uptake by mainstream programmers; this is problematic because of the importance of alias control in regards to encapsulation and modularity in software. Our aim, therefore, is to develop a more flexible and expressive alias protection scheme which can express and control a wide variety of aliasing conditions, including the single ownership already supported by other schemes, as well as multiple ownership and ownership transfer.

To this end, we introduce *aliasing contracts* which can be used to express and enforce assumptions about the circumstances under which an object can be accessed. If an aliasing contract is violated, this represents an unexpected object access which may in turn indicate a bug. We envisage that aliasing contracts will be used during testing to search for such alias-related bugs.

An aliasing contract consists of two boolean expressions attached to a variable declaration; the first boolean expression states under which circumstances the *object to which the variable points* can be read, while the second boolean expression concerns write accesses.

Access to an object is allowed only if *all* of the contracts of variables currently pointing to the object are satisfied; contracts thus essentially specify *preconditions* for object accesses. Unlike many existing alias protection schemes, aliasing contracts thus do not limit aliasing itself but restrict when aliases can be used to access objects.

For each contract, we call the nearest enclosing object the contract's *declaring* object. Whenever a contract needs to be evaluated, this is done in the context of the declaring object; that is, during contract evaluation `this` points to the declaring object.

Contracts may make use of two special variables: `accessed` points to the object which is being accessed; `accessor` points to the object requesting the access. The value of `accessor` is determined immediately prior to contract evaluation and so, for a given contract, may vary from one evaluation to the next.

We also introduce the special binary operators `canread` and `canwrite` which check if one object has read or write access to another. These allow us to define the contracts `accessor canread this` and `accessor canwrite this` which simply pass on contract evaluation to the declaring object; the object stored in a variable can thus be accessed if the variable's declaring object can be accessed. These contracts are very useful, as we show in our example below.

We further introduce *encapsulation groups* which allow objects to be grouped; group membership can then be tested in contracts using the `in` operator. Encapsulation groups are very powerful since they allow us to give access to not just a single object but any object in a group, thus supporting multiple ownership.

To see how aliasing contracts can easily describe complex aliasing conditions, we consider the example of a `LinkedList`. A `LinkedList` holds a reference to the `head` node of the list and each following `Node` then holds a reference to the `next` node. Each `Node` also contains a piece of `Data`.

We want the `LinkedList` to be able to read and write all of its `Node` objects; similarly, we want `Node` objects in a list to be able to access each other. The `Data` in each `Node`, on the other hand, should be accessible by other parts of the system. Below, we show the aliasing contracts corresponding to this situation.

We use encapsulation groups to make `Node` objects in the list accessible to the `LinkedList` and all other `Nodes` in the list; we use the `canread` and `canwrite` operators to apply the same contract to each `Node` in the list.

```
class LinkedList {
  private Node head {accessor in allNodes, accessor in allNodes};
  group allNodes = {this, head, head.nextNodes};
}

class Node {
  private Node next {accessor canread this, accessor canwrite this};
  private Data data {true, true};
  group nextNodes = {next, next.nextNodes};
}
```

The example above shows how easy it is to express complex aliasing conditions using aliasing contracts, achieving our stated research goal. Encapsulation groups enable us to easily group objects and give ownership to all objects in the group, rather than just a single one. Transferring an object from one variable to another achieves ownership transfer, as one contract is removed from the object and another one added.

Using aliasing contracts, we can model all of the existing alias protection schemes mentioned above. This shows that aliasing contracts form a superset of existing schemes.

Unlike existing alias protection schemes, in the general case aliasing contracts cannot be checked statically and require dynamic checking. At runtime, when an object is accessed, the contracts of all variables pointing to it must be checked; this represents a significant performance overhead and a disadvantage over statically checkable alias protection schemes.

While we have not yet quantified the performance overhead of aliasing contracts, we envisage them as a testing tool which will be enabled during the unit testing stage of software development but disabled in production code. In addition, we expect that a number of simple aliasing contracts can be checked statically, enabling us to remove them before program execution.

3 Methodology

We have developed operational semantics for aliasing contracts; these form an important base for the remainder of our work by unambiguously stating the exact semantics of aliasing contracts.

In addition, we have implemented a prototype for aliasing contracts in Java. We selected Java due to its popularity and the large number of open source programs written in Java, providing us with many potential test programs.

For the prototype, we modified the Java compiler `javac` from the OpenJDK [7] to inject contract checking statements into the source code; these contract checks call a runtime library we developed which keeps track of contracts at runtime. When an assignment takes place, an object is transferred from one variable to another; this requires a call to our library to remove the old contract and add the new one. When an object access is made, our runtime library is called to check all contracts currently associated with the object; if one of the contracts evaluates to `false`, an exception is thrown to report the error.

Our prototype shows that adding aliasing contracts to a standard language like Java is feasible. As future work, we plan to run the prototype on a corpus of programs to quantify the performance overhead caused by aliasing contracts.

We are also currently working on a static checker for simple aliasing contracts to address the performance overheads caused by dynamic contract checking. The static checking we have developed so far can check many simple aliasing contracts; such contracts could be removed during compilation, significantly decreasing the number of contracts which need to be checked at runtime.

4 Conclusion

We have presented a new alias protection scheme called aliasing contracts which aims to address the lack of flexibility and expressiveness inherent in existing schemes. Aliasing contracts are highly flexible and can be used to specify a wide variety of aliasing conditions, forming a superset of existing schemes.

So far, we have developed operational semantics as an exact and unambiguous specification of aliasing contracts. We have also implemented a prototype for aliasing contracts in Java which we will use to quantify the performance overhead of our scheme. We are currently working on developing static checking for aliasing contracts which will enable us to statically check certain aliasing contracts without the runtime performance overheads caused by dynamic checking.

References

1. Hogg, J.: Islands: aliasing protection in object-oriented languages. In: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '91, ACM (1991) 271–285
2. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. ACM SIGPLAN Notices **33** (1998) 48–64
3. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. In Poetzsch-Heffter, A., Meyer, J., eds.: Programming Languages and Fundamentals of Programming. (1999)
4. Li, P., Cameron, N., Noble, J.: Mojojojo - more ownership for multiple owners. In: International Workshop on Foundations of Object-Oriented Languages. FOOL (2010)
5. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In: Proceedings of the 2007 Symposium on Dynamic Languages. DLS '07, New York, NY, USA, ACM (2007) 41–52
6. Sergey, I., Clarke, D.: Gradual ownership types. In: Proceedings of the 21st European conference on Programming Languages and Systems. ESOP'12, Berlin, Heidelberg, Springer-Verlag (2012) 579–599
7. Oracle Corporation: OpenJDK. `http://openjdk.java.net/` (2013)