Abstract

In object-oriented programs, the possibility of having an alias to an object’s mutable state is a known source of bugs making programs difficult to test and maintain. We propose an access mode system for limiting the effects of aliasing by associating, with each reference, an access right to the object to which the reference refer. Mutator methods may not be invoked on read references, and only read references to the state can be obtained via any method invoked on a read reference. This enables exporting read references to objects without risking the objects being modified. The system realises access modes by annotations on variables, methods, method returns and parameters to methods. It is completely statically checkable without any need for run-time representation of modes and does not impose any run-time overhead.

1 Introduction

‘Aliasing is endemic in object-oriented programming’ [13]. The possibility of having an alias to an object’s mutable state is a known source of bugs making programs difficult to test and maintain. It also makes programs hard to reason about. In many object-oriented languages such as Java [6] and Smalltalk [5], a reference always gives full access to the referenced object’s public protocol. Thus, instances of a class with mutator methods in its public protocol cannot be exported without risking being changed by its receiver.

In this paper, we propose a system for class-based, object-oriented, statically-typed programming languages where a reference either have write access or read-only access to the object it refers. The system enables exporting objects via such ‘read references’ with a guarantee that the transitive\footnote{The transitive state of an object is the values of its fields plus the transitive states of all objects to which its fields refers, recursively.} state of the referenced object cannot be modified via those references. The system works by annotations on variables, formal parameters and method returns that are statically validated against a set of rules. The annotations control possible access rights of a reference. The annotations are non-executable and thus, a valid program can be stripped from annotations without affecting its run-time behaviour. We also annotate each method as either accessor or mutator to reflect the access right needed by the caller to invoke the method. Mutators may only be invoked on references with write access. In our system, write access implies read access and thus accessors may be invoked on any reference. The annotations on methods are statically validated prohibiting a mutator from being annotated as an accessor and vice versa.

1.1 Contributions

The main contributions of this paper are the programming model with references with different access rights and the semantics of the annotations (though only informally described in this paper) that statically guarantee that a reference’s access right is never violated without any run-time or memory overhead.

1.2 Outline

Section 2 contains a brief discussion of aliasing and read-only references. Section 3 presents the proposed system and its annotations in the form of an example. Section 4 briefly presents related approaches. Section 5 discusses the benefits and drawbacks of a system such as ours and compares it to similar proposed systems and Section 6 concludes.

2 Motivation

The aim of the proposed system is to increase alias control by limiting the possible effects of aliasing. By allowing methods to be defined as only accepting read reference arguments, argument side-effects can be eliminated where necessary. Using read references, references to data that must not be changed from outside the object can be safely exported since read references prohibits changes. Since the protection is applied to references
and not objects, an object referenced by a read reference is still mutable and may be changed via any existing writable reference to it. This change is visible to any object referencing the changed object regardless of the mode of the reference. This makes the read reference construct a both powerful and practical one, but also makes the guarantees offered by a read reference weaker that those offered by immutable objects, since there are no guarantees that the object referenced by a read reference will not be changed from elsewhere. A similar remark is also found in Boyland et.al. [2] where the dynamic semantics of a number of common language extensions are described, including a non-transitive read. In this paper, we only consider transitive protection since we believe that transitive protection is more natural that non-transitive. An example of how non-transitive protection can be circumvented is found in the discussion of const in Section 4.

Systems for limiting the effects of aliasing in general, and read-only reference systems in particular, are common building bricks in many larger systems for aliasing prevention, such as Hogg’s seminal Islands [8] and the Universes proposal [12] where read references are used to create side-effect free functions and to safely pass objects between protection domains. The presence of aliasing makes it hard to reason about a program [13]. Thus, limiting the effects of aliasing by making stateful objects non-mutable from certain references should be an aid in the construction of larger proofs of a program.

A practical example of where a system such as the one presented here is useful is in situations where data must be shared between objects of which some should only be allowed to read the data and where copying is not feasible (e.g. because of a need for synchronised data or for performance reasons). A compound object with some integral data in a subobject that must not be (wrongfully) mutated from outside of the compound may be safely exported via a read reference, even to untrusted code, since read references prohibit changes and their protection cannot be circumvented.

For a more in-depth discussion of the problems of aliasing in presence of a mutable state, see The Geneva Convention on The Treatment of Object Aliasing [9] or [13]. Discussions of how read references can be used to tackle the aliasing problem can be found in [10, 14].

3 A short description of the system

The system is essentially an annotated type system. Each type is annotated with a qualifier which controls the access right of the reference stored in the variable. Each method is also annotated to advertise whether it mutates its receiver object or not. We use the qualifiers read, write and context to annotate the types in declarations of variables, formal parameters and method returns. A variable, formal parameter or method return annotated with read can only hold/return read references. A variable, formal parameter or method return annotated with write can only hold/return write references. A variable, a formal parameter or a method return annotated with context holds/returns a read reference in a method invoked on a read reference and a write reference otherwise. This enables a read method (see below) to return write references to the state when invoked via a write reference.

Methods are annotated as either read (a read method) or write (a write method). In the body of a read method, only read methods can be invoked on member variables or context variables and assignment to member variables is not allowed.

3.1 Example

The example in Fig. 1, admittedly somewhat contrived for pedagogical reasons, makes use of the mode system in the construction and use of an event class. The example is written in a Java-like language extended with mode qualifiers on variables and methods. The purpose of the event is to notify radiator objects in a room object that a thermometer object has been inserted into the room.

In (18) a room object that holds a write reference to the newly inserted thermometer creates a new ThermEvent object and assigns it to the variable event. The variable event is annotated with the write qualifier to denote that the variable will always hold references that can be used for modification of the referenced object.

In (19), the room invokes setSource with this casted as read passed as an argument. The casting makes the exported reference a read reference, i.e. a reference that cannot be used for modifications of the referenced object. Then the setTherm method is invoked with the write reference to the thermometer passed as an argument (20). The reference to the thermometer is stored in the thermometer member variable in event (8). The declaration of the member variable thermometer is extended with the context qualifier (3) which means that the thermometer member variable will be treated differently depending on the method where it is used – as a write variable in methods that have write access to its enclosing objects’ states and as a read variable in methods that only have read access to their states. A method is declared as either write or read by appending the qualifier write or read to the method prototype declaration, e.g. (4). In (21), the radiators in the room are notified about the newly inserted thermometer by invocation of notify (implementation of notify not shown here).
public class ThermEvent {
    private read Object source;
    private context Thermometer thermometer;
    public void setSource(read Object s):write {
        this.source = s;
    }
    public void setTherm(write Thermometer th):write {
        this.thermometer = th;
    }
    public read Object getSource():read {
        return this.source;
    }
    public context Thermometer getTherm():read {
        return this.thermometer;
    }
    ...}

write ThermEvent event = new ThermEvent();
event.setSource(read this);
event.setTherm(thermometer);
radiators.notify(read event);

Figure 1: Example of an event class annotated with mode qualifiers

passing event casted to read to protect it from modifications.

The member variable source holds a reference to the creator of the event object. Since source is declared as read (2) it is safe to export it from the event via the getSource() method rest assured that no receiver will be able to modify it or any of its subobjects via the exported reference.

4 Related approaches

A system for restricting access to objects has long since been present in C++ in the const construct (const pointers, references and methods). In C++, pointers and references may be declared const meaning that only const methods may be invoked on them. A const method may not update any fields in its receiver, but does not treat the references in the fields as const, allowing non-const methods to be invoked on the references. Hence, C++ const offers only non-transitive protection and subsequently, a pointer to this in a field would void the entire protection since non-const methods may be invoked on the alias. Also, the protection offered by const may be circumvented easily since C++ allows ‘constness’ to be cast away.

In contrast to the non-transitive protection offered by C++ const, several authors have proposed read-only constructs which offer transitive protection, most notably [8, 10, 11].

Hogg’s seminal Islands paper [8] includes a read access mode on variables, part of an aliasing prevention system, which offers transitive protection. The read variables may however not be statically aliased or be on the right hand side of an assignment. No system for static checking is presented.

Kniesel and Thiesen [10] define a similar construct for Java named JAC – Java with Transitive Access Control that enables transitive protection in the same fashion as we do. Our improvement over this system is the ability to return references to the state from read methods and our independence on a particular type system. Also, in our system, functionality cannot be effectively hidden since the annotation of a read method as read may not be neglected by the programmer, deliberately or undeliberately. This is possible in JAC, which can prohibit perfectly safe methods from being invoked on read receivers.

Müller and Poetzsch-Heffter [12] defines a readonly construct to enable safe passing of references between protection domains. The protection offered by the readonly is transitive, but the usage of readonly objects is limited since the only methods that may be invoked on readonly objects are functional methods as discussed above.

A disadvantage of this system compared to ours is that the definition of their functional methods are more restrictive than our read methods. A functional method protects the receiver object by not allowing update of any object in the system. The same level of protection, although with the ability to create and modify objects, can be achieved in our system.
5 Benefits and drawbacks

The proposed mode system makes it possible to export a reference to an object rest assured that the exported reference cannot be used to modify the object it references. This can be useful in systems development e.g. when building systems by assembling software components or using class libraries acquired from different vendors, perhaps developed by different programmers. The presence of read references makes it possible for the exporter to control the receiver’s usage of the exported reference. A system such as Conﬁned Types [1] that conﬁnes objects of certain types within certain boundaries could be modiﬁed to use read references instead of copies when exporting an object of a conﬁned type outside of a protection domain. This would avoid the issues of copying (see e.g. [7]) and also enable modelling of structures shared across protection domains.

The presence of a system for read reference should be a beneﬁt to developers of frameworks, class libraries and software components since the mode annotations can be seen as veriﬁed complementary information to the documentation of the system regarding the programmers intentions of exported and imported references. Also, the extension of a reference to also include an access mode enables closer modelling of domains where access rights are naturally present.

The system proposed here is an improvement over other similar proposed read-only constructs in that it automatically partitions all methods into either read or write. Thus, as in e.g. JAC [10], a method is not write by default be it in fact a read method or not. Also, the system veriﬁes that a method is not wrongfully declared: a method that changes its receiver object must be declared write and a method that does not must be declared read. This is an improvement over e.g. Flexible Alias Protection [13] that relies on the programmer to correctly annotate the methods. Correctly annotating all methods manually can be an overwhelming and sometimes even impossible task since the programmer must, for example, determine the runtime binding of each method and also keep track of all aliases to member variables created within a method.

Another improvement over similar systems is the possibility to return write references to objects that are not part of the transitive state from a read method. This is not possible in e.g. Universes [12] which makes their functional methods (their read method counterpart) impossible to use with common programming idioms such as the factory method pattern [4]. With the context annotation on method returns, read methods may return write references when invoked on write references.

The system is completely statically checkable without any need for run-time representation of access rights with no run-time overhead. A program can be checked to behave correctly at compile-time.

On the downside, the system imposes a considerable syntactical overhead since it requires the programmer to annotate every variable, formal parameter, method return and method with a mode qualiﬁer. This syntactic overhead can, however, be somewhat reduced by the use of default annotations thus reducing the number of annotations required to be made by the programmer. Default annotations could also be used to control certain aspects of the system, e.g. by making the default mode of a parameter to methods read, methods becomes free from argument side-effects by default.

Another important issue of the system is the deﬁnition of a read method. Although ﬂexible, since a read method accepts write references as arguments and thus could be supplied an alias to the state of its receiver object via an argument, it does not provide the strong invariants on a method’s receiver object as is offered e.g. by functional methods of [12].

However, the read method provides the same protection as functional methods if all formal parameters are declared as read. Such a method is also more powerful than a functional method since objects may be created and modiﬁed within the method, allowing e.g. read methods to be used in the factory method patterns mentioned above. We choose to allow write references as arguments to read methods since any protection from changes via such references inside a read method would only apply during its execution. Rather, we view the presence of a write reference as a permission to change the state of the object regardless of any existing read references to the same object. Since functional methods may not update any object, functional methods cannot be used to perform in-place updates of arguments, even when it is statically veriﬁable that any possible argument cannot be an alias to the transitive state of the receiver object. Thus, the proposed system can achieve the same level of protection of a method’s receiver as functional methods where desired.

In a system such as Universes [12], one protection domain may not have write references into another. This precludes the possibility of write aliases to an object in one protection domain to be passed as arguments to a read method invoked on the object from some other domain. Thus a Universes system with the read reference system presented here seems to be a fruitful combination.

A formalised type system supporting the mode system in a modiﬁcation of ClassicJava [3] has been presented in earlier work [14] at the 4th Workshop for Formal Techniques for Java Programs in conjunction with ECOOP 2001. That system also included a fourth mode, any, which allowed a reference’s run-time access mode to be dependent upon run-time conditions, such as e.g. user input. Due to space limits, this mode has been left out of this paper since it must be accompanied by run-time access modes and dynamic checks increasing the overall
complexity of the system.

6 Conclusions and future work

We have proposed an access mode system for references in class-based, object-oriented, statically-typed programming languages. The system is completely statically checkable and has no need for run-time representation of modes.

The system enables increased alias control by limiting the effects of aliasing by prohibiting state changing methods from being invoked from certain references. Using read references, subobjects of compound objects may be safely exported outside of the compound without risking the exported objects being modified via the exported reference. The access mode constructs offer the same level of protection as related constructs but do not impose as heavy restrictions on a programming language. The system can be used to implement side-effect free methods and to eliminate argument side-effects.

Future work include the implementation of a mode checker for the system and proving the read reference invariants.

References


