Technical Report

UCAM-CL-TR-836
ISSN 1476-2986

Number 836

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Aliasing contracts: a dynamic approach to alias protection

Janina Voigt, Alan Mycroft

June 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Aliasing contracts: a dynamic approach to alias protection

Janina Voigt and Alan Mycroft
Firstname.Lastname@cl.cam.ac.uk

Abstract

Object-oriented programming languages allow multiple variables to refer to the same object, a situation known as aliasing. Aliasing is a powerful tool which enables sharing of objects across a system. However, it can cause serious encapsulation breaches if not controlled properly; through aliasing, internal parts of aggregate objects can be exposed and potentially modified by any part of the system.

A number of schemes for controlling aliasing have been proposed, including Clarke et al.'s ownership types and Boyland et al.'s capabilities. However, many existing systems lack flexibility and expressiveness, making it difficult in practice to program common idioms or patterns which rely on sharing, such as iterators.

We introduce *aliasing contracts*, a dynamic alias protection scheme which is highly flexible and expressive. Aliasing contracts allow developers to express assumptions about which parts of a system can access particular objects. Aliasing contracts attempt to be a universal approach to alias protection; they can be used to encode various existing schemes.

1 Introduction

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, allowing objects to be shared by different parts of a program and enabling the efficient implementation of many important idioms, such as iterators. However, it also causes serious problems in software development since it reduces modularity and encapsulation. An aliased object can be changed through an aliasing reference by a seemingly unrelated part of the system, making programs difficult to understand and bugs hard to trace. As Hoare put it as early as the 1970s, "References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover" [14].

To see how easily aliasing problems can occur, consider the example of a standard `LinkedList`, made up of `Node` objects. `LinkedList` holds a reference (which may be null) to the `head Node` of the list. Each `Node` holds a reference to the `next Node` in the list and also stores a piece of `data`. `LinkedList` supports adding a new `Node` to the list

and searching for a `Node` in the list given the piece of `data` it contains:

```
class LinkedList
  private Node head;

  public void addItem(Object data)
    Node newNode = new Node(data);
    ...


  public Node search(Object data)
    Node n = head;
    while(n != null)
      ...
      n = n.getNext();
```

```
class Node
  private Node next;
  private Object data;

  public Node getNext()
    return next;
```

`Node` provides a getter method for its `next` field. This is required by the `LinkedList` class in order to loop over all `Nodes` in the list. However, this method returns an *alias* to the `next` field. By calling `getNext`, any part of the system can thus obtain a direct reference to the `next Node`, allowing it to access and modify the `next Node` without the knowledge of the containing `Node`. As a result, the `Node` stored in `next` is no longer encapsulated.

It is important to note here that making the `next` field `private` is not sufficient to protect it from exposure. Many modern OO programming languages provide *access modifiers* such as `private`, `protected` and `public`. However, these modifiers protect only the *variable* and not the *object* to which the variable points. They limit the scope of the variable and in this way statically restrict which parts of the system can read or write it but have no impact on accesses to the object itself, which may come through other aliasing references.

A number of alias protection schemes have been proposed to protect the object rather than just the variable from unwanted accesses; we give an overview of such existing schemes in Section 2. However, many of them are too inflexible and restrictive to implement common idioms such as iterators; as a result, they have not yet been widely adopted by the programming community.

In this paper, we propose a dynamic alias protection scheme called *aliasing contracts* which aims to be flexible and expressive, while at the same time remaining conceptually simple. Aliasing contracts allow developers to annotate each variable with assumptions about which parts of the system can access the object to which the variable points. Unlike existing programming-language access modifiers, they do not protect the *variable* but the *object* to which the variable refers.

Aliasing contracts can be used to detect unintended leaking of references as in the example above; we expect that this will be particularly valuable during the testing phase of software development.

We complement aliasing contracts with the concept of *encapsulation groups* which allow objects to be grouped so that read/write access to an object can be given to an entire encapsulation group, rather than single objects. Encapsulation groups are very powerful, since they can be nested; this allows transitive or *deep* contract specifications where access is given to directly as well as indirectly contained objects.

Our dynamic approach to alias control has similar advantages and disadvantages as dynamic type checking. Dynamic approaches are more flexible and can cover conditions which cannot be checked statically. They also tend to be conceptually simpler; static schemes often require more artifacts and features to compensate for the restrictions inherent in static checking. On the other hand, the runtime checks required by dynamic schemes cause performance overheads.

The name *contract* comes from work on software contracts [18] which allow the specification of preconditions and postconditions for methods. Aliasing contracts essentially specify preconditions of object access; they behave like assertions, reporting errors at run time. Aliasing contracts do not restrict the existence of references to an object (that is, any reference to any object is allowed) but they limit which references can be used to access the object. Any violations of aliasing contracts may represent bugs and are reported at run time.

Aliasing contracts are highly expressive and can be used to model other alias protection schemes, as we show in Section 5. They can be used to encode and compare existing alias protection schemes, providing a unifying model of alias protection.

The remainder of the paper is structured as follows: Section 2 gives an overview of existing alias protection schemes. Section 3 introduces aliasing contracts; Section 4 gives a formal syntax and semantics for aliasing contracts. Section 5 compares aliasing contracts with existing alias protection schemes. Section 6 discusses some interesting points of aliasing contracts before we present conclusions and discuss possible future work in Section 7.

2 Background

The literature on aliasing and its control is huge; see [7] for a detailed decription. Here, we summarise the main strands of research for completeness.

The overall idea of alias control is to construct software engineering "design patterns"—or more formal programming language structures—to discipline programmer use of aliasing; a key concept is *encapsulation* whereby usage of some objects ("rep" objects) is restricted to certain other objects (the "owners"). Early conceptual designs include Hogg's islands [15] and Almeida's balloons [2]. They implement *full encapsulation*: each object is either encapsulated within another object or shared; any object reachable through an encapsulated object is also encapsulated. To increase flexibility (though at the cost of soundness) Hogg and Almeida restrict only pointers from fields of other objects ("static aliasing") but allow pointers from local variables and method parameters ("dynamic aliasing")—the latter being more transient and easier to track.

Clarke-style ownership types [8] added significantly to the subject area by showing a type-like system could capture aliasing restrictions (later known as *owners-as-dominators*) and type soundness meant perfect alias control in this discipline (no runtime checks are needed).

Clarke-style ownership types, however, are too inflexible to deal with iterators and other common programming idioms. They require each object to have a *single* owner and also do not allow ownership of an object to be transferred at runtime. These shortcomings were partially addressed by follow-up work, including work on multiple ownership types [17], work by Boyapati et al. [3] on ownership with inner classes, gradual ownership types [21], and Universe types (*owners-as-modifiers*) [19].

There has also been some work on dynamic ownership. Gordon et al. [10] propose a system where ownership information is checked at runtime. Like our dynamic aliasing contracts, dynamic ownership types do not directly restrict aliasing itself, but allow any references to exist; instead, they limit how these references can be used. Gorden-style dynamic ownership types differ from our work since they support only one particular aliasing policy (owners-as-dominators), while aliasing contracts support many different ones.

Another approach to alias protection is that of capabilities [5, 11] and permissions [4, 23]. Capabilities and permissions associate access rights with each object reference, specifying whether the reference is allowed to, for example, read, write or check the identity of an object. This can be used to model various aliasing conditions, such as uniqueness and borrowing.

Throughout the vast literature on alias protection schemes, there is no unifying framework which can be used to embed and compare them. Boyland et al.'s [5] capabilities do this to some extent, but at a relatively low level where there is a large semantic gap to be bridged between them and high-level constructs like owners-as-dominators. With aliasing contracts we build a language-level framework with primitives relating to existing notions of ownership in the same way that dynamic types relate to static types. We exemplify the embedding of existing systems in our framework in Section 5.

3 Aliasing contracts

This work proposes aliasing contracts which express and enforce restrictions about the circumstances under which an object can be accessed. An aliasing contract consists of two boolean expressions e_r and e_w attached to a variable declaration. (Note that in this paper we use the term *variable* to refer to fields, local variables and method parameters.) Access to an object is allowed only if *all* of the contracts of variables currently pointing to the object are satisfied; contracts thus essentially sely prior to contract evaluation and so, for a given contract, may vary from one evaluation to the next. Thus, an alternative view of a contract is a method which takes an object parameter (`accessor`) and returns a boolean value. pecify *preconditions* for object accesses. The expression e_r specifies preconditions for *read* accesses, while e_w concerns *write* accesses.

The distinction between read and write accesses requires a similar distinction between *pure* and *impure* methods: pure methods do not modify any state and may be called with read access permissions, while impure methods require both read and write access permissions. Where the read and write contract expressions are the same, one can be omitted; we call such a contract a *rw-contract* (read-write-contract).

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

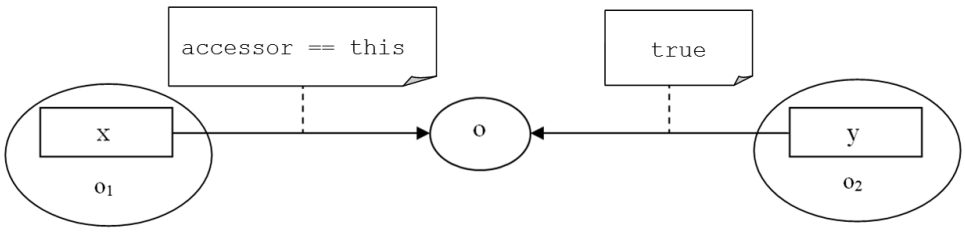

Figure 1: A simple aliasing contract example

Alternatively, we can view a contract as a boolean method of the enclosing object.

Contracts also have access to two special variables: `accessor`, which points to the object requesting access to an object, and `accessed`, which points to the accessed object. (When the access comes from a method, `accessor` points to that method's `this` object.) The value of `accessor` is determined immediat Figure 1 shows a scenario where two variables, `x` and `y`, point to the same object `o`. Both `x` and `y` define an aliasing contract for `o`. The contract on `x`, the rw-contract "`accessor == this`", means that `x` expects its own object, o_1, will be the only object to ever access `o`. The contract is evaluated in the context of the declaring object o_1 and will only evaluate to `true` if `accessor` is equal to `this`; that is, if the access comes from o_1. The contract on `y`, on the other hand, is the rw-contract "`true`". This contract always evaluates to `true`; thus, `y` places no restrictions on who can access `o`.

Looking at the contracts in our example, we can see that accesses to `o` from o_1 are valid (as they do not violate either of the contracts), while accesses to `o` from o_2 are illegal, since they violate the contract specified on `x`. Of course, it is possible for two variables to declare conflicting contracts. For example, if `x` and `y` both declared the rw-contract "`accessor == this`", neither of them would be allowed to access `o`, as this would violate the contract on the other variable.

Aliasing contracts are very flexible since their evaluation depends on the current state and aliasing structure of the program. If `x` is reassigned to point to another object, accesses to `o` from o_2 become valid.

This example also shows that aliasing contracts do not restrict aliasing itself but only accesses to objects. It is legal for any object to hold a reference to `o`; however, this reference can only be used when all aliasing contracts are satisfied.

3.1 Basic contracts

Using our system, we can express a variety of different aliasing contracts. Here, we present them informally; formal syntax and semantics are given in Section 4.

A contract is any valid *side-effect-free* boolean condition. It can use the special variable `accessor` to refer to the object making the access and `accessed` to refer to the accessed object. In addition, we provide the special operator `createdby` for use in contracts which checks if one object was created by another object: `obj1 createdby obj2` evaluates to `true` only if `obj2` created `obj1`.

An object can have multiple creators: if object `a` directly creates object `b` in one of its methods then `b createdby a` evaluates to `true`; if object `a` calls a method of object

`b` which in turn creates object `c` then `c createdby a` is also `true`. (Note, however, that `a createdby b` and `b createdby c` does not necessarily imply `a createdby c`.) Thus, whenever a new object is created, all of the objects currently executing methods on the stack become the new object's creators. This behaviour is important in cases where the creating object may not be directly known, for example when using the Abstract Factory design pattern [9].

We can use aliasing contracts to better encapsulate the `next` field of the `Node` class from the example in Section 1. Ideally, we want to limit access to a `Node`'s `next` field to the `Node` itself and the `LinkedList` that contains it.

By giving `next` the rw-contract "`accessor == accessed || accessor == this || accessor instanceof LinkedList`", we forbid any accesses, except those coming from itself, the `Node` containing it and any `LinkedList` objects. Other parts of the system could still obtain a reference to `next` using the `getNext` method, but would be unable to use it. However, this contract is not ideal, since it allows *any* `LinkedList` object to access a `Node`'s `next` field, not just the `LinkedList` to which the `Node` belongs.

We can use the `createdby` operator to further increase the encapsulation of the `next` field of the `Node` class. We note that a `LinkedList` creates its own `Node` objects in the `addItem` method. Thus, with the rw-contract "`accessor == accessed || accessor == this || (accessor instanceof LinkedList && this createdby accessor)`", only the `LinkedList` that created the `Node` can access its `next` field.

In Section 3.2 and Section 3.3 we introduce contract parameters and encapsulation groups which enable the definition of richer contracts.

We also define two binary boolean operators, `canread` and `canwrite`, which can be used to check if one object has access to another one. This allows testing for access validity before the access is made, for example in an `if`-statement.

3.2 Contract parameters

In some cases, the object holding a reference is not the same as the object wanting to specify a contract. In our `LinkedList` example, the `Nodes` should be encapsulated inside the `LinkedList`, rather than inside the previous `Node` of the `LinkedList`; however, the `LinkedList` does not hold a reference to most of the `Nodes` in the list, only the `head Node`, so cannot directly specify a contract.

To solve this problem, we introduce *contract parameters*. These are dynamic analogues of *ownership contexts* in Clarke et al.'s ownership types [8].

A class can be declared to take contract parameters, which it can use in its contract specifications. When an instance of the class is instantiated, values for the contract parameters must also be provided; we say that the contract parameters are *instantiated*. We can use contract parameters to specify the contracts in our `LinkedList` example:

```
class LinkedList
  Node head accessor == this
      || accessor == accessed ;

  public impure void addItem(Object o)
    Node n = new Node<(accessor == this
        || accessor == accessed)>(o);
    ...

  ...
```

```
class Node<cp>
  Node next cp ;
  ...
```

The `LinkedList` class specifies the contract for `head` as usual: the rw-contract "`accessor == this || accessor == accessed`" ensures that `head` can be read and written only by itself and the `LinkedList`. The `Node` class takes a contract parameter, `cp`, which it uses to specify the contract for `next`. When `LinkedList` creates a `Node`, it binds `cp` to the rw-contract "`accessor == this || accessor == accessed`". This contract is then used as the contract for `next` in `Node`.

In the presence of contract parameters, a contract's declaring object is no longer the nearest enclosing object but the object which provides the instantiated values for the contract parameters. Thus, when the contract for `next` in `Node` is evaluated, `this` does not point to the enclosing `Node` object but to the `LinkedList` object. The effect is that `Node` objects stored in `next` are fully encapsulated in the enclosing `LinkedList` object.

3.3 Encapsulation groups

Although the contracts presented above allow the definition of a wide variety of aliasing conditions, they are all one-level contracts; we need support for transitive contracts.

Above, we specified contracts for the `LinkedList` example using contract parameters, which encapsulated all `Node` objects inside their `LinkedList`; only the `LinkedList` had access to all the `Nodes`. In order to get more flexibility, we may want to allow any `Nodes` to access each other within a single `LinkedList`.

To make this possible, we introduce *encapsulation groups*, which allow us to group together several objects and give access to them. The power of encapsulation groups lies in the fact that they can contain an unlimited and varying number of objects, making it possible to refer to many instead of just one object in a contract.

Semantically, an encapsulation group g is a (possibly infinite) set of access paths which can be used in expressions E `in` $E'.g$ to test whether the object resulting from E is a member of those objects reachable from a path in g starting from object E'. Syntactically, groups are specified by a regular grammar (see *groupDef* in the formal syntax). Note that paths specified by groups are syntactically specified to be rooted in `this`, but $E'.g$ is evaluated at run-time starting from the object resulting from E'.

Like contracts, encapsulation groups are evaluated at run time in the context of the nearest enclosing object. The expressions in the group must thus be valid in the context of that object.

An encapsulation group g_1 may contain the same or another encapsulation group g_2; in

that case, the contents of the g_2 are added to g_1. Note that `this` in groups effectively plays the role of empty path, so that in the example below `next.nextNodes` could equivalently be written `this.next.nextNodes`.

We can use encapsulation groups to achieve the desired encapsulation for our `LinkedList` example, giving both the `LinkedList` and all `Nodes` in the list access to every `Node`. First, we create an encapsulation group `nextNodes` in `Node` that will transitively contain all *following* `Nodes` in the list; then we declare an encapsulation group called `allNodes` in `LinkedList` which contains *all* `Nodes` in the list:

```
group nextNodes = {this, next.nextNodes};       //In Node
group allNodes = {head, head.nextNodes};        //In LinkedList
```

Now that we have an encapsulation group containing all `Nodes` in our list, we can use the '`in`' operator in our contracts to check if a given object is inside a particular encapsulation group. In our example, we can replace the contract parameter "`accessor == this || accessor == accessed`" from the example in the previous section with the parameter "`accessor == this || accessor in this.allNodes`". This contract checks if the accessing object is either the `LinkedList` itself or a `Node` in the list.

Encapsulation groups have simple semantics but naturally result in cycles which complicates the implementation of the `in` operator; it must ensure that cycles are evaluated only once and do not lead to indefinite looping. We anticipate that many uses of groups in '`in`' expressions can be checked statically; this paper introduces them into aliasing contracts as a unifying formalism rather than an efficient implementation technique.

3.4 Checking aliasing contracts

Since aliasing contracts depend on the dynamic aliasing structure of a program at the time an access to an object is made, they cannot generally be checked statically. Instead, the semantics we present here checks them at run time, before an access to an object is allowed to proceed; if a contract violation is detected, an error is reported.

Any accesses to objects within contracts will themselves trigger contract evaluations. This ensures that objects are fully protected, even from contracts themselves. This design choice can, of course, lead to cyclic and infinite contract evaluation if not enough care is taken by developers writing their contracts. The simple contract below, for example, leads to cyclic contract evaluation when `s` is dereferenced:

```
String s {s.length() > 0};
```

The contract states that `String s` can only be read and written when its length is larger than zero. However, the contract itself requires a read access to `s` and will lead to another evaluation of `s`'s contract, which in turn will cause another contract evaluation and so on; we have a cyclic contract evaluation which never terminates. Our semantics simply loops by recursively evaluating the contract; implementations of aliasing contracts may prefer to check for such cycles.

4 Formalising aliasing contracts

In this section, we formalise aliasing contracts. We first give a syntax for aliasing contracts, before presenting the operational semantics as a set of reduction rules.

4.1 Syntax

The syntax for aliasing contracts is shown below. In this syntax, C ranges over class names, m over method names, f over field names, g over encapsulation group names, x and y over variable names and cp over contract parameter names. τ denotes types, either a class type or `bool`. For brevity, we write, for example, $\overline{fieldDef}$ for $fieldDef_1 \ldots fieldDef_n$.

$$
\begin{array}{rcl}
program & = & \overline{classDef} \\
classDef & = & \texttt{class}\ C\ \ \overline{cp}\ \ \texttt{extends}\ C\ \ \ \overline{fieldDef}\ \overline{groupDef}\ \overline{constrDef}\ \overline{methodDef} \\
\tau & = & C\ \ \texttt{bool} \\
fieldDef & = & \tau\ f\ \ \ contractDef\ \ ; \\
groupDef & = & \texttt{group}\ g\ =\ \ \overline{groupMember} \\
groupMember & = & e\ \ \ e.g \\
constrDef & = & C\ (\ \overline{argDef}\ )\ \ \ \overline{varDef}\ E \\
methodDef & = & (\ \texttt{pure}\ \ \texttt{impure}\ )\ \tau\ m\ (\ \overline{argDef}\ )\ \ \ \overline{varDef}\ E \\
argDef & = & \tau\ x\ \ \ contractDef \\
varDef & = & \tau\ x\ \ \ contractDef\ \ ; \\
contractDef & = & e, e\ \ \ cp \\
E & = & x\ \ \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E\ \ \ E.f\ \ \ E.f\ =\ E\ \ \ E.m\ (\ \overline{E}\ )\ \ \ x\ =\ E \\
 & & \texttt{new}\ C\ \ \overline{contractDef}\ \ \ (\ \overline{E}\ )\ \ E; E\ \ \ E\ binOp\ E\ \ \ E\ \texttt{instanceof}\ C \\
 & & E\ \texttt{in}\ E.g\ \ \ !E\ \ \ E\ \&\&\ E\ \ E\ \ \ E\ \ \texttt{this}\ \ \texttt{accessor}\ \ \texttt{accessed} \\
 & & \texttt{true}\ \ \texttt{false}\ \ \texttt{null} \\
e & = & \text{subset of } E \text{ (see below)} \\
binOp & = & \texttt{createdby}\ \ \texttt{canread}\ \ \texttt{canwrite}\ \ \ ==
\end{array}
$$

A program consists of a number of class definitions, each of which contains field, group, constructor and method definitions. Each class may take a number of contract parameters. Methods and constructors both take arguments and contain definitions of local variables along with the method body expression. Constructors additionally require instantiations for the contract parameters declared by their class. Unlike constructors, methods have a return type and must be declared either `pure` or `impure`.

Contracts may be defined for any kind of variable: fields, method parameters and local variables. A contract is made up of two contract expressions; alternatively, it can consist of one of the contract parameters declared by the enclosing class. The use of a contract parameter is marked with angle brackets to clearly distinguish it from standard contracts.

Note that we distinguish here between e and E. We use e for expressions in contracts and groups. Expressions e used both in contracts and in encapsulation groups must be valid in the context (scope) of the nearest enclosing object (the declaring object). Therefore, they cannot refer to method parameters and local variables (but can refer to `this`). We distinguish them from standard expressions E to highlight this difference.

The keywords `this`, `accessor` and `accessed` behave exactly like user-defined variables, but do not themselves have contracts, nor may they be updated. The variable `this` is available in every execution context and points to the current object; `accessor` and `accessed` are logically λ-bound only within aliasing contracts.

We impose several additional syntactic restrictions on programs:

The two contract expressions must be of type boolean.

The keywords `accessor` and `accessed` and the operators `createdby` and `in` can be used only in contracts.

The expressions used in contracts cannot have side effects. Variable assignments and calls to impure methods cannot be used in this context.

The scope and lifetime of variables is defined as follows:-

The lifetime of a field is that of the object which contains it. In our current semantics objects live forever and thus their fields are also never deallocated. Garbage collection would be transparent, while explicit deallocation (with nullification of all references to the deallocated object) may cause other contracts to succeed, but never cause any contract to fail, because of conjunctive treatment of multiple contracts.

Local variables and method parameters exist only while the associated method executes. They are deallocated from the stack once the method returns and are not accessible outside the scope of the method. On deallocation, their contracts are removed from the objects to which they point.

4.2 Reduction rules

Figures 2, 3 and 4 present the reduction rules of our operational semantics. In the following sections, we first introduce necessary notation for our reduction rules, before explaining the rules in more detail.

We have omitted some standard reduction rules here in the interest of conciseness; these include rules for sequences and conditional statements, as well as rules for common boolean operators such as `instanceof`, `!`, `==`, && and `||`. We have also omitted context rules which express our assumed left-to-right evaluation strategy when multiple subexpressions occur within an expression.

4.2.1 Notation

We define our operational semantics in the context of the current machine state consisting of a *stack* Δ, a *heap* Ψ, a *contract store* Λ, which tracks the aliasing contracts for each object, and a *creation store* Ω, which records the creators of each object. We use $\rightsquigarrow$ to denote reduction.

The stack Δ stores the contents of local variables. Objects are stored on the heap Ψ; for a local variable which points to an object on the heap, the stack Δ records the address ι of the object on the heap or the special value `null`. For local variables of the primitive type `bool`, the stack directly stores the value, either `true` or `false`.

For convenience, we define function $init(\tau)$ which gives the initial value for a variable of type τ: `false` for `bool` and `null` otherwise.

A boolean value b may be either `true` or `false`. An address a may be a valid heap address ι or the null-address `null`. A value v is anything that can be stored in a variable: a boolean b or a heap address a.

We use ϕ to refer to *syntactic contracts*, which can either consist of two contract expressions or a contract parameter. To evaluate a contract, we need to know the address ι of the contract's declaring object; during contract evaluation, `this` will be bound to ι.

By providing an address ι, we transform a syntactic contract into a *contract closure*, which we denote ψ. Note that contract closures bind `this` but leave `accessor` and `accessed` unbound; a binding for `accessor` and `accessed` is given only just before a contract is evaluated.

In order to simplify the lookup of field, method, contract and contract parameter information in our operational semantics, we define the compile-time dictionaries , , and . The information in these dictionaries can easily be determined from the source of a program.

For each class C, ^{C} maps each field name f onto the field's type τ: $^C = f \tau$. Similarly, for each class C, ^{C} is a mapping from method m to a tuple of the method's parameters y and their types σ, the method body E, the method's local variables x and their types τ, and the method's purity π (`true` or `false`): $^C = m y \sigma, E, x \tau, \pi$. We often use ':' instead of ' ' for types, thus $x : \tau$

Note that we treat constructors as special kinds of methods. They can be looked up in ^{C} under the method name C. Constructors do not require any contract checking, since the object to be constructed does not yet have any associated aliasing contracts. Therefore, it is not necessary to distinguish between pure and impure constructors and thus no purity is recorded for constructors in ^{C}.

For class C, ^{C} gives the list of its formal contract parameters: $cp_1, \ldots, cp_n$.

For each variable (field, parameter or local variable) x in class C, ^{C} contains the contract associated with the variable. (For simplicity, we assume here that each variable name is globally unique.) A contract can either contain two contract expressions e_r and e_w or consist of a contract parameter cp; if the contract contains two contract expressions, ^{C} stores them; in this case $^C(x) = e_r, e_w$. When a contract parameter is used, ^{C} records the name of the contract parameter surrounded by angle brackets: $^C(x) = cp$.

Now we turn to the definition of run-time values. Objects o are stored on the heap Ψ. In our semantics, we do not refer to objects directly, but instead uniquely identify an object using its heap address ι; there is a one-to-one correspondence between objects and heap addresses.

For each object o we record its type τ and a dictionary mapping each field name f to its current value v. For each object, we also store metadata about its contract parameters, recording the actual values that the contract parameters were given when the object was created. These values are not visible or accessible to the programmer and are fixed for the lifetime of the object. We record them in a dictionary ρ, with each contract parameter name cp mapping to the heap address ι' of the object which instantiated the contract parameter and the read and write contract expressions e_r and e_w; thus, $o = \tau, f v, cp \iota', e_r, e_w$. Given a syntactic contract ϕ (e_r, e_w or cp), an object ι and dictionary ρ its *contract closure* ψ is given by $closure(\phi, \iota, \rho)$ defined by:

$$\begin{aligned} closure(\ e_r, e_w\ , \iota, \rho) &= \iota, e_r, e_w \\ closure(\ cp\ , \iota, \rho) &= \rho(cp) \end{aligned}$$

We use $_i$ to select the ith element of a tuple; thus $(x_1, \ldots, x_n)_i = x_i$.

We use square brackets to indicate the update of a mapping. For example, to update the value of a field f of object o, we write $o_2[f v]$. We define update as usual: $o_2[f v](f) = v$ and $o_2[f v](f') = o_2(f')$ if $f' = f$.

The stack Δ is a list of stack frames, where each stack frame δ is a partial mapping of variable names to current values. The operator denotes the pushing of a stack frame δ onto the top of the stack. Thus, $\Delta = \delta_1 \ \ldots \ \delta_n$ where δ_n is the top (most recent) stack frame.

Given stack frame δ and variable x, $\delta(x)$ gives the current value of x (provided $(x \ v) \ \delta$) and $\delta[x \ v]$ gives a new stack frame with x updated to v.

Note that since we do not allow inner classes or nested methods and functions, local variables and parameters can always be found in the top frame of the stack. Thus, variable lookups and updates for a stack Δ are defined as $\Delta(x) = \delta(x)$ and $\Delta[x \ v] = \Delta_1 \ \delta[x \ v]$ where $\Delta = \Delta_1 \ \delta$.

We define the heap Ψ as a partial function mapping addresses ι to objects o.

While the program is executing, we need to track which aliasing contracts apply to which objects. This information changes with the aliasing structure of the program. We define a contract store Λ, a partial function which, for each object at heap address ι, records the set S of aliasing contracts which currently apply to the object. S is a set of tuples, each containing the address ι' of contract's declaring object and the read and write contract expressions e_r and e_w: $\Lambda(\iota)$ is a set of triples ι', e_r, e_w .

Parameters and local variables exist only while the method which defines them executes; their contracts must therefore also be removed from the contract store when the method returns. To facilitate this, we define our contract store as a stack of frames λ_0 to λ_n: $\Lambda = \lambda_0 \ \ldots \ \lambda_n$.

Frame λ_0 holds the contracts associated with fields; these are not bound to the execution of any method but persist throughout program execution. Every time a method is called, we add a new stack frame, λ_n, which holds the contracts declared for any local variables and method parameters. This frame is removed when the method returns.

To get the set of all aliasing contracts for an object at heap address ι, we must perform a lookup in every frame of the contract store stack. Thus, $\Lambda(\iota) = \lambda_0(\iota) \ \lambda_n(\iota)$ where $\Lambda = \lambda_0 \ \ldots \ \lambda_n$. Whenever a field is pointed to a new object, we update the information in frame λ_0 to reflect this; assignments of objects to local variables and method parameters require updating the information in the top frame, λ_n.

At first sight, Δ and Λ appear very similar. However, the semantics of lookups differ significantly: a lookup in Δ requires a lookup only in the top frame of the stack; each new stack frame overrides the information in the lower frames. A lookup in Λ, on the other hand, requires a lookup in all frames.

During program execution, we also need to keep track of which objects create which other objects. This information is required by the `createdby` operator. The creation store Ω records this information. It is a partial function mapping an object's heap address ι to the set of addresses of the object's creators. Equivalently, we can think of Ω as containing a set of pairs of object addresses, where each pair maps an object's address to the address of one of its creators. (As we noted in Section 3.1, an object may have multiple creators; all objects executing methods on the stack at the point of an object's creation are recorded as that object's creators.)

The `in` operator needs to evaluate all paths in an encapsulation group g with respect to a current heap Ψ and a starting address ι. We say that it calculates the *g-reachable set* of the object at address ι.

In our semantics, we denote $G^*(\iota, g, \Psi)$ the g-reachable set in the context of the object

at address ι. An encapsulation group contains paths which at run time evaluate to objects reached by following these paths using Ψ and starting from ι. If other encapsulation groups appear within g then these are recursively followed—allowing specification of any regular set of paths.

4.2.2 Contract evaluation

Contracts are declared on variables (fields, parameters and local variables) in the code and apply to the objects to which the variables point at run time. Whenever an object on the heap is accessed in some way, all contracts associated with it must be evaluated. Only if they are all satisfied is the access to the object allowed to proceed.

We distinguish between read and write accesses:

Field accesses (*field-get*) (see Figures 2, 3 and 4) and calls to pure methods (*method-call-pure*) constitute read accesses.

Field updates are write accesses (*field-put*).

Calls to impure methods (*method-call-impure*) constitute both read *and* write accesses.

The operators `canread` (*canread*) and `canwrite` (*canwrite*) check the read and write contracts respectively.

Note that accesses and updates of local variables do not require contract evaluation, as shown in (*local-get*). Accessing and updating local variables requires accesses and modifications to the unaliased *stack* only; no objects on the *heap* are accessed or updated and thus no aliasing contracts need to be checked. Similarly, constructor calls (new) do not change existing objects on the heap and therefore also do not require contract checks.

In order to describe the execution order of contract evaluations, we introduce three auxiliary constructs which do not explicitly appear in source programs: *assert*, and *eval*.

(canread)

$$\frac{}{\iota \texttt{ canread } \iota', \Delta, \Psi, \Lambda, \Omega \rightsquigarrow eval(\iota_1, \iota, \iota', e_{r1}) \&\& \ldots \&\& eval(\iota_n, \iota, \iota', e_{rn}), \Delta, \Psi, \Lambda, \Omega}$$
$$\text{where } \iota_1, e_{r1}, e_{w1}, \ldots, \iota_n, e_{rn}, e_{wn} = \Lambda(\iota')$$

(canwrite)

$$\frac{}{\iota \texttt{ canwrite } \iota', \Delta, \Psi, \Lambda, \Omega \rightsquigarrow eval(\iota_1, \iota, \iota', e_{w1}) \&\& \ldots \&\& eval(\iota_n, \iota, \iota', e_{wn}), \Delta, \Psi, \Lambda, \Omega}$$
$$\text{where } \iota_1, e_{r1}, e_{w1}, \ldots, \iota_n, e_{rn}, e_{wn} = \Lambda(\iota')$$

(eval-contract)

$$\frac{e, \Delta \quad \delta, \Psi, \Lambda, \Omega \rightsquigarrow e', \Delta \quad \delta, \Psi, \Lambda, \Omega}{eval(\iota, \iota', \iota'', e), \Delta, \Psi, \Lambda, \Omega \rightsquigarrow eval(\iota, \iota', \iota'', e'), \Delta, \Psi, \Lambda, \Omega}$$
$$\text{where } \delta = \texttt{this } \iota, \texttt{accessor } \iota', \texttt{accessed } \iota''$$

Figure 2: Operational Semantics for Aliasing Contracts

(eval-done)

$$\frac{}{eval(\iota, \iota', \iota'', b), \Delta, \Psi, \Lambda, \Omega \rightsquigarrow b, \Delta, \Psi, \Lambda, \Omega}$$

(field-get)

$$\frac{}{\iota.f, \Delta, \Psi, \Lambda, \Omega \rightsquigarrow assert(\texttt{this canread } \iota); \iota \quad f, \Delta, \Psi, \Lambda, \Omega}$$

(field-get-)

$$\frac{}{\iota \quad f, \Delta, \Psi, \Lambda, \Omega \rightsquigarrow v, \Delta, \Psi, \Lambda, \Omega}$$
where $v = \Psi(\iota) \quad _2 (f)$

(field-put)

$$\frac{}{\iota.f = v, \Delta, \Psi, \Lambda, \Omega \rightsquigarrow assert(\texttt{this canwrite } \iota); \iota \quad f = v, \Delta, \Psi, \Lambda, \Omega}$$

(field-put- -bool)

$$\frac{}{\iota \quad f = b, \Delta, \Psi, \Lambda, \Omega \rightsquigarrow b, \Delta, \Psi \ ', \Lambda, \Omega}$$
where $\Psi' = \Psi[\iota \quad \Psi(\iota) \quad _2 [f \quad b]]$

(field-put-)

$$\frac{}{\iota \quad f = a, \Delta, \Psi, \lambda \ _0 \quad \Lambda, \Omega \rightsquigarrow a, \Delta, \Psi \ ', \lambda'_0 \quad \Lambda, \Omega}$$
where $\Psi' = \Psi[\iota \quad \Psi(\iota) \quad _2 [f \quad a]]$
and $a' = \Psi(\iota) \quad _2 (f)$
and $\tau = \Psi(\iota) \quad _1$
and $\psi = closure(C^\tau(f), \iota, \Psi(\iota) \quad _3)$
and $\lambda'_0 = \lambda_0[a' \quad \lambda \ _0(a') \quad \psi \ ]$ (if $a \quad ' = \texttt{null}$) $[a \quad \lambda_0(a) \quad \psi \ ]$ (if $a \ = \texttt{null}$)

(local-get)

$$\frac{}{x, \Delta, \Psi, \Lambda, \Omega \rightsquigarrow \Delta(x), \Delta, \Psi, \Lambda, \Omega}$$

(local-put-bool)

$$\frac{}{x = b, \Delta, \Psi, \Lambda, \Omega \rightsquigarrow b, \Delta[x \quad b], \Psi, \Lambda, \Omega}$$

(local-put)

$$\frac{}{x = a, \Delta, \Psi, \Lambda \quad \lambda_n, \Omega \rightsquigarrow a, \Delta[x \quad a], \Psi, \Lambda \quad \lambda \ '_n, \Omega}$$
where $a' = \Delta(x)$
and $\iota = \Delta(\texttt{this})$
and $\tau = \Psi(\iota) \quad _1$
and $\psi = closure(C^\tau(x), \iota, \Psi(\iota) \quad _3)$
and $\lambda'_n = \lambda_n[a' \quad \lambda \ _n(a') \quad \psi \ ]$ (if $a \quad ' = \texttt{null}$) $[a \quad \lambda_n(a) \quad \psi \ ]$ (if $a \ = \texttt{null}$)

Figure 3: Operational Semantics for Aliasing Contracts

(method-call-pure)

$$\frac{}{\iota.m(v_1,\ldots,v_n),\Delta,\Psi,\Lambda,\Omega \rightsquigarrow \mathit{assert}(\texttt{this canread } \iota);\ \iota\ \ m(v\ _1,\ldots,v_n),\Delta,\Psi,\Lambda,\Omega}$$

where $\tau = \Psi(\iota)\ \ _1$ and $\ \ ^{\tau}(m) = (\_,\_,\_,\texttt{true})$

(method-call-impure)

$$\frac{}{\iota.m(v_1,\ldots,v_n),\Delta,\Psi,\Lambda,\Omega \rightsquigarrow \mathit{assert}(\texttt{this canread } \iota \ \&\&\ \texttt{this canwrite } \iota);\ \iota\ \ m(v\ _1,\ldots,v_n),\Delta,\Psi,\Lambda,\Omega}$$

where $\tau = \Psi(\iota)\ \ _1$ and $\ \ ^{\tau}(m) = (\_,\_,\_,\texttt{false})$

(method-call-)

$$\frac{}{\iota\ \ m(v\ _1,\ldots,v_n),\Delta,\Psi,\Lambda,\Omega \rightsquigarrow E,\Delta\ \ \delta,\Psi,\Lambda\ \ \lambda,\Omega}$$

where $\delta = \texttt{this}\ \ \iota, y\ _1\ \ v\ _1,\ldots,y_n\ \ v\ _n, x_1\ \ \mathit{init}(\tau\ _1),\ldots,x_m\ \ \mathit{init}(\tau\ _m)$

and $\lambda = v\ _i\ \ \psi\ _i\ \ 1\ \ i\ \ n\ \ v\ \ _i\ / \texttt{null},\texttt{true},\texttt{false}$

and $\psi_i = \mathit{closure}(C^{\tau}(y_i),\iota,\Psi(\iota)\ \ _3)$

and $\tau = \Psi(\iota)\ \ _1$

and $\ \ ^{\tau}(m) = (\ y\ _1 : \sigma_1,\ldots,y_n : \sigma_n\ , E,\ x\ _1 : \tau_1,\ldots,x_m : \tau_m\ ,\pi)$

(new)

$$\frac{}{\texttt{new } c\ (\phi_1),\ldots,(\phi_m)\ (v_1,\ldots,v_n),\Delta,\Psi,\Lambda,\Omega \rightsquigarrow E;\iota,\Delta\ \ \delta,\Psi\,',\Lambda\ \ \lambda,\Omega\,'}$$

where ι is new in Ψ

and $\iota' = \Delta(\texttt{this})$

and $\Psi' = \Psi[\iota\ \ \ c,\ f\ \ _1\ \ \ \mathit{init}(\tau\ _1),\ldots,f_k\ \ \mathit{init}(\tau\ _k)\ ,\ cp\ _1\ \ \psi\ _1,\ldots,cp_m\ \ \psi\ _m\ ]$

and $\ \ ^{c} = f\ _1 : \tau_1,\ldots,f_k : \tau_k$

and $\psi_i = \mathit{closure}(\phi_i,\iota',\Psi(\iota')\ \ _3)\ \ (1\ \ i\ \ m)$

and $\ \ ^{c} = cp\ _1,\ldots,cp_m$

and $\delta = \texttt{this}\ \ \iota, y\ _1\ \ v\ _1,\ldots,y_n\ \ v\ _n, x_1\ \ \mathit{init}(\tau\ '_1),\ldots,x_l\ \ \mathit{init}(\tau\ '_l)$

and $\lambda = v\ _i\ \ \psi\ '_i\ \ 1\ \ i\ \ n\ \ v\ \ _i\ / \texttt{null},\texttt{true},\texttt{false}$

and $\psi'_i = \mathit{closure}(C^{c}(y_i),\iota,\Psi(\iota)\ \ _3)\ \ (1\ \ i\ \ n)$

and $\ \ ^{c}(c) = (\ y\ _1 : \sigma_1,\ldots,y_n : \sigma_n\ , E,\ x\ _1 : \tau'_1,\ldots,x_l : \tau'_l\ ,\pi)$

and $\Delta = \delta_1\ \ \ldots\ \ \delta\ _r$

and $\Omega' = \Omega[\iota\ \ \ \delta\ \ _i(\texttt{this})\ \ 1\ \ i\ \ r\ ]$

(created-by)

$$\frac{}{\iota\ \texttt{createdby}\ \iota',\Delta,\Psi,\Lambda,\Omega \rightsquigarrow b,\Delta,\Psi,\Lambda,\Omega}\ \text{where } b = \begin{cases}\texttt{true} & \iota'\ \ \Omega(\iota)\\ \texttt{false} & \iota'\ /\Omega(\iota)\end{cases}$$

(in)

$$\frac{}{\iota\ \texttt{in}\ \iota'.g,\Delta,\Psi,\Lambda,\Omega \rightsquigarrow b,\Delta,\Psi,\Lambda,\Omega}\ \text{where } b = \begin{cases}\texttt{true} & \iota\ \ G^{*}(\iota',g,\Psi)\\ \texttt{false} & \iota\ /G^{*}(\iota',g,\Psi)\end{cases}$$

Figure 4: Operational Semantics for Aliasing Contracts

Given an expression $\iota.fm$ (a field or method access of the object at address ι) which requires contract checks, we transform the expression to

$$assert(\texttt{this canread}\ \iota); \iota\ \ fm \quad \text{or} \quad assert(\texttt{this canwrite}\ \iota); \iota\ \ fm$$

as appropriate—see (*field-get*), (*field-put*), (*method-call-pure*) and (*method-call-impure*). Here ' ' represents object access *without* contract checking.

The `canread` and `canwrite` operators (rules (*canread*) and (*canwrite*)) then reduce to

$$eval(\iota_1, \iota, \iota'', e_1)\ \&\&\ \ldots\ \&\&\ eval(\iota_n, \iota, \iota'', e_n)),$$

where $e_1\ \ldots\ e_n$ are the contracts that currently apply to the object at address ι', $\iota_1\ \ldots\ \iota_n$ are the objects that declared those contracts, ι is the object performing that access (`accessor`) and ι' is the accessed object (`accessed`). Each contract associated with an object is evaluated separately using the *eval* construct which is given (by `canread`/`canwrite`) the address ι of the object whose method is making the access (bound as in a new stack frame δ as `accessor`), the address ι' of the object which statically declared the contract (bound as `this`) and the address ι'' of the accessed object (bound as `accessed`). Execution proceeds (*eval-contract*) until a boolean value is produced (*eval-done*). The *assert* function is conventional—it continues for `true`, but raises an exception for `false`. (An alternative semantic view is that $assert($`false`$)$ is a *stuck state*.)

In our operational semantics, contract evaluation constitutes the very last step before accessing or updating the object; any sub-expressions are reduced first, before contracts are evaluated. For example, for a field update of the form $E_1.f = E_2$, E_1 and E_2 are evaluated first before the contract checks are performed. This means that any side-effects in the expressions E_1 and E_2 could potentially influence the outcome of the contract evaluations.

We note here that if there are multiple contracts to evaluate, the order in which they are evaluated is irrelevant. Expressions in contracts may not have side-effects; this means that Δ, Ψ, Λ and Ω must all be the same before and after the contract evaluation. This implies that contracts cannot declare new variables or contracts, change existing variables or contracts or create new objects.[1] Thus, a contract can change neither the state of the program nor its aliasing structure and therefore cannot affect other contract evaluations.

4.2.3 Contract transfer

Whenever a variable assignment occurs, the contracts in the contract store Λ must be updated: the contracts associated with the variable must be removed from the object to which the variable previously pointed and added to the object to which the variable now points.

Such a contract transfer occurs for field updates and local variable updates, as shown in (*field-put-*) and (*local-put*). Note that (*field-put-*) modifies the contracts in λ_0, the bottom-most frame of the contract store which holds contracts associated with fields;

[1] Of course, this does not mean that Δ, Ψ, Λ and Ω cannot change *during* the evaluation of the contract, as long as they are returned to their original state at the end of the evaluation. This happens, for example, when a contract calls another method, modifying Δ. The other method may declare local variables and contracts but these changes all disappear when the method returns. Incidentally, (*eval-contract*) would cause a stuck state if any of Δ, Ψ, Λ or Ω change.

(*local-put*), on the other hand, performs contract updates in the top frame of the contract store stack, λ_n, which stores the contracts associated with local variables.

When a contract containing a contract parameter needs to be stored in the contract store Λ, we look up the actual contract associated with the contract parameter in the enclosing object's metadata on the heap. This gives us the actual contract expressions associated with the contract parameter, as well as the contract's declaring object. We store this information in Λ, enabling us to evaluate the parameterised contract like any other contract.

Booleans are primitive types which are not subject to aliasing; consequently, updates to boolean fields and local variables do not require updates to the contract store, as shown in (*field-put- -bool*) and (*local-put-bool*).

4.2.4 The `createdby` operator

Our system also tracks information about the creation of objects: for each object, it records which objects created it. The `createdby` operator simply looks up this information in the creation store Ω, as shown in (*createdby*).

As noted earlier, an object may have multiple creators. Thus, when an object is constructed as shown in (*new*), each object which is currently executing a method is regarded as the creator of the new object; we essentially walk the stack, recording the address stored in `this` in each stack frame as one of the creators in Ω.

4.2.5 The `in` operator

The evaluation of `in` relies on the computation of the set of objects reachable from an encapsulation group g. Given the right-hand-side of `in` being $E.g$, we evaluate E to give object ι and then follow all paths in g starting from ι using heap Ψ. G^* is used in rule (*in*).

Calculating the reachable set of an encapsulation group is potentially expensive; in the worst case, it may contain every object in the system. To find the reachable set, one must keep track of which encapsulation groups have been evaluated in order to avoid cycles and infinite evaluation. The reachable set could alternatively be calculated by our machine; however, the need to avoid cyclic evaluation leads to complex reduction rules which have to record which encapsulation groups have already been evaluated; the formalisation given here keeps our reduction rules simple and concise.

4.2.6 Method and constructor calls

Method and constructor calls are performed in very similar ways, as shown in (*method-call-*) and (*new*).

In (*new*) a new object is constructed; (*new*) modifies the heap to create a new object, recording the object's type c, the fields and their associated values (initialised to default values of `false` for boolean or `null` for all other types) and the contract parameters cp_i with their associated contracts ϕ_i. Each contract parameter ϕ_i which is passed in to the constructor can either be a full syntactic contract with two contract expressions or can itself be a contract parameter cp. The function *closure* converts this to a semantic

contract by adding the address ι' to the two contract expressions (in the former case), or looking up *cp* in the second case.

To perform the actual method or constructor call, (*method-call-*) and (*new*) first create a new stack frame δ; for method calls, `this` is bound to the address of the object whose method is called; for constructor calls `this` is bound to the address of the new object. The values of the method's or constructor's parameters $y_1, \ldots, y_n$ are initialised to the values $v_1, \ldots, v_n$ given in the method call. The local variables $x_1, \ldots, x_m$ are initialised to their default values $w_1, \ldots, w_m$, (each `false` or `null`).

Secondly, a new frame λ for the contract store is created to store the contracts for local variables and parameters. Since local variables are initialised to `null` their contracts do not yet need to be added to the contract store. Parameters, on the other hand, may already point to heap objects and thus their contracts must be added to the contract store.

Finally, the new stack frame δ and contract store frame λ are added to the stack Δ and the contract store Λ respectively and the method or constructor body E is reduced in the new context. A constructor call reduces the body expression E and then returns the address ι of `this`.

5 Comparison with existing alias protection schemes

The dynamic nature of aliasing contracts makes them highly flexible and expressive and enables us to use them to model a wide range of encapsulation policies. In this section, we look at how to model borrowing, uniqueness, owners-as-dominators and owners-as-modifiers encapsulation using aliasing contracts.

5.1 Borrowing

Borrowing allows temporary aliasing of an object, for example for the duration of a method call. A borrowed reference may not be stored permanently, for example in a field, and only exists for a limited space of time.

Many systems support some form of borrowing, often through an annotation which may be called `borrowed`, `limited`, `temporary`, `unique`, `unconsumable` or `lent` [5]. AliasJava [1], for example, provides a `lent` annotation. Systems which distinguish between static and dynamic aliasing, including Hogg's islands [15], Almeida's balloons [2] and Clarke et al.'s extension to ownership types [6], implicitly support borrowing by allowing temporary (dynamic) references to protected objects. Read-only references such as those provided by universe types [19] allow only read accesses to object; this can be seen as a different kind of borrowing, which restricts what can be done with a borrowed reference, rather than the amount of time for which a borrowed reference can be used.

Although we do not have a contract in our system which is equivalent to borrowing, we can simulate borrowing behaviour by temporarily reassigning a protected object to a variable with rw-contract "`true`".

In the example below, we want to call method `bar` of another object `foo` and pass it the object pointed to by variable `o` as a parameter. However, `o` has a contract which does not allow accesses to its object from `foo`. (For example, it may have the rw-contract

"`accessor == this || accessor == accessed`".) We therefore create a temporary variable with rw-contract "`true`" and use it to store the object to which `o` points. Once we nullify `o`, the object in question becomes accessible; the strict contract has been removed. We can now call the method `bar` which can use the object. Once the method has finished executing, we store the object back into `o`, protecting it once again. In this way, the object is made available, but only for the duration of the method call.

```
Object o {...};       //o has a contract that makes its object
                      // inaccessible to the method we want to call.
Object temp {true};   //temporary variable with a loose contract
temp = o;             //the object is moved to the temporary variable
o = null;
foo.bar(temp);        //bar can now access the object due to the loose
                      // contract of temp
o = temp;             //the object is again protected by the contract on o
```

5.2 Uniqueness

Many systems support the definition of *unique* references. A unique reference is the only reference pointing to a particular object. It thus has exclusive access to the object, allowing it to safely access, modify and move the object. Borrowing is frequently used in conjunction with unique objects to *temporarily* transfer access rights for an object to another part of the program, for example for the duration of a method's execution.

Several programming languages, including Clean [20] and Mercury [12], provide support for unique variables. AliasJava [1] and Hogg's islands [15] include a `unique` annotation.

Boyland et al.'s capabilities [5] can be used model uniqueness. A reference holding all seven capabilities, including all base rights, all exclusive rights and the ownership right, is a unique reference. The object to which it points cannot be modified through other references because of the existence of exclusive access rights.

Aliasing contracts do not restrict aliases themselves, but only object accesses. As such, there is no way in our system to ensure that an object is referenced by only one variable. However, we can model similar semantics by specifying that an object can be accessed by only one other object.

To imitate uniqueness semantics, we could annotate a variable with the rw-contract "`accessor == this`". In this way, only the object o holding the reference could access the object to which the variable points, o'. Other references to o' may exist; any of these could be used by o to make accesses as these accesses would not violate the contract; this is shown in the example below. However, no others object could use references to o', as this would cause a contract violation. Thus, our system does not restrict accesses to a *single reference*, but to a *single object*, in this case o.

```
class Foo {
  Object obj {accessor == this};  //An owned object.

  method() {
    Object obj2 {true};
    obj2 = obj;   //A second reference to obj's object.
```

```
    Foo f = new Foo();
    f.obj = obj;  //A third reference to obj's object.

    print(obj);   //Accessing the object through obj is legal.
    print(obj2);  //Accessing the object through obj2 is also legal.
                  // since the access still comes from this.
    print(f.obj); //Accessing the object through f.obj is also legal.
  }
}
```

Another difference between uniqueness semantics in other systems and our aliasing contracts is that encapsulating a variable in a single object (for example by using the rw-contract "`accessor == this`") does not necessarily guarantee access. Other references to the same object with conflicting contracts may exist; for example, several objects may have a reference with rw-contract "`accessor == this`" to the same object o. In this case, none of the objects could access o. This shows that encapsulating a reference within a single object means that it can be accessed by either one or zero objects.

In many schemes, uniqueness of variables is maintained with the help of destructive reads. Assignment then involves nullification of the old variable when the reference is transferred from one variable to another. We can use the same approach to transfer the access rights from one object to another when using aliasing contracts:

```
Object o {accessor == this};  //o's object can only be accessed from this
Object o2 {accessor == foo};  //o2's can only be accessed from foo
o = o2;                       //the object in o and o2 cannot be accessed
                              // from either this or foo (assuming this
                              // and foo do not refer to the same object).
o = null;                     //o2's object can be accessed from foo, we
                              // have transferred access rights to foo.
```

5.3 Owners-as-dominators and owners-as-modifiers

Two influential encapsulation policies, both revolving around the concept of ownership, are *owners-as-dominators* and *owners-as-modifiers*. Both of these schemes stipulate that references to an object must pass through the object's owner. Owners-as-dominators enforces this property for all references, while owners-as-modifiers considers only writable references. Clarke-style ownership types are an owners-as-dominators scheme, while universe types implement owners-as-modifiers.

We can easily model owners-as-dominators using the rw-contract "`accessor == this || accessor == accessed`"; the object pointed to by the annotated reference thus becomes owned by `this`. Any accesses coming from outside `this` will cause a contract violation; only accesses from the owner and the object itself are allowed. A slight modification of the contract to allow any read accesses to the object, even those not passing through the owner, allows us to model owners-as-modifiers: "`true, accessor == this || accessor == accessed`".

This is a very simple view of owners-as-dominators and owners-as-modifiers. Clarke-style ownership types [8] extend this notion and implement *deep* or transitive owners-as-dominators; they allow an object to be referenced by its owner, as well as any objects

owned directly or transitively by the same owner. Such references do not break the owners-as-dominators property, since the *reference path* to the encapsulated object still passes through its owner, *via* a number of other objects.

Encapsulation groups are designed to express this and similar encapsulation policies. We can use them to represent a hierarchy of objects similar to the ownership tree. The fact that one encapsulation group can contain other groups makes them very suitable for modelling transitive encapsulation policies such as that of Clarke-style ownership types. For each object, we define an encapsulation group (which we call `repGroup` in the following explanation) which contains all the objects it owns, both directly and indirectly. If an object `o1` owns another object `o2`, we add both `o2` and `o2.repGroup` to the encapsulation group `o1.repGroup`. If `o2.repGroup` has been defined correctly itself, it will contain all the objects directly and transitively owned by `o2`.

We can use the encapsulation group `repGroup` in the contract to give access to an object's owner, as well as all objects transitively or directly owned by the object's owner: "`accessor == this || accessor in repGroup`".

6 Discussion

In the last section, we showed that aliasing contracts can be used to model borrowing, uniqueness, owners-as-dominators and owners-as-modifiers. In addition, they can model a large range of other encapsulation policies including full encapsulation as used by Hogg's islands [15] and Almeida's balloon's [2] and module encapsulation as used by confined types [22] and type universes in universe types [19]. Aliasing contracts further support both multiple ownership and ownership transfer, addressing the widely criticised limitations of Clarke-style ownership types.

The fact that aliasing contracts can be used to model so many existing alias protection schemes shows that they represent a more general approach to controlling aliasing. Most existing schemes implement one specific encapsulation policy; for example, Clarke-style ownership types enforce owners-as-dominators while islands and balloons both use full encapsulation. The focus on a specific policy limits the flexibility of these systems since they are unable to support any other encapsulation policies; we may want to mix encapsulation policies even within a single program. Aliasing contracts allow developers to specify whichever policy is most applicable for each object; different parts of the system may have completely different encapsulation policies.

Encapsulation groups are a key feature of aliasing contracts which enables us to model transitive encapsulation policies such as owners-as-dominators in Clarke-style ownership types. Their power comes from the ability to nest encapsulation groups, creating a recursive structure. Encapsulation groups are extremely flexible; for example, they can easily create an encapsulation structure where only every second item in a list is encapsulated, while the remaining items are shared.

Despite being so flexible and expressive, aliasing contracts remain easy to specify: each contract declares the assumptions about only one variable and the object it references, without impacting any other part of the system and without imposing a particular structure on the program. Other systems, such as Clarke-style ownership types, impose a rigid ownership structure on the system which may be difficult for developers to construct in practice.

Aliasing contracts gain much of their flexibility by using dynamic rather than static checking. This allows the definition of complex encapsulation policies which depend on the aliasing structure of the system at run time.

Both static and dynamic checking have advantages and disadvantages, which are similar to the trade-off between static and dynamic type checking. Static checking allows errors to be discovered before the program is executed; dynamic checking requires the program to be executed. Dynamic checks have the added disadvantage of causing runtime performance overheads. On the other hand, static checking must be conservative in order to discover all possible errors; it may mistakenly report an error when there is not enough information available at compile-time to prove correctness.

The runtime performance overhead of aliasing contracts is the main disadvantage over existing, statically checkable schemes. However, we see aliasing contracts primarily as a testing tool, similar to assertions, which is used to identify bugs during the testing phase of a project but disabled in production code. Therefore, the runtime overhead caused by aliasing contracts is less concerning.

As future work on aliasing contracts, we want to see how much contract checking can be done at compile-time without compromising flexibility. Simple contracts could be verified at least partially at compile-time, reducing run-time overheads.

Most existing alias protection schemes (particularly statically checked schemes such as Clarke-style ownership types) directly control aliasing by restricting which variables can point to which objects. Dynamic schemes like dynamic ownership types [10], Boyland's capabilities [5] and our aliasing contracts restrict only accesses: they allow references which would be illegal in other systems but report an error if the reference is used to make an access. As a result, the encapsulation guarantees provided by both approaches are equivalent.

The reason that we prohibit accesses rather than aliases themselves is the increased flexibility it gives our system. Although a reference may not be usable at one point in the program's execution, a change in the program's aliasing structure may change this, as shown below.

```
Object o {true};                    //o's object is accessible through o
o = ...; //o refers to some object
Object o2 {false};                  //o2's object is hidden
o2 = o;                             //o's object is no longer accessible
o2 = new Object();                  //o's object is accessible again
```

Section 5.1 showed how aliasing contracts could be used to be model existing forms of alias control. We noted some degree of encoding of "borrowed", where we modelled temporary contract replacement by assigning to and from a new variable. However, we have resisted the temptation to add new primitives to explore aliasing contracts based on a principle similar to Hoare logic [13], types or JML [16]. In these systems, contract predicates (including types as a special case) may refer to program variables, but there are only predicate constants not predicate variables. This avoids discussions such as "what is the type of a type variable and can such variables be assigned?".

As a compromise, we could allow contracts to be temporarily suspended, for example for the duration of a method call. This would make borrowing easy to model without introducing a large amount of complexity for programmers. We plan to investigate contract suspension as part of our future work.

7 Conclusions and future work

We have presented a novel, dynamic approach to alias control called aliasing contracts. They are a very general and flexible scheme, able to model both one-level and transitive encapsulation through the powerful concept of encapsulation groups.

Aliasing contracts provide a unifying view of aliasing and are capable of modelling existing encapsulation policies including full encapsulation, uniqueness, owners-as-dominators, owners-as-modifiers and module encapsulation. Aliasing contracts can, for example, model all aspects of Clarke-style ownership types, including ownership contexts and transitive owners-as-dominators encapsulation. They also overcome issues of existing alias protection schemes, including lack of support for multiple ownership and transfer of ownership.

We are currently working on a prototype implementation of aliasing contracts in Java; we plan to use this prototype to quantify the performance overhead caused by aliasing contracts. We are also working on a static verifier for contracts which can verify simple contracts at compile time; this would reduce the number of dynamic contract checks required and thus increase performance.

Acknowledgement

The authors thank the Rutherford Foundation of the Royal Society of New Zealand for the scholarship which partly funded this work.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'12, pages 311–330. ACM, 2002.

[2] Paulo Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, 1997. 10.1007/BFb0053373.

[3] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'03, pages 213–223. ACM, 2003.

[4] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Proceedings of the 10th International Conference on Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[5] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 2–27, London, UK, UK, 2001. Springer-Verlag.

[6] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[7] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, chapter Ownership types: a survey, pages 15–58. Springer, 2013.

[8] Dave Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33:48–64, October 1998.

[9] Erich Gamma, Richald Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] Donald Gordon and James Noble. Dynamic ownership in a dynamic language. In *Proceedings of the 2007 Symposium on Dynamic Languages*, pages 41–52. ACM, 2007.

[11] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 354–378. Springer-Verlag, 2010.

[12] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, Chris Speirs, Tyson Dowd, Ralph Becket, Mark Brown, and Peter Wang. The Mercury language reference manual version 11.07. `http://www.mercury.csse.unimelb.edu.au/information/doc-release/reference_manual.pdf`, 2011.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[14] C. A. R. Hoare. Hints on programming language design. Technical Report CS-403, Stanford, CA, USA, 1973.

[15] John Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'91, pages 271–285. ACM, 1991.

[16] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, October 1998.

[17] Paley Li, Nicholas Cameron, and James Noble. Mojojojo - more ownership for multiple owners. In *International Workshop on Foundations of Object-Oriented Languages*, FOOL, 2010.

[18] Bertrand Meyer. Writing correct software. *Dr. Dobb's Journal*, 14(12):48–60, 1989.

[19] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, 1999.

[20] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. Clean language report version 2.2. `http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf`, 2011.

[21] Ilya Sergey and Dave Clarke. Gradual ownership types. In *Proceedings of the 21st European Conference on Programming*, ESOP'12, pages 579–599. Springer, 2012.

[22] Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'99, pages 82–96. ACM, 1999.

[23] Edwin Westbrook, Jisheng Zhao, Zoran Budimlic, and Vivek Sarkar. Practical permissions for race-free parallelism. In James Noble, editor, *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 614–639. Springer, 2012.