Technical Report

UCAM-CL-TR-880
ISSN 1476-2986

Number 880

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Access contracts: a dynamic approach to object-oriented access protection

Janina Voigt

February 2016

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

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

This technical report is based on a dissertation submitted May 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

ISSN 1476-2986

Abstract

In object-oriented (OO) programming, variables do not contain objects directly but addresses of objects on the heap. Thus, several variables can point to the same object; we call this aliasing.

Aliasing is a central feature of OO programming that enables efficient sharing of objects across a system. This is essential for the implementation of many programming idioms, such as iterators. On the other hand, aliasing reduces modularity and encapsulation, making programs difficult to understand, debug and maintain.

Much research has been done on controlling aliasing. Alias protection schemes (such as Clarke et al.'s influential ownership types) limit which references can exist, thus guaranteeing the protection of encapsulated objects. Unfortunately, existing schemes are significantly restrictive and consequently have not been widely adopted by software developers.

This thesis makes three contributions to the area of alias protection. Firstly, it proposes *aliasing contracts*, a novel, dynamically-checked alias protection scheme for object-oriented programming languages. Aliasing contracts are highly flexible and expressive, addressing the limitations of existing work. We show that they can be used to model many existing alias protection schemes, providing a unifying approach to alias protection.

Secondly, we develop a prototype implementation of aliasing contracts in Java and use it to quantify the run-time performance of aliasing contracts. Since aliasing contracts are checked dynamically, they incur run-time performance overheads; however, our performance evaluation shows that using aliasing contracts for testing and debugging is nevertheless feasible.

Thirdly, we propose a static analysis which can verify simple aliasing contracts at compile time, including those contracts which model ownership types. Contracts which can be verified in this way can subsequently be removed from the program before it is executed. We show that such a combination of static and dynamic checking significantly improves the run-time performance of aliasing contracts.

Author publications

Parts of the work reported in this thesis have been published during the period of PhD study. In particular:

- Chapter 2 contains extracts from a book chapter entitled "Notions of aliasing and ownership" by Alan Mycroft and Janina Voigt published in "Aliasing in Object-Oriented Programming" [70].
- Chapter 3 contains extracts from a technical report entitled "Aliasing contracts: a dynamic approach to alias protection" by Janina Voigt and Alan Mycroft published by the University of Cambridge Computer Laboratory [96].
- Chapter 5 contains extracts from a conference paper entitled "Dynamic alias protection with aliasing contracts" by Janina Voigt and Alan Mycroft published in the proceedings of APLAS 2013 [97].

Acknowledgements

My biggest thanks naturally go to my supervisor Alan Mycroft for his constant guidance and encouragement throughout the development of this thesis. His feedback and expertise have been invaluable to me and I have always found him willing to take time out of his busy schedule to teach me and help improve my work. I am particularly grateful for his frequent proofreading and his useful suggestions for improvements, in terms of both content and presentation.

I would also like to thank the Cambridge Computer Laboratory, particularly the other students from the CPRG research group (Dominic, Raoul, Stephen, Raphael, Tomas, Peter and others).

This work would not have been possible without the generous financial support of the Rutherford foundation of the Royal Society of New Zealand and the Cambridge Trust who supported me through the Cambridge-Rutherford memorial scholarship. I would also like to thank Trinity College for their academic and pastoral support and several very enjoyable feasts!

Finally, I want to thank my friends and family who were always on hand to distract me from my work and make my life more fun and enjoyable. In my time at Cambridge, I have met so many amazing people that I can't list them all, but I am sure they will know who they are! I want to particularly mention Steffen who helped proofread this thesis and Will whose expertise in theoretical computer science I required on more than one occasion and who was always keen to discuss his work or mine. Thanks also to Wal and Liz for frequent Skype conversations and constant encouragement. And of course a big thank you goes to my boyfriend Tobias for managing to put up with me on both good and bad days for the past years and for always finding ways to cheer me up when I was struggling with my work. Last but certainly not least, I want to thank my mother Angelica for her continued love and support through my many years at university (and of course beforehand).

Contents

CHAPTER 1

Introduction

In current OO programming languages, such as Java and C#, objects are 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 very important part of OO programming but can cause significant problems; in this thesis, we introduce and develop aliasing contracts to address the problems associated with aliasing.

1.1 Motivation

Aliasing is a powerful and central feature of OO programming, allowing an object to be shared efficiently by several parts of the system. Objects can be easily passed around a shared-memory system, simply by passing a reference, avoiding expensive copy operations. In this way, aliasing enables developers to describe complex data structures and construct systems as networks of interacting objects [64, 93].

There are a number of common programming idioms that rely on aliasing. Iterators [37] are frequently discussed in literature about aliasing and are difficult to implement efficiently without the ability to share the objects in the collection between the collection itself and the iterator traversing the collection [73]. Similarly, the fly-weight design pattern [37], which aims to share common objects in order to decrease storage costs, relies on aliasing. Even the simple scenario of one object belonging to two or more separate collections at the same time requires support for aliasing [76].

While aliasing provides flexibility and enables sharing, it also causes a number of problems in practical programming and formal verification. These problems stem from the fact that an object can be changed through an aliasing reference by a seemingly unrelated part of the system. 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" [47].

Through aliasing, the internal details of an object can be referenced from anywhere in the system and may be modified without the object's knowledge. This phenomenon is known as *representation exposure* since an object's representation is exposed to modifications from the outside [68], constituting a serious breach of encapsulation and information hiding [80].

Although many OO programming languages provide access modifiers such as `private`, `protected` or `public`, they are an inadequate solution to the problem of representation exposure. Access modifiers limit the scope in which variables can be used; they protect only the variable name, not the object to which the variable points [48, 76]. This means that even objects stored in `private` variables may not be fully encapsulated if their reference is leaked to other parts of the system.

Consider the simple `LinkedList` example below, written in Java. A `LinkedList` contains a number of `Nodes` and holds a reference to the `head` node of the list; each `Node` in the list then has a reference to the `next` node in the list:

```
class LinkedList {
  private Node head;
  public void addItem(Object data) {
    Node newNode = new Node(data);
    newNode.setNext(head);
    head = newNode;
  }
  public boolean member(Object data) {
    for(Node n = head; n != null; n = n.getNext()) {
      if(n.getData().equals(data)) {
        return true;
      }
    }
    return false
  }
  public Node getHead() { return head; }
}
class Node {
  private Node next;
  private Object data;
  public Node(Object data) { this.data = data};
  public Node getNext() { return next; }
  public Object getData() { return data; }
  public void setNext(Node next) { this.next = next; }
}
```

Although `next` in `Node` and `head` in `LinkedList` are both declared to be `private`, the list structure is exposed to modification from other parts of the system: references to `next` and `head` are leaked in several places.

The first leak is caused by the `getNext` and `getHead` methods in `Node` and `LinkedList` which return a reference to `next` and `head` respectively; a client calling these methods can thus obtain a direct reference to objects stored in `next` and `head` and could modify them, for example as follows:

```
LinkedList list;
...
list.getHead().setNext(new Node(newData));
```

Another encapsulation leak occurs in the `setNext` method in `Node`: it accepts an existing `Node` object as an argument and stores the reference to this object in its `next` field. However, the given argument may already be aliased by the client calling the method, thus giving the client continued access to the supposedly private `next` node.

Generally speaking, there are two main ways in which a reference to a `private` object can be leaked, as illustrated by the above example: it can be passed out to another part of the system (for example as a method return value or method call argument) or it can be passed in by another part of the system (for example as a method call argument). Exposure of `protected` fields can additionally occur in subclasses without a class' knowledge [51].

A well-known bug caused by aliasing commonly cited in literature occurred in Sun's Java Development Kit 1.1.1 [3, 57, 76, 94]. The code causing the bug is shown below:

```
class Class {
  private Object[] signers;

  Object[] getSigners() {
    return signers;
  }
}
```

The defect was caused by the security system method `Class.getSigners()` which returned a reference to the internal (and `private`) array of trusted applets, allowing malicious applets to modify it and pose as trusted code [57]. Vitek et al. note that "what is interesting about this example is that none of the standard Java protection mechanisms seem to help" [94].

In fact, standard OO programming languages do not include any features to avoid representation exposure (since, as we saw above, access modifiers are insufficient for this purpose); Noble et al. conclude that "in practice, then, careful programming and eternal vigilance are the only defences against aliasing problems in current object oriented languages" [76].

Representation exposure and encapsulation breaches caused by aliasing severely reduce the modularity of software. Since objects (even supposedly `private` and encapsulated ones) can be modified by any part of the system which can obtain a reference to them, modules are no longer independent of each other. This makes software difficult to understand, debug and maintain.

Due to the significant negative effects of aliasing, many systems been developed to control it; such systems are often called *alias protection* schemes. Perhaps the best-known such system is Clarke et al.'s ownership types [27], which ensures that all reference paths to an encapsulated object pass through the object's single *owner*. This prohibits references to encapsulated objects being leaked, as in the two examples above, thus preventing representation exposure. We discuss and compare various existing alias protection schemes in Chapter 2.

Many existing alias protection schemes are, however, very restrictive, complicating how users express common idioms and design patterns (such as iterators) which rely on sharing; this often makes them impractical to use. Ownership types, for example, stipulate that each object must have a single owner. This means that an object must be *either* encapsulated or shared, but not both at the same time. An additional issue with ownership types is that ownership is optional and alias protection can be completely avoided by declaring all objects to be shared.

1.2 Aliasing contracts

In this thesis e propose a dynamic approach to alias protection which we call *aliasing contracts*. Using aliasing contracts, developers can specify which parts of a system should be allowed to access a particular object. This avoids the problem of representation exposure because accesses to the internal (and encapsulated) representation objects become illegal. In the above example, we could specify that the representation of the `LinkedList` (that is, its nodes) can be accessed only be the `LinkedList` itself; if any other part of the system tries to access them, an error is reported.

Aliasing contracts highly flexible and expressive, allowing them to model many aliasing conditions which cannot be handled by existing systems. They are easy to use for software developers, which we suggest means that they are more likely to be taken up in practice; in addition, we expect that users will be less likely to "hack around" or avoid them. In addition, aliasing contracts are powerful enough to model many existing alias protection schemes such as ownership types. They thus give us a unifying approach to alias protection; we can express existing systems in the same base language – aliasing contracts – facilitating direct comparisons between them.

One disadvantage of aliasing contracts over existing systems is their run-time performance overhead. Aliasing contracts are checked dynamically, not statically as, for example, ownership types. However, we suggest that aliasing contracts will be particularly valuable during the testing phase of software development, as they help developers detect unintended reference leaks and representation exposure. During testing, performance overheads caused by aliasing contracts are less of a concern.

We further mitigate the performance overheads of aliasing contracts by developing static analysis that can verify some aliasing contracts at compile time; contracts that can be verified statically can be removed from the program and do not need to be re-checked at run time.

1.3 Thesis structure

The remainder of the thesis is structured as follows:

- Chapter 2 describes the background of this work and gives detailed descriptions and comparisons of existing alias protection schemes.
- Chapter 3 introduces and formalises aliasing contracts; it includes a syntax and operational semantics for aliasing contracts, as well as a case study showing how they can be applied in practice.
- Chapter 4 performs a detailed comparison between aliasing contracts and existing alias protection schemes. It shows how existing systems can be modelled using aliasing contracts.
- Chapter 5 develops JaCon, a prototype implementation of aliasing contracts in Java and uses it to quantify the run-time performance of aliasing contracts.
- Chapter 6 presents a static analysis which can verify many simple aliasing contracts at compile time and demonstrates the benefits of combining static and dynamic checking of aliasing contracts.

- Chapter 7 discusses various aspects of aliasing contracts and promising future research directions.
- Chapter 8 presents the work's conclusions and summarises our contributions.

CHAPTER 2

Background

In Chapter 1, we introduced the concept of aliasing, which is a both essential and problematic part of OO programming. We discussed representation exposure, where a reference to an object's internal representation is leaked to other parts of the system, exposing it to unexpected accesses and modifications.

Reynolds describes another, albeit less common, scenario where aliasing can cause problems [84]: a method expecting two separate arguments is in fact given two references to the same object. More generally speaking, problems occur if a program expects two variables to refer to distinct objects when they are in fact aliases for the same object (or vice versa).

Aliasing causes additional problems in the context of class invariants which describe valid states of objects. If an object's state is improperly encapsulated (and aliases exist to it from other parts of the system), the object's invariant may be broken without its knowledge; this can happen if the state is modified through aliases by other parts of the system [52]. The object essentially loses control of its own invariant and can no longer ensure the validity of its state. This problem was first identified by Meyer in [64] and is commonly called the *indirect invariant effect*.

Because multiple references may point to the same object, memory deallocation and storage management are also more complex in the presence of aliasing [12, 66]. Deallocating an object can lead to dangling references which have the potential to crash program execution if they are dereferenced. This can in turn make programmers reluctant to deallocate objects, leading to memory leaks. For this reason, many modern programming languages provide garbage collection (GC).

As a result of these disadvantages, much work has been done to control aliasing. In this chapter, we survey various existing approaches to the problems associated with aliasing.

In their seminal paper "The Geneva convention on the treatment of object aliasing", Hogg et al. classify approaches to the problem of aliasing using four categories: detection (finding potential or actual aliasing statically or at run time), advertisement (declaring the aliasing properties of program parts), prevention (prohibiting certain aliasing from occurring in the first place) and control (isolating the effects of aliasing) [49]. We start by looking at approaches to *preventing* aliasing in Section 2.1; prevention is the approach taken by so-called alias protection schemes. We also consider alias and pointer analyses used for alias *detection* in Section 2.2.

2.1 Alias protection

Much research has been done to control aliasing and address its negative effects. A number of *alias protection* schemes have been developed which limit the existence and usage of aliases. The variety of such schemes is huge; in the sections below, we discuss various important properties of existing alias protection schemes and compare and categorise existing work in this area. An overview of the most important works discussed here and their properties is shown in Table 2.1 and Table 2.2.

2.1.1 Aliasing policies

Alias protection schemes usually limit aliasing according to a particular aliasing policy. In this section, we describe various different aliasing policies and give examples of alias protection schemes which use each policy.

2.1.1.1 Full encapsulation

Full encapsulation is a simple, but restrictive aliasing policy supported by the earliest alias protection schemes for OO languages: Hogg's islands [48], Almeida's balloons [4] and Utting's local stores [93].

Full encapsulation means that a group of objects is encapsulated inside a single object called the *bridge* in Hogg's islands [48] and the *balloon* in Almeida's balloons [4]. Encapsulated objects can reference each other freely, as can unencapsulated objects outside islands and balloons. However, no referencing is allowed between encapsulated and unencapsulated objects; all such references must pass through the bridge or balloon object. This situation is illustrated in Figure 2.1. Solid arrows represent legal references, while dotted arrows with crosses represent illegal or impossible references.

Objects transitively referenced by the bridge or balloon are considered to be encapsulated. This means that references from an encapsulated object to an unencapsulated object are *impossible*; if such a reference existed, the referenced object would also be encapsulated. References from the outside to encapsulated objects are *illegal*; this is enforced by ensuring that the bridge or balloon does not leak references to encapsulated objects.

Full encapsulation implies that the state of encapsulated objects cannot change between method invocations of the bridge or balloon object; thus, the group of encapsulated objects can be treated as a black-box and can be tested independently, allowing for modular verification [4, 48].

While conceptually simple, full encapsulation is highly restrictive. The requirement that encapsulated objects may only reference other encapsulated objects is particularly problematic, as it prohibits all sharing between encapsulated and unencapsulated objects.

To mitigate the restrictiveness of full encapsulation, islands and balloons unsafely loosen the aliasing constraints for certain types of aliases (see Section 2.1.2.4 below). However, this does not address the actual shortcomings of full encapsulation; Noble et al. argue that this approach results in alias protection that is both too lax and too restrictive at the same time [74].

Utting also proposes a full encapsulation scheme, which is based around the notion of *local stores* [93]; this technique is an application of *syntactic control of interference* [93]

System	References or object accesses	Static or dynamic checking	Static versus dynamic aliasing	Shared	Read-only	Borrowing
Islands [48]	References	Static	Yes	No	Yes - `read` annotations	Yes - implicit with dynamic aliasing
Balloons [4]	References	Static	Yes	No	No	Yes - implicit with dynamic aliasing
Clarke-style ownership types [27]	References	Static	No	Yes - objects in the root context	No	No
Ownership types: JOE [24]	References	Static	Yes	Yes - objects in the root context	No	Yes - implicit with dynamic aliasing
Ownership types with inner classes [15]	References	Static	No	Yes - objects in the root context	No	No
Multiple ownership: MOJO [22] / Mojojojo [60]	References	Static	No	Yes - objects in the root context	No	No
Dynamic ownership types [40]	Accesses	Dynamic	No	Yes - objects in the root context	No	No
Universe types [68]	References	Static	No	Limited - read-only references	Yes - read-only references	No
Confined types [94]	References	Static	No	Yes - all instances of unconfined types	No	No
Ownership domains [2]	References	Static	No	Yes - objects in the `shared` domain	No	No
Boyland et al.'s capabilities [21]	Accesses	Both	No	Yes - objects with no exclusive capabilities	Yes - objects without base write right	Yes

Table 2.1: Properties of influential existing alias protection schemes

System	Uniqueness	Linearity	Multiple ownership	Ownership transfer	Encapsulation policy
Islands [48]	Yes - objects annotated as `unique`	No	N/A	N/A	Full encapsulation
Balloons [4]	Yes - unique references to balloons	No	N/A	N/A	Full encapsulation
Clarke-style ownership types [27]	No	No	No	No	Transitive owners-as-dominators
Ownership types: JOE [24]	No	No	Limited - dynamic aliasing	Limited - dynamic aliasing	Transitive owners-as-dominators
Ownership types with inner classes [15]	No	No	Limited - inner classes	No	Transitive owners-as-dominators
Multiple ownership: MOJO [22] / Mojojojo [60]	No	No	Yes	No	Transitive owners-as-dominators
Dynamic ownership types [40]	No	No	Limited - exported objects	Yes	Transitive owners-as-dominators
Universe types [68]	No	No	Limited - type universes	Yes - extension of universe types [69]	Peer owners-as-modifiers
Confined types [94]	No	No	N/A	N/A	Module encapsulation
Ownership domains [2]	No	No	No	No	Various
Boyland et al.'s capabilities [21]	Yes - objects with all 7 capabilities	Yes	N/A	N/A	Various

Table 2.2: Properties of influential existing alias protection schemes

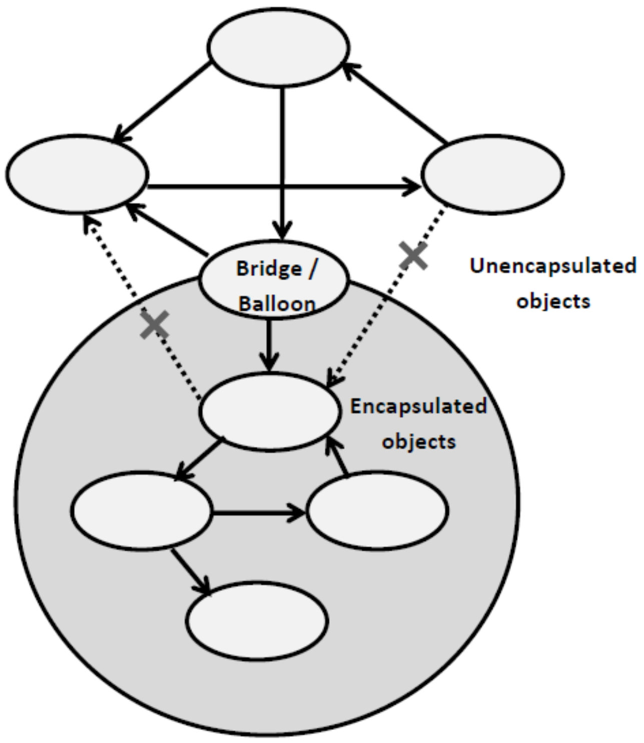


Figure 2.1: Alias protection with full encapsulation

described by Reynolds [84]. One local store contains and protects any number of objects. All accesses to a group of objects must go through the local store which effectively acts as a bridge or balloon. To mitigate the restrictiveness of full encapsulation, Utting's local stores support transferring objects from one store to another, providing added flexibility [93].

2.1.1.2 Uniqueness and linearity

Many alias protection schemes support the definition of *unique* references. A unique reference is the only reference pointing to a particular object. It thus has exclusive access rights to the object, allowing it to safely access, modify and move the object, even in the presence of concurrency.

Uniqueness significantly simplifies reasoning about programs [28] and GC, as it requires no reference counting [98]. Several programming languages, including Clean [81], Mercury [44] and C++ [91], provide support for unique variables.

Figure 2.2 shows the semantics of uniqueness. A new reference to a unique object can be added only when the previous reference has been removed.

To maintain uniqueness, a system must ensure that a unique reference remains the only reference to an object. The simplest way to achieve this is to prohibit assignments of unique variables, ensuring that once stored in a unique variable, a reference can never be transferred to another variable. This approach is used by Almeida's balloon types [4]: references to balloon objects must be `unique` and this is enforced by prohibiting assignments of variables referencing existing balloons.

Another less restrictive option is to modify the semantics of assignment to nullify the original variable at the end of the assignment operation; we call this *destructive assign-*

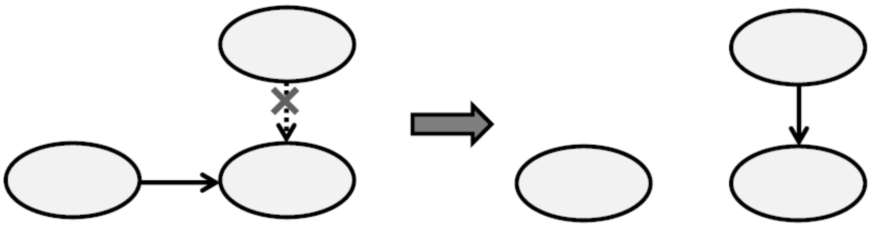


Figure 2.2: Alias protection with uniqueness

ments. This approach is used by the `auto_ptr` class template in C++ [91], by Minsky in his proposal for *unsharable objects* [66] and by Hogg to support `unique` references [48].

Linear types, based on Girard's linear logic [38], are a stricter approach to uniqueness: in addition to being unique, a variable of a linear type must be *used exactly once*. (A slightly looser version is *affine* linearity, where a linear variable must be used *at most once*.) A good overview and formalisation of linear types is given by Wadler [98].

A linear variable is nullified at run time when it is read through a *destructive read*. (This differs from the previous approach, where a variable can be used any number of times and is nullified only on assignment.) Of course, if a variable is nullified as soon as it is read, we can be certain that only one alias to the referenced object can ever exists.

Baker proposes *use-once* variables with linear semantics [7]; they are useful when working with inherently linear objects such as input and output streams. Baker notes that the requirement to use a variable exactly once leads to a slightly different programming style: a function with a use-once variable must either pass it to another part of the system to be used there or explicitly consume it. Baker claims that an experiment he conducted, using use-once variables to program various benchmarks and algorithms, showed that this linear style of programming is easy to adopt [7].

Kobayashi's *quasi-linear types* [55] represent a middle-way between linear and uniqueness types. Kobayashi argues that linear types have not been applied widely in practice because they are highly restrictive; instead, he proposes quasi-linear variables which can be used any number of times *locally*, but only once elsewhere in the program.

PacLang [36] uses a quasi-linear type system to simplify the programming of network processors. It aims to protect data packets from concurrent accesses by multiple threads, while facilitating easy flow of packets between threads. The type system allows only one thread to reference a data packet at a time but allows multiple local references within one thread. Kilim, an actor framework for Java, extends the unstructured data packets used by PacLang [89] to support the passing of unique tree-structured objects between processes.

Boyland argues that destructive reads and assignments for maintaining uniqueness have several disadvantages [18]: they force the programmer to constantly restore nullified values, increasing complexity; they may make a method appear to have more side-effects than it actually does; they require modification of assignment semantics to fit with existing programming languages.

Boyland also claims that destructive reads are often unnecessary if the nullified variable is not used again. He proposes *alias burying* [18], which marks existing aliases as undefined (buried) when a unique variable is read at run time. Accessing previous (now undefined) aliases halts the program with an error. Boyland then shows that modular static analysis can be used to identify programs where all buried aliases are dead (that is, never used again). He argues that such programs do not require uniqueness at all; they execute in exactly the same way with standard programming language semantics.

2.1.1.3 Ownership: owners-as-dominators and owners-as-modifiers

A number of alias protection schemes are based on the notion of object ownership: an object owns its components and controls access to them. References (and thus accesses) to components must pass through the owner, allowing it to remain in control of its representation. Many alias protection schemes based on ownership can be type-checked at compile time. A detailed overview and classification of ownership-based alias protection schemes can be found in [26].

Two significantly different approaches to ownership have been highlighted in previous work: *owners-as-dominators* and *owners-as-modifiers* [22, 32]. In an owners-as-dominators scheme, the owner objects fully control their owned object: all references (and accesses) to the owned object must pass through the owners. Owners-as-modifiers limits only write accesses to an object while allowing read accesses to bypass an object's owners [32]. This approach is problematic in concurrent settings, where unrestricted read accesses may result in the observation of inconsistent object states.

Ownership schemes can be further categorised by whether they support single or multiple ownership. Single ownership results in the object graph forming a hierarchical tree structure called the *ownership tree*, while multiple ownership produces a *directed acyclic graph*.

While single ownership is conceptually simple, it has been widely criticised for limiting sharing of objects, making it difficult to implement iterators and other idioms that require sharing [15, 52]. Empirical studies have shown that a tree-shaped ownership structure does not suit all programs [1, 67, 83]. For example, a study of heaps by Mitchell found that multiple ownership was required by 75% of object structures [67].

Another important issue in ownership-based alias protection schemes is ownership transfer, where an object is transferred from one owner to another. Most ownership-based alias protection schemes do not support ownership transfer, as it does not fit well with the static ownership system [27]; they require an object's owner to remain the same throughout its lifetime.

Mycroft and Voigt use the example of bank accounts to demonstrate that both multiple ownership and ownership transfer are common in the real world [70]. Shared bank accounts, for example, require multiple ownership (with different owners having potentially different access rights); transfer of ownership occurs when adding and removing account holders. The authors argue that given its prevalence in real-world applications, supporting both multiple ownership and ownership transfer in alias protection schemes is desirable.

Nägeli similarly argues that both multiple ownership and ownership transfer are essential in practice [71] after attempting to apply ownership-based alias protection to 23 design patters described by Gamma et al. [37].

Owners-as-dominators Owners-as-dominators stipulates that all references to an object must pass through the object's owners. In this section, we look at single-ownership systems; extensions to support multiple ownership are discussed below.

In the simplest form of owners-as-dominators (which we call *strict owners-as-dominators*), an object can be accessed only by its owner. This situation is shown in Figure 2.3a.

Strict owner-as-dominators is rather restrictive; it enforces a rigid tree-structured object heap, where references can go only from the owner to the owned object. Such a tree structure is difficult to achieve in practice, as it does not allow any sharing of objects to occur at all.

To gain more flexibility, we can slightly relax the ownership condition, allowing objects with the same owner to access each other. We call this *peer owners-as-dominators*. Figure 2.3b shows which references are now allowed; the legal references in peer owners-as-dominators are a superset of the legal references in strict owners-as-dominators.

Transitive owners-as-dominators, shown in Figure 2.3c, relaxes the ownership condition even further; the references it allows are a superset as those legal in peer owners-as-dominators.

Transitive owners-as-dominators stipulates that all accesses to an object must *pass through* the object's owner:

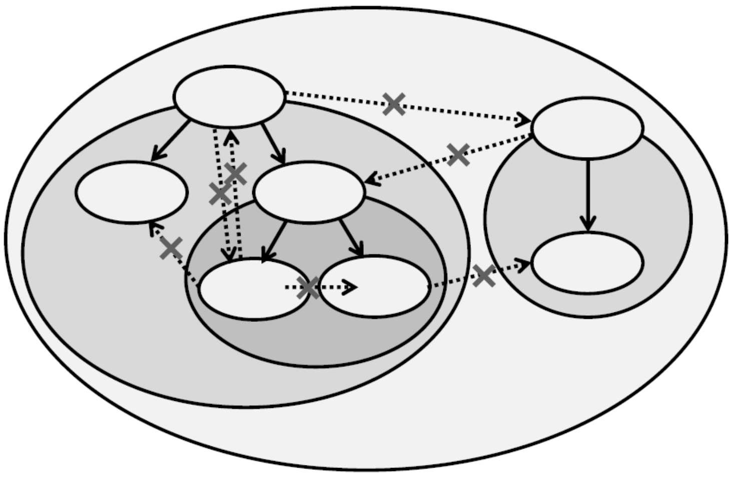
- References may go at most one level down the ownership tree, from an object to the objects it owns. For example, in Figure 2.3c a reference from `o1` to `o2` is permitted. However, `o1` may not reference `o3` as such a reference would bypass `o3`'s owner, `o2`.
- References may go any number of levels up the ownership tree, from within an object to the outside. For example, a reference from `o3` to `o4` is permitted. If we look at the entire reference path from the root of the ownership tree (`o1`) through `o3` to `o4`, we can see that the path passes through `o4`'s owner, `o1`.
- As a result of the above rules, the objects at the top-level of the ownership tree (for example `o1`) are shared and accessible to all other objects. They can be reached from every object in the system by going any number of levels up the ownership tree and at most one level down.

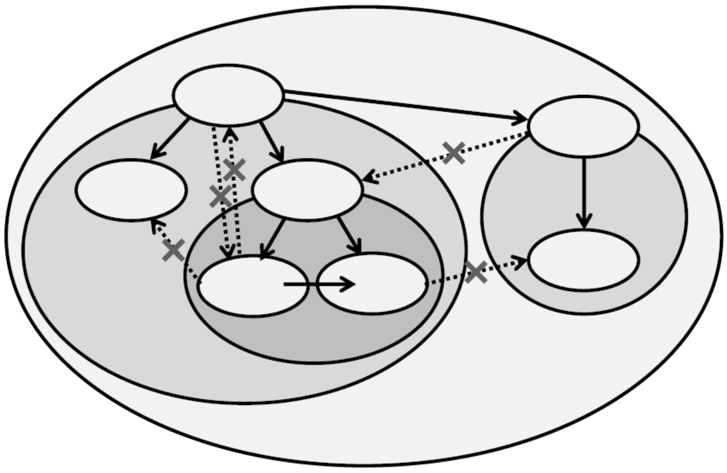
In their seminal paper "Ownership types for flexible alias protection", Clarke et al. introduce ownership types [27], a transitive owners-as-dominators system. Since the original proposal, many extensions to ownership types have been proposed (as we discuss below). We use the term *Clarke-style ownership types* to refer to the original ownership types proposal.

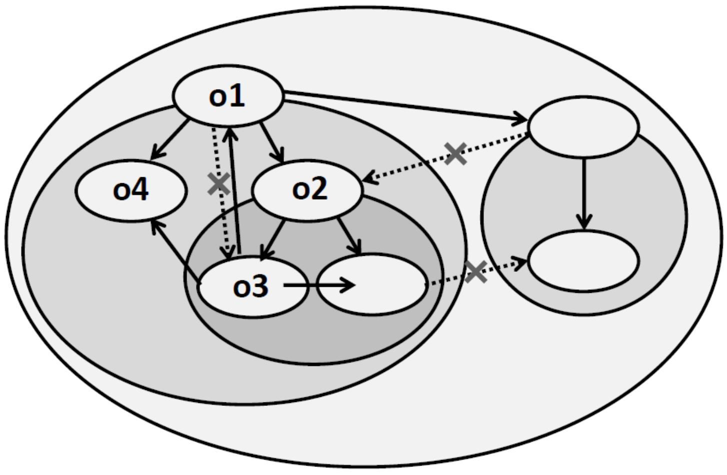
Clarke-style ownership types grew out of work on *flexible alias protection* by Noble et al. [76] which aimed to address the limitations and inflexibility of full encapsulation schemes such as Hogg's islands and Almeida's balloons [74]. In a major departure from full encapsulation systems, it allows encapsulated objects to reference unencapsulated objects (without the referenced objects also becoming encapsulated). This significantly increases the flexibility of Clarke-style ownership types compared to full encapsulation systems [42].

Clarke et al. explain that in Clarke-style ownership types "each object owns a context, and is owned by the context that it resides within" [27]. These contexts form the object's interior and exterior. Clarke-style ownership types are a single-ownership transitive owners-as-dominators scheme; they ensure that all reference paths to an object pass through its single owner context.

Every variable is annotated to specify its aliasing properties. The annotation `rep` signifies that the object stored in the variable is owned by the object which declares variable. Ownership types provide "object protection" [27]: the `rep` context is different for each object, including for different objects of the same class. Therefore, unlike for


(a) Alias protection with strict owners-as-dominators


(b) Alias protection with peer owners-as-dominators



(c) Alias protection with transitive owners-as-dominators

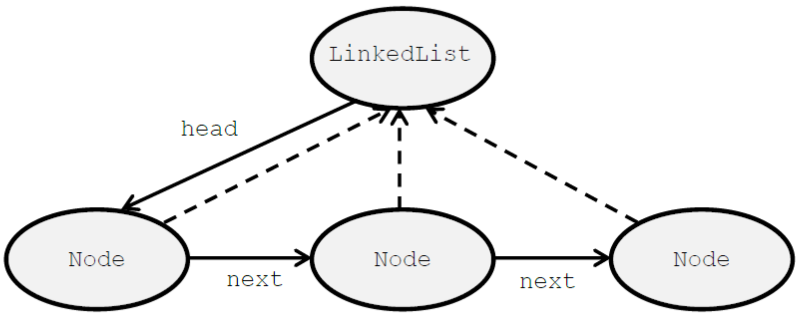


Figure 2.4: Ownership structure of the `LinkedList` example

example in Java, private (that is, owned) representation members cannot be accessed by other objects of the same class [27].

Alternatively, references may be annotated with `norep`, in which case the object is owned by the entire system and may be accessed from anywhere [27]; `norep` objects reside at the top-level of the ownership tree.

In addition to `rep` and `norep`, Clarke-style ownership types include a third annotation called `owner`. An object stored in an `owner` reference has the same owner as the object holding the reference. In the example below, a `LinkedList` implementation owns the first `Node` in the list (`head`) and each subsequent `Node` (`next`) has the same owner – the `LinkedList`:

```
class LinkedList {
  rep Node head;
}
class Node {
  owner Node next;
}
```

A visual overview of the ownership structure in this example is shown in Figure 2.4; solid arrows show references, while dotted arrows represent ownership.

This example illustrates the interesting property of Clarke-style ownership types that the owner of an object does not need to have a direct reference to it. This is legal in transitive owners-as-dominators, since the path to the indirectly referenced owned object still passes through the owner.

Clarke-style ownership types further support *context parameters* which allow different objects of the same class to exhibit different ownership properties. This significantly increase the flexibility of Clarke-style ownership types – much in the same way as generic types. Clarke et al. call this *owner polymorphism* [26].

A class can declare one or more context parameters. When an instance of the class is created, instantiations for the context parameters must be provided. These instantiations can be `rep`, `norep`, `owner` or another context parameter. Instead of declaring references as `rep`, `norep` or `owner`, a class can use its context parameters to annotate references. This means that the exact ownership behaviour depends on the values of the parameters provided when an object of the class is instantiated. Consider the following example taken directly from [27]:

```
class Pair<m, n> {
  m X fst;
```

```
    n Y snd;
  }
```

Here, `Pair<rep, rep>` is a `Pair` which owns both `fst` and `snd`. `Pair<rep, norep>`, on the other hand, is a `Pair` which owns only `fst`, while `snd` is accessible to the entire system.

Similarly to the `owner` annotation, context parameters can be used when an object does not hold a direct reference to an owned object. Instead, it can pass along its `rep` context as a context parameter. For example, this version of `LinkedList` is equivalent to the previous one which used `owner` annotations:

```
  class LinkedList {
    rep Node<rep> head;
  }
  class Node<m> {
    m Node<m> next;
  }
```

Criticisms: Clarke-style ownership types have been criticised for their single-ownership model, which is too restrictive to implement many common design patterns and programming idioms [15, 22, 25, 32, 57, 74]. Clarke-style ownership types also lack support for ownership transfer.

A further criticism of ownership types is their lack of support for inheritance [27]. Early ownership type papers considered only a simple programming language without inheritance and did not describe how ownership types could work in its presence [25, 68, 94]. Clarke et al. address this criticism in follow-up papers [24, 25], noting that in the presence of subtyping, an object's owner must remain invariant, while its representation context may vary [25].

Some authors have also criticised context parameters, arguing that, although they increase flexibility, they are quite complex to use [82].

Owners-as-modifiers Unlike owners-as-dominators, owners-as-modifiers limits only write accesses to objects. We again distinguish between strict owners-as-modifiers, peer owners-as-modifiers and transitive owners-as-modifiers.

Universe types first proposed by Müller et al. in 1999 [68] is an example of a peer owners-as-modifiers system. Each object is associated with a partition of the object store called its *universe* which contains the object's representation. An object can be in only one universe, making universe types a single-ownership scheme.

An object may reference and modify objects belonging to its own universe and two objects in the same universe (that is, sharing the same direct owner) may modify each other freely. All other references across universe boundaries must be read-only references which cannot be used to modify objects [32, 68]. Read-only references can experience *observational exposure*: although they cannot themselves modify an object, they can observe changes in the object [26].

Universe types can be checked statically and have been formalised and proven using Isabelle/HOL [54].

Müller et al. argue that universes are more flexible than Clarke-style ownership types because of their owners-as-modifier approach [32, 68]. They suggest that iterators for collections could be implemented by having read-only references to the items in the collection; this would allow the iterator to traverse, but not modify, the collection. When the iterator needs to make modifications, it could delegate these to the collection [32].

Nevertheless, universe types still suffer from similar shortcomings as Clarke-style ownership types, since they do not support multiple ownership or ownership transfer. Although an iterator implementation as described above is feasible, universes remain unable to model iterators and collections as closely collaborating, trusted objects which share full access rights to the collection objects.

Extensions to ownership-based systems We now look at various extensions that have been proposed to ownership-based alias protection schemes such as Clarke-style ownership types and universe types.

Dynamic ownership: Some work has been done on applying the static concept of ownership to dynamically-typed and prototype-based languages [40, 75]. Gordon et al., for example, implement dynamic ownership in the language ConstrainedJava [40]. Ownership information for every object is stored in a special `owner` field [75] and is used to check the validity of method calls at run time; any invalid method calls (that is, calls to an object's method where the access path does not pass through the object's owner) raise an exception. This dynamic checking of ownership information causes execution overheads [75]. Gordon et al., for example, report a performance decrease of 40 to 50 percent compared to statically-checked ownership types [40].

Dynamic ownership types represent a major departure from previous work on ownership types because it does not restrict aliasing itself but ensures that no messages can be sent through illegal aliases. We take the same approach in our work on aliasing contracts, as explained in Chapter 3.

Gradual ownership: Wrigstad et al. note that a significant disadvantage of many ownership types systems is their "all-or-nothing approach" [101]. Partially annotating a program with ownership types does not usually produce any guarantees about encapsulation. This problem is addressed by Sergey et al.'s *gradual ownership types* [86]: partially annotated programs are checked at run time, while fully annotated programs are checked statically as in previous work. This allows programmers to gradually migrate programs with no ownership types to fully annotated programs. The authors argue that such a migration is analogous to a move from untyped to typed code, as supported by gradual typing [87].

External uniqueness: While uniqueness, as described in Section 2.1.1.2 above, simplifies reasoning about programs, it can also be quite restrictive as it completely disallows any sharing of objects. Clarke et al. [28] and Wrigstad [100] propose *external uniqueness*, an application of the ownership concept to uniqueness. Their proposal loosens uniqueness constraints to allow any number of internal references within aggregate objects but only a single external reference [28]. The authors argue that internal references to an aggregate object which do not cross encapsulation boundaries are innocuous since they do not affect how the aggregate is viewed externally. External references on the other hand are dangerous and should be restricted to allow only one unique reference [28].

Clarke et al. note that external uniqueness differs from Clarke-style ownership types in that it "refines the object graph property underlying ownership types from dominating

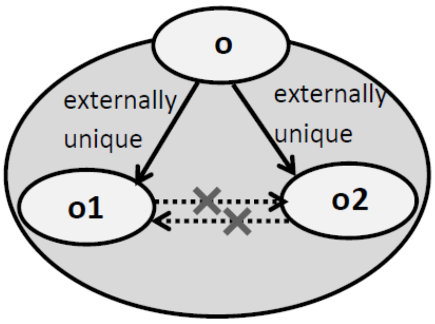


Figure 2.5: An example of forbidden references in external uniqueness

nodes to dominating edges" [28]; Clarke et al. call this *owners-as-dominating-edges* [26]. Any references to the aggregate object must pass through the single external reference, rather than the single owning object.

The difference becomes clear, if we consider an object o which owns two externally unique objects o_1 and o_2. With Clarke-style ownership types, o_1 and o_2 could reference and access each other since they are owned by the same object. With external uniqueness, all references to o_1 must pass through the externally unique reference held by o; thus, o_2 cannot hold a reference to o_1. We can see external uniqueness as a transitive owners-as-dominators system which disallows references between (externally unique) peers. This is shown in Figure 2.5.

Ownership transfer: Ownership transfer is not supported by most ownership systems; an exception to this is Müller et al.'s work on ownership transfer in universe types [69]. The authors apply the concept of external uniqueness [28] to ownership transfer. They present UTT, an extension of universe types, which allows a cluster, or group, of objects to be transferred from one owner to another. UTT uses modular, intraprocedural static analysis to ensure that references to clusters are externally unique at the time of ownership transfer [69].

Gordon et al.'s dynamic ownership types also support ownership transfer [40]. Permission for ownership transfer must be granted by the object's current owner.

Multiple ownership: One major criticism of many existing ownership-based alias protection schemes is their lack of support for multiple ownership. Like ownership transfer, multiple ownership is difficult to capture in simple, static type systems.

A first step towards allowing multiple ownership is taken by Boyapati et al. [13, 15]. They allow objects of inner classes to access `rep` objects of their containing outer object. This represents a very restricted version of multiple ownership, where the representation of the outer object is effectively owned by both the outer object and the inner object; however, this does not significantly change the underlying single-ownership model and its associated limitations.

Gordon proposes a limited form of multiple ownership for dynamic ownership types, allowing one object to *export* another, thus giving both objects the same ownership context [39]. After the export operation, the exported and exporting object occupy the same place in the ownership tree; this means that they have the same owner and own the same objects. The objects owned by the exporting object have effectively gained the exported object as a second owner.

Cameron et al. implement full multiple ownership in the language MOJO, a relatively simple extension of existing single-ownership languages [22]. Unlike other ownership schemes, this system is descriptive rather than prescriptive; it does not attempt to enforce an owners-as-dominators discipline. Instead, it relies on a powerful effect system to detect interference between computations. However, the required effect specifications are arguably complex for users to define and significantly increase the annotation overhead.

In follow-up work, Li et al. present Mojojojo, a successor to MOJO. Mojojojo uses existential types and generics, allowing more expressiveness than is possible with MOJO [60]. For example, in Mojojojo different instances of the same class can have a different number of owners. The authors also claim that the formal foundation of Mojojojo is smaller and less complex than that of MOJO.

Östlund et al. propose *ombudsmen-as-dominators*, supporting the definition of multiple bridge objects (or *ombudsmen*) to a shared representation [79]. Their work can be seen as a further simplification of Mojojojo [26].

Östlund et al. explain that in ombudsmen-as-dominators "every path from a root in the system to an object in an ombudsman-dominated context contains one of the context's ombudsmen" [79]. In addition to their private (that is, standard) ownership contexts, objects at the same level of the ownership tree share a common ombudsman-dominated context with shared objects. The authors use the term ombudsman to emphasise that the ombudsmen work together in a "benevolent" [79] manner.

Although ombudsmen-as-dominators allows some multiple ownership (and can for example be used to implement iterators), unrestricted multiple ownership is not possible; objects sharing a multiply-owned object must be at the same level of the ownership tree.

Generics: A common criticism of ownership-based alias protection schemes is the annotation overhead they cause. This is particularly true in languages which already support type genericity; adding ownership types and particularly ownership genericity (like context parameters in Clarke-style ownership types) significantly increases the complexity for programmers. To address this problem, Potanin et al. present work combining ownership information with other generic type information so that both can be expressed in a single parameter space [82, 105].

Ownership inference: Another approach to reducing the annotation overhead for programmers is the development of ownership annotation inference tools. Several such tools have been developed for Clarke-style ownership types, including dynamic [99] and static approaches [65].

Dietl et al. present a tool to dynamically infer universe type annotations [33]. Their system observes a program at run time, deducing ownership relationships between objects. A static universe type inference tool which uses programmer annotations to infer ownership relationships has also been developed [31].

Effects: Several authors recognise that ownership provides a suitable base for effect systems. Clarke et al., for example, extend ownership types to add effects reporting [24]. In their system, a method is annotated with the effects that it causes; that is, the objects it reads and writes. Since ownership contexts correspond to objects, the authors use them as a basis for declaring effects.

Lu et al. propose combining ownership with an effect system to ensure that an object's invariant remains valid. Their work does not restrict read accesses, resulting in an owners-as-modifiers system [61].

Owners-as-local-dominators: Cameron et al. use the language Tribe which supports nested classes for the implementation of an ownership system [23]; they call their approach *tribal ownership*. The lexical nesting of classes defines the program's ownership structure. All instances of a class are owned by the object enclosing the class.

The authors show how various encapsulation policies, including standard owners-as-domi-nators and owners-as-modifiers can be modelled using tribal ownership. They also introduce a new encapsulation policy which they call *owners-as-local-dominators*. Owners-as-local-dominators enforces standard owners-as-dominators encapsulation for local sub-heaps instead of the entire heap.

Applications of ownership: Since being proposed, ownership types have been used for a variety of purposes, for example safe memory management in real-time Java programs [17], prevention of data races and deadlocks [14, 16], program verification [8, 34] and software visualisation [45].

For example, Boyapati et al. adapt Clarke-style ownership types to statically prevent data races and deadlocks [14, 16]; this approach is often called *owners-as-locks* [26]. Owners-as-locks is based on the idea that if a thread owns a particular object (that is, it owns the root of the ownership tree containing the object) it can safely access and modify the object since no other thread can simultaneously own the same object. Essentially, ownership acts as an implicit locking mechanism.

Owners-as-dominators completely locks an object, restricting both read and write accesses, while owners-as-modifiers represents only a partial locking mechanism. Because it does not restrict read accesses, owners-as-modifiers is problematic in the context of concurrency.

The Spec# programming language [11], an extension of C#, uses ownership to address the indirect invariant effect. Spec# supports the definition of software contracts, including method preconditions, postconditions and object invariants. It uses an owners-as-modifiers system to avoid object invariants being invalidated through aliasing references [10, 59]. Invariants must only use directly or transitively owned objects which can thus be modified only by the owner.

Spec# uses dynamic ownership, but the powerful static verifier Boogie can check many programs at compile time [9]. Dynamic ownership can easily support transfer of objects between different owners, providing additional flexibility [59].

Jacobs et al. extend Spec#'s ownership with an owners-as-locks system similar to that of Boyapati et al. to support concurrency control [50].

2.1.1.4 Module encapsulation

Rather than encapsulating objects inside other objects, we can encapsulate objects inside entire program modules, making them accessible to all other objects in the module; we call this *module encapsulation*. This type of encapsulation is provided by confined types [94] and type universes of universe types [68].

Figure 2.6 illustrates module encapsulation. An encapsulated object can be accessed only by other objects in the same module. Unencapsulated objects provide the interface to the module. We can essentially see module encapsulation as full encapsulation with a

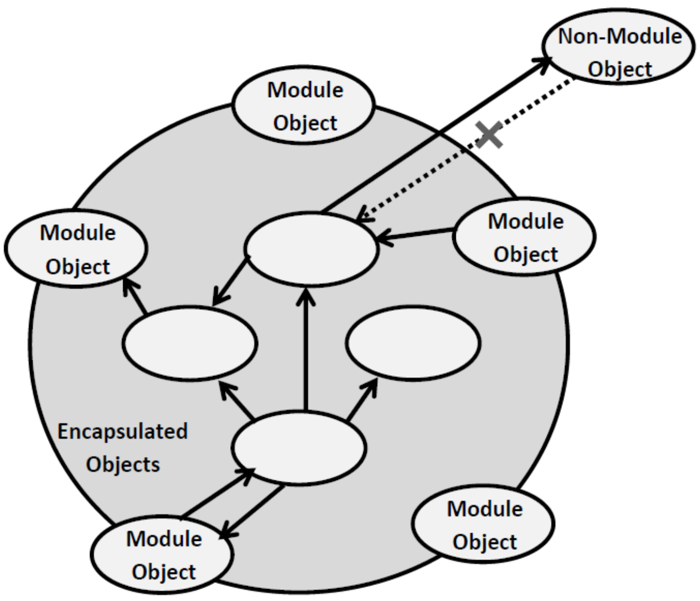


Figure 2.6: Alias protection with module encapsulation

number of bridge or balloon objects which form the connection between unencapsulated and encapsulated objects. However, a major difference between module encapsulation and full encapsulation is that references from encapsulated to unencapsulated objects are generally allowed in module encapsulation, as shown in Figure 2.6 (but such references are impossible in full encapsulation as we explained in Section 2.1.1.1).

Confined types [42, 94, 95] use module encapsulation. A type is *confined* in a domain (or module) if all instances of the type are referenced only within the domain; no references to objects of confined types may be leaked outside of the domain. The domain essentially defines a bound on where a confined object can flow. Originally presented as an informal mechanism [94, 95], confined types were later formalised using a static type system [102, 103].

Vitek et al. extend Java to add annotations for the definition of confined types [94, 95]. They use packages as domains of confinement since they are already a standard language mechanism supported by access modifiers.

Clarke et al. note that confined types require very few annotations to express aliasing properties (a Java class simply has to be annotated as `confined`) but that this comes at the cost of decreased expressiveness [26]. Confined types operate at a much coarser level of granularity than ownership-based systems [25] and do not provide encapsulation at the object level but rather at the class level [32]. All instances of a class are either encapsulated inside a package or not. In other systems, encapsulation applies separately to each object; thus, it is possible for objects of the same type to have different encapsulation and aliasing characteristics.

Universe types also include module encapsulation features. Although mainly enforcing an owners-as-modifiers aliasing policy, they also include so-called *type universes* [68] which use module encapsulation.

Every type `T` in module `M` has an associated type universe. Every object of type `T'` which is declared in the same module `M` is an owner of the type universe of `T`. Thus, objects in the type universe of type `T` can be accessed by all objects whose type is declared in module `M`.

Type universes can be used when several owner objects are required; in order to be able to share representation, the types of all owner objects must declared be in the same module.

Müller et al. argue that while module encapsulation such as that used by type universes decreases the amount of control over aliasing (for example, it cannot separate objects in two different collections of the same module) it still enables modular reasoning, since all code which can cause changes to an encapsulated object must be located in the same module [68].

2.1.1.5 Systems supporting multiple policies

So far, we have considered systems which implement one particular aliasing policy. In this section, we discuss alias protection schemes which support several different aliasing policies.

Ownership domains Aldrich et al. propose *ownership domains*, intended to be a more flexible alternative to ownership types, which is expressive enough to define common constructs such as iterators and recursive data structures [2, 57]. They can be seen as a *generalisation* of ownership types [2]. Each object can declare any number of separate domains (instead of a single implied owned context) and arbitrary links between domains can be specified. Ownership domains developed out of Aldrich et al.'s earlier work on AliasJava [3].

An important contribution of ownership domains is the separation between the ownership mechanism (determining which object owns which other objects) and the aliasing policy (determining who can access an object). These two concepts were previously tied together: ownership types establish the owner of an object and at the same time stipulate that all accesses to the object must come through the owner. Aldrich et al. argue that decoupling these mechanisms makes ownership domains both more precise and more flexible than ownership types [2].

Objects are collected in ownership domains; an object's domain is specified when it is created. There is a top-level domain called `shared` which contains objects that can be accessed globally. Links and relationships between domains may also be specified, giving one domain access rights to another domain.

Specifically, object `o1` in domain `d1` can access object `o2` in domain `d2` if an only if:

- `d1` has been given explicit access permissions to domain `d2`; or
- `d2` is the top-level domain `shared`; or
- `d1` and `d2` are in fact the same domain; or
- `o1` declares domain `d2`; or
- `o1` has access to another object which declares a `public` domain `d2` [2].

Like Clarke-style ownership types, ownership domains do not support multiple ownership (that is, each object is in exactly one domain) or ownership transfer (that is, an object stays in the same domain throughout its lifetime).

Domain access permissions are not transitive. If `Domain A` has access to `Domain B` and `Domain B` has access to `Domain C`, this does not necessarily mean that `Domain A` has access to `Domain C`.

The validity of a program with ownership domains can be verified at compile time by checking that no inter-domain accesses occur unless the appropriate permissions are in place [57].

Abi-Antoun et al. present an implementation of ownership domains for Java using annotations [1]. They use their tool to annotate two 15000-line programs and summarise their experience. The authors report several advantages of using ownership domains, including their ability to expose tight coupling. On the other hand, they criticise the lack of support for multiple ownership and ownership transfer.

Capability and permission systems In 2001, Boyland et al. introduced run-time capabilities which can model various aliasing policies [21]. A capability is an access right which one must quote to be able to indirect on a given reference. Boyland's capabilities combine each object reference with a set of access rights, specifying if the possessor of the reference has the right to access or modify the object to which the reference points.

Boyland et al.'s work includes seven different access rights. Three primitive access rights provide read access, write access and identity access (for checking the identity of an object; for example, to perform `x == y` identity capabilities for both `x` and `y` are required). Three corresponding exclusive access rights guarantee exclusive read, write and identity access, stripping incompatible access rights of other capabilities where necessary. Exclusive access rights can be seen as "negative" [21] access rights; they do not give access rights to their associated capability but deny access rights for all other capabilities. For example, a capability with the exclusive write but *without* the base write access right cannot write the associated object, while also preventing all other capabilities from doing so.

The final access right, ownership, is used to resolve conflicts between incompatible capabilities. Although identical in name, the ownership capability is semantically quite different from the concept of ownership we discussed in Section 2.1.1.3. A capability with the ownership right can strip access rights from other, incompatible capabilities; a capability which lacks the ownership right cannot remove incompatible access rights from a capability with ownership rights.

In later work, Boyland proposes *fractional permissions* [19]; a good overview and formalisation of fractional permissions is given in [20]. Whenever an alias to an object is created, permissions for the object are split into smaller fractional permissions (all adding up to one). In order to modify an object, a whole permission is required; reading an object requires only a fractional permission. Thus, read accesses can coexist but modifications are only possible if there is a single, whole permission for the object. The elegance of Boyland's work lies in the fact that this is enforced automatically as permissions are split and merged.

Haller et al. introduce compile-time capabilities which serve as names for disjoint heap regions and represent access permissions to that heap region [43]. Holding a capability gives access rights to variables in that heap region. Variables are "guarded" [43] by capabilities and can only be accessed if the capability associated with the heap region containing the variable is available. The type checker statically ensures that the required capabilities are always available when an access occurs in the program.

Capabilities and permissions can be seen as a unifying approach to alias protection; they can model various aliasing policies, including uniqueness and ownership. Uniqueness can be enforced by ensuring that, when an assignment occurs, the capability or permission to access the unique object is transferred from one variable to another; the old reference loses its access rights, ensuring that it cannot access the object in the future. This approach is quite similar to alias burying, as errors occur only when a buried reference (without capabilities) is used.

With Boyland et al.'s capabilities [21], a reference holding all 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 exclusive access rights. The reference's capabilities cannot be easily stripped away by other capabilities because it possesses the ownership right. Linearity can be modelled by stripping away all access rights every time a reference is read, making the reference unusable for future accesses. This enforces affine linearity, but cannot ensure that a variable is used *exactly* once (as is required for true linearity).

Zhao et al. show how fractional permissions can be used to encode owners-as-dominators and owners-as-locks encapsulation [104]. While it is possible to model such complex aliasing policies using capabilities and permissions, we note that this is complex and unlikely to be suitable for most software developers. This complexity limits the usefulness of capabilities and permissions as a unifying approach.

2.1.2 Other characteristics

We now consider various other properties of alias protection schemes and give examples of systems which exhibit the different characteristics.

2.1.2.1 References and object accesses

There are two distinct approaches to restricting aliasing: some systems restrict which aliases can exist; we say that they restrict *references*. An alternative approach is to allow any references to exist but limit under which circumstances they can be used to access objects; we say that such systems restrict *object accesses*. Clarke et al. use slightly different terminology to describe this, distinguishing between systems enforcing *topological restrictions* and systems enforcing *encapsulation* [26], but we find our terminology more intuitive.

Reference restrictions are commonly used in existing alias protection schemes. Clarke-style ownership types [27], Universe types [68], islands [48] and balloons [4] all take this approach. Gordon et al.'s dynamic ownership types [40] and Boyland et al.'s capabilities [21], on the other hand, are examples of systems which check the relevant access rights only when an object is accessed. Yu et al.'s work on combining ownership with an effect system also allows references to be leaked but restricts when they can be used to perform object accesses [61]. This approach makes sense in the presence of an effect system (which can be used to ensure that no illegal object accesses occur).

Although these two approaches to alias protection differ significantly, both provide the same encapsulation guarantees. While some systems prohibit the introduction of a particular reference, other systems may allow the reference to exist but later prohibit any object accesses through it, effectively rendering it useless.

2.1.2.2 Static and dynamic checking

Although some existing alias protection schemes require dynamic checking, the majority of schemes can be checked at compile time.

Systems which restrict references rather than object accesses can usually be checked statically. They achieve this by ensuring that references cannot be assigned from a variable with one set of aliasing properties to a variable with different properties. For example, an object stored in a standard variable cannot be assigned to a unique variable because we do not know how many aliases point to the object at the time of the assignment. An object in a unique variable, however, can safely be moved to another unique variable, since we know that there is only one reference to it.

Systems which restrict object accesses cannot usually be checked statically. The validity of an object access depends on the aliasing structure of the program at the time the access occurs at run time. This makes compile-time checking undecidable in the general case. Gordon et al.'s dynamic ownership types, for example, use dynamic checking [40]; Boyland et al. suggest that *some* programs involving their capabilities can be checked using static analysis, while others require run-time checks, but do not give further details about the proposed static analysis [21]. Their approach is similar to type checking in dynamically-typed languages such as Python, where some programs can be checked statically while others require dynamic checks.

Both static and dynamic checking have some advantages and disadvantages, similar to the trade-offs between static and dynamic type checking. Static checking allows errors to be discovered before the program is executed, while dynamic checking requires executing the part of the program that contains the error. Static checking also ensures that alias protection has no effect on the program's execution and does not incur run-time performance overheads.

On the other hand, static alias protection schemes are usually less flexible than dynamic schemes. They require checking to be conservative in order to discover all possible errors and report errors when there is not enough information available at compile time to prove correctness. For example, an object stored in a standard variable can be safely transferred to a unique variable if only one reference to the object exists at the time of the transfer. Static schemes would reject such an assignment as they cannot prove at compile time how many references to the object will exist. This is similar to the conservative approach used by languages with static type checking such as Java: conversions between types which cannot be proven correct at compile time generate compilation errors and require explicit type casts (which are checked only at run time).

2.1.2.3 Temporal and spatial aliasing

Mycroft and Voigt introduce a distinction between *temporal* and *spatial* aliasing [70]. Spatial aliasing is the traditional and most common notion of aliasing: it occurs when two or more references to the same object exist at the same time. This is distinct from temporal aliasing, where one reference to an object is used multiple times.

Modern OO programming languages allow both spatial and temporal aliasing: an object can be referenced by several variables and each alias to it can be used any number of times.

Many alias protection schemes, such as Clarke-style ownership types [27], islands [48] and balloons [4], limit the spatial aliasing of encapsulated objects. To enable limited sharing of objects, most systems still allow several aliasing references to encapsulated objects; thus, spatial aliasing is limited but not prohibited. No attempt is made by these systems to address temporal aliasing.

Uniqueness types disallow spatial aliasing completely by ensuring that only one reference to a unique object can exist at any time; the unique reference can be used any number of times (allowing temporal aliasing). Linear types go further and prohibit both spatial and temporal aliasing: a unique reference to an object can be used only once before it is *consumed*. Kobayashi's quasi-linear types provide a middle ground: temporal aliasing of a variable is possible only in a limited, statically determined scope.

2.1.2.4 Static and dynamic aliasing

In addition to the distinction between temporal and spatial aliasing, some aliasing protection systems distinguish between *static aliasing* and *dynamic aliasing* [4, 48, 49]. Dynamic aliasing involves stack-based variables such as local variables and method arguments; these are easy to track and relatively transient as they disappear at the end of a method's execution. Static aliasing, on the other hand, involves instance variables and is thus more permanent. Hogg argues that this makes it significantly more problematic since static aliases "can cause unpleasant surprises at an arbitrarily distant point in an execution" [48].

Some alias protection schemes allow unlimited dynamic aliasing and restrict only static aliasing. Hogg's islands [48] and Almeida's balloons [4] take this approach to lessen the restrictiveness of full encapsulation, allowing dynamic aliases from unencapsulated to encapsulated objects. While this increases flexibility [76], it also allows encapsulation to be broken temporarily by dynamic aliases, compromising the safety of these alias protection schemes [27].

In addition to standard balloon types which may be aliased dynamically, Almeida proposes *opaque balloons* which disallow both static and dynamic aliases to their internals [4]. Almeida remarks that "opaque balloons have so many 'nice' properties that it can even be argued that all balloons should be opaque" [4] but does not address this further in his paper.

Clarke et al. also introduce the distinction between static and dynamic aliases in their work on ownership types [24], allowing dynamic aliases to temporarily break the owners-as-dominators property. They argue that this makes their system more expressive, for example allowing the implementation of iterators. However, this compromises safety and requires a disciplined approach from programmers.

2.1.2.5 Sharing

Most alias protection schemes support the definition of *shared* objects, whose reference semantics are equivalent to the default aliasing policy in most languages: they can be aliased freely and used by any part of the system.

For example, ownership domains [2] provide a special domain called `shared` for objects accessible from anywhere in the system. For confined types [94], all instances of unconfined types are shared. With Boyland et al.'s capabilities [21], an object with no exclusive capability applying to it can be shared freely. Several other systems, including AliasJava [3], provide a `shared` annotation.

Sharing properties vary in different versions of owners-as-dominators and owners-as-modi-fiers. For transitive owners-as-dominators and owners-as-modifiers, all objects at the top level of the ownership tree are shared, because they can be accessed by any objects at the same or lower level of the ownership tree. This corresponds to `norep` objects in Clarke-style ownership types [27].

On the other hand, shared objects do not fit well with strict and peer owners-as-dominators and owners-as-modifiers semantics. In strict owners-as-dominators and owners-as-modifiers systems, the objects at the top of the ownership tree may not be accessed by any other object (as they have no owner). In peer owners-as-dominators and owners-as-modifiers schemes, the objects at the top of the ownership tree can access each other, but they still cannot be accessed by objects below them. As a result, top-level objects are not in fact shared among the entire system. Universe types [68] as a peer owners-as-modifiers system thus does not have globally shared objects (although all objects may be *read* from anywhere in the system).

Full encapsulation systems, such as Islands [48] and Balloons [4], also do not have globally shared objects. Encapsulated objects can be referenced only by other encapsulated objects and unencapsulated objects can be referenced only by other unencapsulated objects. As we explained in Section 2.1.1.1 references from unencapsulated to encapsulated objects are illegal and references from encapsulated to unencapsulated objects are impossible.

2.1.2.6 Borrowing

Borrowing allows temporary aliasing of objects; an object's owner can temporarily give access (or aliasing) rights for an object to another part of the system, 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 period of time. This adds flexibility to alias protection systems, by allowing a reference to be leaked to and used by another part of the system temporarily. We can see borrowing as a time-limited alternative to shared objects.

Many systems support some form of borrowing, often through annotations which may be called `borrowed`, `limited`, `temporary`, `unique`, `unconsumable` or `lent` [21].

Systems which allow unrestricted dynamic aliasing, including Hogg's islands [48], Almeida's balloons [4] and Clarke et al.'s extension to ownership types [24], implicitly support borrowing by allowing temporary (dynamic) references to protected objects.

Universe types [68] differentiate between read-only references and references with full access rights. We can see read-only references in universe types as a different kind of borrowing: they restrict what can be done with a borrowed reference, while standard borrowing restricts the duration for which a borrowed reference can be used.

2.1.2.7 Immutability

In the context of aliasing, it becomes important to distinguish between mutable and immutable objects. Immutable objects can always be aliased freely, since they cannot be modified [41, 62, 76]. This means that none of the disadvantages of aliasing described above apply to immutable objects. As MacLennan puts it, "they exhibit *referential transparency*: there is never any danger of one expression altering something which is used by another expression. Any sharing that takes place is hidden from the programmer" [62].

Immutable objects are often used to represent constant values and abstractions which never change and exist only once, such as numbers or `Strings` in Java. Mutable objects, on the other hand, are used to simulate the real world; they are created and destroyed and their state changes over time [62].

Given the significance of immutability for aliasing, many alias protection schemes provide ways of making an object or a reference immutable. However, the semantics of immutability vary.

In some systems, immutability protects the *reference*; it cannot be changed to point to another object. This does, however, not mean that the object stored in the reference cannot be changed. C++'s `const` [91], Scala's `val` [77] and Java's `final` [6] keyword produce these semantics.

Alternatively, immutability may ensure that a given reference is not used to modify the object to which it points; however, this does not necessarily protect the object from changes, which may be made via aliasing references. These are the semantics of read-only references in universe types.

In yet other systems, immutability protects the objects themselves, so that they can never be modified. The `read` annotation in Hogg's islands, for example, protects the object stored in the reference from modifications, rather than just the reference itself [48]. Scala also includes the concept of immutable objects [77]. Similar semantics are introduced by Grogono et al. [41] and MacLennan, who distinguishes between *values*, which are immutable, and *objects*, which are mutable [62].

Boyland et al.'s capabilities [21] can be used to model the last two semantics of immutability. A capability with no write access right cannot modify an object. However, this does not prevent the object being modified through other aliases. On the other hand, a capability holding an exclusive write access right but no base write protects the object to which it points from modification. It cannot modify the object to which it points itself since it does not have the base write access right; at the same time, the exclusive write access right ensures that the object cannot be modified through other references.

2.1.3 Limitations of existing work

As our discussion above shows, there is a huge variety of alias protection schemes with many different properties, every one of them with its own advantages and disadvantages.

However, there is little empirical software engineering research about which alias protection properties are actually useful. Any empirical research that exists is small in scale. We know of no studies conducted with actual software developers. Instead, most studies are undertaken by the authors themselves or university students on small systems or example design patterns. As a result, we really have no idea what software developers want or need in terms of alias protection.

Another problem is the lack of a unifying system or theory. While capabilities, which can be used to model several different alias protection policies, represent a small step in this direction, their approach is very low-level. Modelling complex encapsulation policies such as transitive owners-as-dominators is difficult. While possible in theory, we cannot envisage actual software developers doing this.

Many existing alias protection systems are too inflexible for practical use. For example, sharing between encapsulated and unencapsulated objects is often difficult to implement. Most ownership-based systems lack support for ownership transfer and multiple ownership which is essential for modelling common idioms such as iterators. We suggest that when encountering such issues, software developers may "solve" the problem by simply reverting to using shared objects (with standard reference semantics), such as `norep` objects in Clarke-style ownership types.

Although authors have developed systems which support multiple ownership and ownership transfer (and allow some iterator implementations, for example), many of these systems feel like "work-arounds" rather than solutions to the underlying problem. Any true implementations of ownership transfer and multiple ownership results in complex type system, which software developers are likely to avoid. Dynamic approaches appear to be significantly better at tackling these issues.

From a programmer's point of view, many of the systems are complex to use. Ownership-based systems, for example, require developers to carefully structure their program to fit the prescribed, hierarchical ownership structure. This rigid structure is also likely to make refactoring and redesign difficult. Many alias protection schemes additionally impose a significant conceptual and annotation burden on programmers.

Given these limitations of existing work, it is not surprising then that alias protection schemes have not been widely adopted in practice.

2.2 Alias and pointer analysis

Alias and pointer analysis attempts to determine the values of memory locations and thus their aliasing relationships at run time. We can distinguish between three different types of analysis – alias, pointer and points-to analysis – although the boundaries between the three are fluid and terminology is sometimes used interchangeably in literature [46].

Alias analysis tries to discover which variables may at run time contain the same value (and may thus be aliased). In this work, we are mainly interested in this kind of analysis.

Pointer analysis takes a broader approach and tries to find possible pointer values that can occur at run time. This information can subsequently be used to perform alias analysis.

Points-to analysis determines which variables point to each other; for example $x \rightarrow y$ means that variable x points to variable y. Like pointer analysis, points-to analysis can be used to perform alias analysis: two variables are aliased if there is a points-to path from one to the other. Despite the close relationship between alias and points-to analysis, Khedker et al. note that the points-to relation is fundamentally different from the alias relation [53]: unlike points-to, aliasing is a symmetric relation.

Alias analysis is used for alias detection, as described by Hogg et al. [49] in *The Geneva convention on the treatment of object aliasing*. Detecting aliasing is arguably a first step to addressing the negative effects it causes. Alias analysis forms a base for many different kinds of static analyses, including compiler optimisation, program verification, parallelisation of sequential programs and compile-time GC.

The problem of alias (and pointer and points-to) analysis is undecidable in the general case; therefore, any solution must be an approximation. Many different analyses have been proposed. Approaches to the problem vary widely as we describe below and, as a result, alias analysis theory and tools exhibit a wide range of performance and precision. These two attributes conflict with each other: highly precise analyses have high asymptotic complexities (up to doubly exponential), whereas other analyses achieve near-linear performance but significantly lower precision.

We can categorise existing alias, pointer and points-to analysis work using a number of different properties; some of these categorisations are taken from Hind [46]:

- **May-aliasing and must-aliasing**: May-aliasing describes which variables may be aliased at run time; if no may-alias exists between two variables, we know for sure that no aliasing is possible between them at run time. However, we cannot in general deduce that two variables will definitely be aliased at run time from may-aliasing information. Must-aliasing shows which variables will certainly be aliased at run time. The absence of a must-alias between two variables, however, does not indicate that no aliasing is possible at run time.

  Sridharan et al. [88] note that may-aliasing is an *over-approximation*, as it assumes that aliasing may exist if it cannot prove otherwise. The simplest possible such over-approxi-mation is to assume that all variables of matching types are aliased. We can see all other may-aliasing work as an attempt to improve on this base approximation [88].

  Most existing work concerns may-aliasing, including work by Steensgaard [90] and Andersen [5]. Work by Emami et al. [35] computes both may-aliasing and must-aliasing. Depending on the domain, one type of analysis may be more useful than the other. For example, Sridharan et al. argue that may-aliasing often reports many false positives and must-aliasing is therefore more useful [88]. However, many applications of alias analysis, including compiler optimisations, require conservative assumption, making may-aliasing more suitable.

- **Flow-sensitivity**: Flow-sensitive analyses (for example work by Emami et al. [35] and Landi et al. [58]) compute one solution for each program point, while flow-insensitive analyses (for example Steensgaard's [90] and Andersen's [5] analyses) compute one solution for each method or even for the entire program. For example, flow-sensitive analyses distinguish between $A; B$ and $B; A$, while flow-insensitive analyses treat both as equivalent. This reduces the precision of flow-insensitive analyses but also significantly improves their performance.

- Flow-insensitive analyses may be either **equality-based** (for example the analysis proposed by Steensgaard [90]) or **subset-based** (for example Andersen's work [5]). Equality-based analyses consider assignment statements as bidirectional. This means that the two assignment statements `y = x; z = x;` will result in the analysis reporting a possible alias between `y` and `z`, although such an alias clearly cannot exist at run time. Subset-based analyses consider assignments to be unidirectional and can therefore detect that in the above example no aliasing between `y` and `z` is possible.

- **Context sensitivity**: Context-sensitive analyses (for example Emami et al.'s [35] and Landi et al.'s [58] work) take into account the calling context when a procedure is invoked. This allows them to distinguish between different invocations of the same method which may create different aliases. Again, this is clearly more exact but also more expensive than context-insensitive analyses (for example Steensgaard's [90] and Andersen's [5] analyses).

- **Field sensitivity**: OO alias analyses may be field-sensitive or field-insensitive [88]. Field-insensitive approaches merge the values of fields across different objects, while field-sensitive approaches distinguish between them. Sridharan et al. claim that field sensitivity is not very expensive (as opposed to flow-sensitivity and context-sensitivity which incur large performance penalties) [88].

- **Alias representation**: There are three main approaches to representing aliasing information. All aliases may be represented explicitly, including derived aliases (inferred from other aliases in the program). For example, the two assignment statements `y = x; z = y;` create aliasing between `x` and `z`. Analyses using *explicit* alias representation (for example Landi et al.'s work [58]) would record the variable pairs (x, y), (x, z) and (y, z) to show that all three variables may be aliased with each other. A tool using *compact* alias representation would record only (x, y) and (y, z) and derive the third pair when needed.

  A third possible representation, called *points-to* representation (for example used by Emami et al. [35]) records which variables point to which other variables. For the example above, a tool using this representation would record $(y \rightarrow x)$ and $(z \rightarrow y)$, which can easily be converted to explicit or compact alias representations [35].

- **Referring to memory locations**: Alias analyses need a way of referring to specific memory locations (or objects in OO programming languages). For example, Landi et al. use *object names* (an expressions consisting of a variable followed by a series of field accesses, for example `v1.v2.v3`) to refer to different memory locations [58]. Recursive data structures are problematic because they create an infinite number of possible object names.

  Many analyses simply treat a recursive data structure as a single object without distinguishing between individual sub-objects. This allows the analyses to limit the number of possible memory locations and object names to a finite number. Deutsch calls this a *store-based* approach and notes that it is quite inaccurate [30].

  Landi et al. propose *k-limiting* [58]. They introduce a constant k which is the maximum number of variable accesses an object name is allowed to contain. Although more accurate than store-based approaches, k-limiting also conflates sub-objects to a single object below depth k.

  Deutsch proposes a pointer analysis which uses *access paths* to represent memory locations [29]; these are roughly equivalent to Landi et al.'s object names. In a follow-up paper, he proposes *symbolic access paths* which can represent repetitions in an access path caused by recursive data structures [30]. For example, the access path `a.b.a.b.c` is represented as $(a \rightarrow b \rightarrow)^2 c$. This allows more detailed representation of aliasing, particularly for recursive data structures. For example, Deutsch notes that his analysis can represent the fact that "the ith element of list X is aliased to element $2i + 1$ of list Y" [30].

- **Modular and non-modular analysis**: Some alias analyses require the code for the entire program, while others can work with only parts of the program as input. Sridharan et al. note that partial program analysis is particularly important for programs written in modern programming languages such as Java which include huge libraries in addition to programs' source code [88]. Whole program analysis must also consider all libraries used by the program, leading to a large increase in the amount of code to be analysed.

The work by Steensgaard [90] and Andersen [5] is particularly well-known. Analyses with similar characteristics are often referred to as *Steensgaard-style* and *Andersen-style* analyses in literature [46].

Steensgaard proposes a flow-insensitive, context-insensitive and equality-based pointer analysis with linear space and near-linear time complexity [90]. The analysis uses a non-standard type system to infer the possible values of pointers. Although less precise than other analyses, its low asymptotic complexity makes it applicable even to large programs.

Andersen's approach [5] uses a constraint solver. It is flow-insensitive, context-insensitive and subset-based, making it both more precise and more expensive than Steensgaard's work.

2.3 Summary

In this chapter, we presented the background to the remainder of this work. In particular, we compared and contrasted a wide range of existing alias protection schemes, highlighting their similarities and differences and discussing common issues. We also reviewed important work in the area of alias and pointer analysis.

CHAPTER 3

Aliasing contracts: overview and semantics

In this chapter, we introduce a dynamic alias protection scheme called *aliasing contracts*. Aliasing contracts aim to prevent unintended reference leaks and thus address the problems of aliasing, including representation exposure. They are highly flexible and expressive, overcoming the shortcomings of existing alias protection schemes described in Chapter 2, while at the same time remaining conceptually simple.

Aliasing contracts allow developers to express assumptions about which parts of a system can access particular objects. Unlike many existing alias protection schemes, 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*.

We complement aliasing contracts with the concept of *encapsulation groups* which allow objects to be grouped and permission to access an object 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.

The name *contract* comes from work on software contracts [63]; these support the specification of preconditions and postconditions for methods. Aliasing contracts essentially specify preconditions for object accesses; they behave like assertions, reporting errors at run time. Like assertions and software contracts, we intend aliasing contracts to be used primarily as a testing and debugging tool to identify unexpected aliasing.

In Section 3.1 of this chapter, we give an informal description of aliasing contracts; we then formalise them in Section 3.2, giving a syntax and operational semantics. Finally, we present a case study in Section 3.3 to demonstrate how aliasing contracts can be applied in practice.

3.1 Overview of aliasing contracts

Aliasing contracts express and enforce restrictions about the circumstances under which an object can be accessed. We start by defining what exactly we mean by *object access*, before explaining the details of aliasing contracts.

In OO programming, objects are at run time stored on the program's heap. We say that an object is accessed when the region of the heap representing the object is read or written. This for example occurs when an object's fields are read or written: the field read `v.f` represents an access to the object referenced by `v`[1]; it is a lookup of the value stored in field `f` in the heap region representing the object stored in `v`. Field read `v.f` is not, however, an access to the object referenced by `v.f`. Although its address is looked up, the heap region associated with the object in `v.f` is not accessed when executing the expression `v.f`.

In addition to field reads and writes, we also consider method calls `v.m()` as object accesses to the object in `v` (although the access to the heap region representing the accessed object is arguably more indirect in this case)[2].

Local variables are stored on the stack; thus, reading or writing a local variable does not represent an object access – the heap is never read or written. For example, if `x` is a local variable, then expression `x` is not an object access (only a stack access), but (as before) `x.f` is an access to the object referenced by `x`.

While method calls are classed as object accesses, constructor calls are not. Constructor calls modify the heap to create a new object, but they do not directly involve regions of the heap containing *existing* objects[3].

Now, we describe in detail the semantics of aliasing contracts. Each variable (field, method parameter or local variable) declaration in a program is annotated with an aliasing contract, which defines under which circumstances *the object referenced by the variable* can be accessed. An aliasing contract consists of two side-effect free boolean expressions: the read contract expression e^r specifies under which circumstances object reads are legal, while the write contract expression e^w concerns object writes. (Where the read and write contract expressions of a contract are the same, one can be omitted for convenience; we call such a contract a *rw-contract* (read-write-contract).) Reading an object's fields is an object read and therefore the object's read contracts are evaluated; writing an object's fields is an object write and causes the evaluation of the object's write contracts. We further distinguish between pure and impure methods: a call to a pure method is an object read and requires the evaluation of the object's read contracts, while a call to an impure method is both an object read and write and requires the evaluation of both read and write contracts.

Due to aliasing, multiple variables may point to the same object; thus, an object may have multiple contracts associated with it. Access to an object is allowed only if *all* of the contracts of variables pointing to the object are satisfied. Whenever an object is accessed, all contracts are evaluated and the evaluation results are conjoined. If the result is `true`, the object access is legal; otherwise, an exception is thrown to report the contract violation. We can thus see contracts as *preconditions* for object accesses, protecting not the variable to which they apply but the object stored in the variable.

In Chapter 1, we used a `LinkedList` example to demonstrate how references to internal objects can be leaked unintentionally, leading to encapsulation breaches. We now review this example to see how aliasing contracts can be used to address this problem. Specifically, we want to ensure that `Nodes` in the `LinkedList` cannot be accessed from outside the `LinkedList`. We use a Java-like language for the example, with contracts

[1]Syntax may allow a field read `f` without a preceding term `v`. This is of course simply shorthand for `this.f` and thus constitutes an access to the object referenced by `this`.

[2]As above, `m()` is equivalent to `this.m()` and represents an access to the object in `this`.

[3]Object accesses may of course occur *inside* the constructor body whose statements may read or write other heap regions; we consider these object accesses separately from the constructor invocation.

enclosed in curly braces following variable declarations; a formal syntax for aliasing contracts is given in Section 3.2.1 below. In this first example, we underline contracts to make them easier to identify:

```
class LinkedList {
  private Node head {accessor == accessed || accessor == this};

  public impure void addItem(Object data {true}) {
    Node newNode {true} = new Node(data);
    newNode.setNext(head);
    head = newNode;
  }

  public pure boolean member(Object data {true}) {
    for(Node n {true} = head; n != null; n = n.getNext()) {
      if(n.getData().equals(data)) {
        return true;
      }
    }
    return false;
  }

  public pure Node getHead() {
    return head;
  }
}
class Node {
  private Node next {accessor == accessed || accessor == this};
  private Object data {true};

  public Node(Object data {true}) {
    this.data = data;
  }

  public pure Node getNext() {
    return next;
  }

  public pure Object getData() {
    return data;
  }

  public impure void setNext(Node next {true}) {
    this.next = next;
  }
}
```

The aliasing contracts for `head` and `next` are of particular interest and we now explain their exact meaning:

```
//In LinkedList
Node head {accessor == accessed || accessor == this};
```

```
//In Node
Node next {accessor == accessed || accessor == this};
```

Aliasing contracts bind two special variables, `accessor` and `accessed`: `accessed` points to the accessed object; `accessor` points to the object making the access. For example, for an object access inside a method body, `accessor` points to that method's `this` object. The value of `accessor` is determined immediately prior to contract evaluation and so, for a given contract, may vary from one evaluation to the next.

Each contract is evaluated in the context of its *declaring object*; that is, the object which contains the variable that declares the contract. For example, the above contract for `head` is evaluated in the context of the `LinkedList` object. Contract expressions must be rooted in the declaring object. This means that contracts declared in methods may not refer to local variables and method parameters, only to fields and methods of the declaring object. During contract evaluation, `this` points to the declaring object. Thus, we can alternatively view a contract as a boolean method of the declaring class with `accessor` and `accessed` as parameters; for example, the contract for `head` becomes a method in `LinkedList`:

```
public boolean headContract(Object accessor, Object accessed) {
  return accessor == accessed || accessor == this;
}
```

Thus, in the above linked list implementation with aliasing contracts, `Node` objects stored in `head` can be read and written only by themselves ("`accessed`") and the `LinkedList` ("`this`"), while `Node` objects stored in `next` can be accessed by themselves ("`accessed`") and the previous node in the list ("`this`"). Other parts of the system can obtain a reference to `Node` objects stored in `next` and `head` using the `getNext` and `getHead` methods, but would be unable to use it. This prevents a client from modifying the list by calling `list.getHead().setNext(new Node())` for example (as this would cause a contract violation).

In addition to the contracts specified for `next` and `head`, we give all other variables the rw-contract "`true`". This means that we do not impose any restrictions on accesses to objects stored in those variables.

Aliasing contracts gain much of their flexibility through *dynamic* alias checking. The validity of an object access depends on the aliasing structure of the program at run time; a change to that structure can change the result of a contract evaluation. For example, a client could obtain a reference (unusable at first) to the `head` node in the `LinkedList` by calling `list.getHead()`; if that node is subsequently removed from the `LinkedList` (and thus the encapsulating contract disappears), the client gains access to the node. This example also shows that aliasing contracts do not restrict aliasing itself but only object accesses. It is legal for any object to hold a reference to encapsulated nodes; however, this reference can only be used when all aliasing contracts are satisfied.

In some situations, we may want to check whether or not we have access to a particular object *before* performing an access to ensure that we do not cause a contract violation at run time. To this end, we define two binary boolean operators, `canread` and `canwrite`, which check if one object can access another one:

```
if(this canwrite foo) {
  foo.bar = new Bar();  //Perform the write access
}
```

The canread and canwrite operators are also useful in contract specifications. For example, we can modify the contract for the next field in the LinkedList example as follows:

```
class Node {
  Node next {accessor == accessed || accessor canread this,
      accessor == accessed || accessor canwrite this};
  ...
}
```

When a Node object stored in next is accessed, this contract first checks if the access comes from the node itself ("accessed"). If this is true, the contract evaluation returns true due to the lazy semantics of the || operator. Otherwise, it will check if accessor can read or write the previous node in the list – the declaring object of the evaluated contract ("this"). In this way, evaluation continues until the head of the list is reached; at this point, the contract of the head field in LinkedList is evaluated. As explained above, this contract evaluates to true if the access comes from the LinkedList or from the head node. Therefore, the above contract evaluates to true if the access to a node comes from the LinkedList or *any previous* nodes in the list. It is important to note that the values of accessed and this change every time evaluation moves to a contract's declaring object, while accessor remains the same.

3.1.1 Encapsulation groups

Although we can specify a number of different contracts with the basic operators introduced above, some contracts remain cumbersome and difficult to express. Consider the simple example of a Car with four Wheels, where all wheels should be able to access each other. We can write this contract as follows:

```
class Car {
  Wheel wheel1 {accessor == accessed || accessor == wheel2
      || accessor == wheel3 || accessor == wheel4};
  Wheel wheel2 {accessor == accessed || accessor == wheel1
      || accessor == wheel3 || accessor == wheel4};
  Wheel wheel3 {accessor == accessed || accessor == wheel1
      || accessor == wheel2 || accessor == wheel4};
  Wheel wheel4 {accessor == accessed || accessor == wheel1
      || accessor == wheel2 || accessor == wheel3};
```

Clearly, this is highly impractical. Imagine if we have ten fields that need to have mutual access to each other, or even worse, if we also want to give access to directly *and* transitively contained objects as in transitive ownership schemes, such as Clarke-style ownership types. For example:

```
class Car {
  Wheel wheel1 {accessor == wheel1 || accessor == wheel1.tyre
      || accessor == wheel2 || accessor == wheel2.tyre || ...};
```

In some cases, we may not know how many transitively contained objects there are at run time. This is particularly the case for recursive data structures.

Let us reconsider our earlier LinkedList example; above, we specified contracts which encapsulate all Node objects inside their LinkedList; a node could only be accessed by the LinkedList and previous nodes in the list. In order to achieve more flexibility, we may want to allow any nodes to access each other within a single linked list. We could try to write:

```
Node head {accessor == this || accessor == head
    || accessor == head.next || accessor == head.next.next || ...};
```

Not only is this contract less than ideal from an information hiding and encapsulation point of view (the LinkedList should only really know about the head node, not subsequent nodes in the list), it also does not work, since we do not know how many nodes will be in the list at run time. This number is unbounded and is likely to change during the program's execution.

Although the canread and canwrite operators can deal with unknown and changing numbers of objects in the LinkedList, we cannot use them to specify this contract – they are not flexible enough. Using canread and canwrite, we can only model contract evaluation which proceeds either backwards or forwards through the list, but not in both directions at the same time. For example, with the contract "accessor canread this, accessor canwrite this" evaluation moves from a node to the previous node in the list; with "accessor canread next.next, accessor canwrite next.next" evaluation moves from a node to the next node in the list[4]. Attempts to combine the two (for example as "accessor canread this || accessor canread next.next, accessor canwrite this || accessor canwrite next.next") lead to cyclic contract evaluation.

To address these problems, we introduce *encapsulation groups*, which allow us to group together several objects and refer to all objects in the group instead of just single objects.

The example below shows how we can rewrite the contracts in the Car class above. We create an encapsulation group called wheels in Car which contains all four wheels of the Car. We then use the in operator in the contracts of the four wheels to check if accessor is in the wheels encapsulation group (that is, if accessor is one of the four wheels):

```
class Car {
  group wheels = {wheel1, wheel2, wheel3, wheel4};
  Wheel wheel1 {accessor in wheels};
  Wheel wheel2 {accessor in wheels};
  Wheel wheel3 {accessor in wheels};
  Wheel wheel4 {accessor in wheels};
}
```

The real power of encapsulation groups lies in the fact that they can contain an unlimited number of objects, which may vary at run time. Encapsulation groups can also contain other, nested groups; if g_1 contains another group g_2, the contents of the g_2 are added to g_1. In this way, encapsulation groups support the specification of deep and transitive contracts.

We can, for example, use encapsulation groups to specify contracts for the LinkedList, giving both the LinkedList and all nodes access rights to every node in the list. This is possible even when we do not know how many nodes there will be at run time. First,

[4]This contract is evaluated when the object in next is accessed, so to proceed to the next node in the list, we need to evaluate the contract of next.next. The contract "accessor canread next, accessor canwrite next", on the other hand, leads to cyclic evaluation, where we always evaluate the contracts of the same node.

we create an encapsulation group `nextNodes` in `Node` that contains all *following* nodes in the list as shown below; we declare a second encapsulation group called `allNodes` in `LinkedList` which contains *all* nodes in the list:

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

Now that we have an encapsulation group containing all nodes in our list, we can use the `in` operator as follows:

```
//In LinkedList
Node head {accessor == this || accessor in allNodes};
//In Node
Node next {accessor canread this, accessor canwrite this};
```

When a node in the list is accessed, contract evaluation succeeds if `accessor` is either the enclosing `LinkedList` object or one of the `Node` objects in the `LinkedList`. This example shows how aliasing contracts can easily specify complex aliasing conditions.

We call a series of field selections such as $f_1.f_2.f_3$ an access path. 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 expression E at run time is a member of those objects reachable from an access path in g starting from object E'. Syntactically, groups are specified by a regular grammar (see *groupDef* in our syntax in Section 3.2.1).

Like contracts, encapsulation groups are evaluated at run time in the context of their *declaring object*. The expressions in the group must thus be valid in the context of that object. While the paths specified by groups are syntactically defined to be rooted in `this`, $E'.g$ is evaluated at run time starting from the object resulting from expression E'.

Note that "`this`" in groups plays the role of empty path, so that in the example above `next.nextNodes` could equivalently be written `this.next.nextNodes`.

Encapsulation groups have simple semantics but naturally result in cycles which complicates the implementation of the `in` operator. Any implementation must ensure that cycles are evaluated only once and do not lead to indefinite looping.

For example, cyclic encapsulation-group evaluation will occur in a circular linked list implementation if we define the encapsulation groups in the same way as we did for `LinkedList` above. The `next` field of the last `Node` in a circular list simply points back to the first `Node`; therefore, evaluation of the `nextNodes` encapsulation groups will be cyclic. An implementation must detect such a cycle and return `false` if it has not found the object it is looking for after the first traversal of the cycle.

3.1.2 Contract parameters

In their work on ownership types, Clarke et al. introduce context parameters to enable owner polymorphism [27]. In this section, we present *contract parameters*, which are dynamic analogues of Clarke et al.'s context parameters; we say that they enable *contract polymorphism*.

Implementations of collections are an example where contract parameters are particularly useful. In our `LinkedList` example, we have so far used the rw-contract "`true`" for the data stored in each Node. This means that the data is not encapsulated by the LinkedList. However, depending on the exact usage of the collection, we may want to specify different encapsulation conditions for the data it stores. We can use contract parameters to achieve this.

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 be provided; we say that the contract parameters are *instantiated*. We now show how contract parameters can be used to make LinkedList contract polymorphic:

```
class LinkedList<cp> {
  private Node head {accessor == accessed || accessor == this};

  public impure void addItem(Object data {true}) {
    Node newNode {true} = new Node<cp>(data);
    newNode.setNext(head);
    head = newNode;
  }

  public pure boolean member(Object data {true}) {
    for(Node n {true} = head; n != null; n = n.getNext()) {
      if(n.getData().equals(data)) {
        return true;
      }
    }
    return false;
  }

  public pure Node getHead() {
    return head;
  }
}

class Node<cp2> {
  private Node next {accessor == accessed
    || accessor instanceof LinkedList};
  private Object data {<cp2>};

  public Node(Object data {true}) {
    this.data = data;
  }

  public pure Node getNext() {
    return next;
  }

  public impure void setNext(Node next {true}) {
    this.next = next;
  }

  public pure Object getData() {
    return data;
    }
  }
```

The `LinkedList` class now takes a single contract parameter called `cp`. When it creates a `Node` to store `data`, it passes this contract parameter to the `Node`. The `Node` class also takes a contract parameter, called `cp2`, which it uses to specify the contract for its `data` field. In this way, the contract parameter is passed from the client creating the `LinkedList` to all of the nodes in the list and used there to specify the aliasing properties of `data`.

We can now create a `LinkedList` in class `Client` as follows:

```
class Client {
  LinkedList list = new LinkedList<accessor == accessed
      || accessor == this || accessor == list>();
}
```

This instantiates a `LinkedList` with contract parameter “`accessor == accessed || accessor == this || accessor == list`”. This parameter is passed on to all nodes created by the list and subsequently used by the nodes to protect the `data` they store. This means that all `data` in the list is accessible to itself (“`accessed`”), the `Client` object that constructed the list (“`this`”) and to the `LinkedList` object itself (“`list`”). By varying the contract parameter provided when `LinkedList` is constructed we can specify a wide range of encapsulation conditions for the data stored in the list.

It is important to note that in the presence of contract parameters, a contract’s declaring object is no longer the object which declares the variable to which the contract is attached, but the object which provides the instantiations for the contract parameters. In the example above, when the `data` of a node in `list` is accessed, the contract “`accessor == accessed || accessor == this || accessor == list`” is evaluated in the context of the `Client` object that created `list` and instantiated the contract parameter; the declaring object of this contract is the `Client` object which instantiated it, not the `Node` which uses it as a contract for its `data` field (as would be the case with standard, non-parameterised contracts).

3.1.3 Contract suspension

There are situations, where we want to pass a well-protected object (with a strong contract) to a trusted part of the system, giving away *temporary* enhanced access to the object. This is called borrowing in existing work, as explained in Chapter 2. A common form of borrowing occurs when a method is given access to an encapsulated object (that it could not normally access) for the duration of its execution.

Aliasing contracts support the *suspension* of contracts which can be used to model borrowing. A field, local variable or method parameter contract can be suspended temporarily and reinstated at a later point in the program’s execution. While a contract is suspended, it will not be evaluated when the associated object is accessed.

The example below gives the syntax for temporarily suspending the contract of a variable `z` using a `suspend`-block. The contract is automatically reinstated at the end of the block:

```
Object z {accessor == accessed || accessor == this};
...
```

```
suspend this.x.y.z {
  foo.bar(this.x.y.z);
}
```

Method `foo.bar` can access the object in `this.x.y.z`, because the contract of variable `z` is suspended. The `suspend`-block makes it clear which contracts are suspended at which points of the program. They also ensure that a programmer cannot simply forget to reinstate a contract.

Contract suspension can be simulated using temporary variables with rw-contract "`true`" and assignment statements; for example, the `suspend`-block in the example above can equivalently be modelled as shown below. We use this in our semantics for aliasing contracts presented in Section 3.2.1, modelling contract suspension as syntactic sugar:

```
Object temp {true} = this.x.y;    //A
Object temp2 {true} = temp.z;     //B
temp.z = null;                    //C
foo.bar(temp.z);
temp.z = temp2;                   //D
```

In this desugared version, the contract of variable `z` is suspended at program point `C` and reinstated at program point `D`.

The desugaring clarifies which access rights are required in order to suspend a contract: we need both read and write access rights for the object which contains the suspended contract, the "owner" of the contract. In the example above, we need read access rights to the object referenced by `temp` (or equivalently `this.x.y`) at program point `B` and write access rights to `temp` at program points `C` and `D`. This is important to ensure that contracts cannot be suspended (and thus circumvented) arbitrarily. We also need read access rights for `this` and `this.x` at program point `A` (but not afterwards[5]).

In some cases, we can simplify the desugaring to use only a single variable. This is possible if the suspended expression contains at most one field selection. (Otherwise, a second temporary variable is useful to ensure that we can reinstate the contract later.) For example, `suspend this.z` becomes:

```
Object temp {true} = this.z;
this.z = null;
foo.bar(temp);
this.z = temp;
```

Suspension of a variable contract applies only to a single object; the contracts of the corresponding variables of other objects of the same type remain in place. In the example above, suspending the contract of `z` in one object has no effect on the contracts of `z` in other instances of the same type. The same is true when suspending local variable contracts during method execution; contracts of corresponding variables in recursive stack frames of the same method are unaffected.

In order to simplify the semantics of contract suspension, we stipulate that a variable may not appear on the left-hand side of an assignment while its contract is suspended. This means that the variable must point to the same object throughout the `suspend`-block, avoiding a transfer of the suspended contract to another object. This is much simpler to understand and reason about in practice, because, when the contract is reinstated, it will apply to the same object as before.

[5]This avoids problems reinstating the contract if we lose access rights to `this` or `this.x` during the `suspend`-block.

3.1.4 Checking aliasing contracts

Since aliasing contracts depend on the dynamic aliasing structure of a program at the time an object access is made, they cannot be checked statically in the general case. Instead, the semantics we present here evaluates them at run time, before an object access is allowed to proceed; if a contract violation is detected, an error is reported. We consider static checking of aliasing contracts in detail in Chapter 6.

Accesses to objects within contracts 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 contracts; for example consider:

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

The contract states that `String str` can only be read and written when its length is larger than zero. However, the contract itself requires an object read for `str` which leads to another evaluation of `str`'s contract, which in turn causes another evaluation and so on. Our semantics simply loops by recursively evaluating the contract; implementations of aliasing contracts may prefer to check for such cycles.

3.2 Formalisation of aliasing contracts

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

3.2.1 Syntax

A syntax for aliasing contracts is given in Figure 3.1. C ranges over class names, m over method names, f over field names, g over encapsulation group names, x over variable (field, local variable and parameter) names and cp over contract parameter names. τ denotes a type, either a class type or `bool`. For brevity, we write, for example, $\overline{\mathit{fieldDef}}$ for $\mathit{fieldDef}_1 \ldots \mathit{fieldDef}_n$.

A program consists of a number of class definitions, each of which contains field, group, constructor and method definitions. Each class may also declare 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; these expressions must be valid in the context (scope) of the nearest enclosing class (that is, the declaring object at run time). Therefore, they cannot refer to method parameters and local variables (but can refer to `this` and thereby to fields and methods). We distinguish them from standard expressions E to highlight this difference.

The keywords `this`, `accessor` and `accessed` behave just like user-defined variables, but do not themselves have contracts, nor may they be updated. The variable `this`

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

Figure 3.1: A simple syntax for aliasing contracts

is available in every execution context and points to the current object; `accessor` and `accessed` are logically λ-bound only within aliasing contracts.

Although our syntax does not explicitly support loop definitions, these can be modelled using tail-recursion.

We impose several additional syntactic restrictions on programs:

- Contract expressions must have boolean type.
- The keywords `accessor` and `accessed` and the `in` operator can be used only in contracts.
- The expressions used in contracts cannot have side effects; that is, variable assignments and calls to impure methods are not allowed.

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 operational semantics objects live forever and thus their fields are also never deallocated. This is not problematic in the semantics (although it may not be practical for implementations), since contract evaluation is monotonic with respect to GC (that is, object deallocation). Deallocating an object (and therefore removing the contracts of its fields from the objects to which they point) can never make a contract evaluation fail which would otherwise have succeeded.
- 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.

Although included in the syntax, our operational semantics presented below does not mention `suspend`-blocks; these are desugared into simple assignments as described in Section 3.1.3 above.

3.2.2 Operational semantics

Figures 3.2 to 3.5 present the reduction rules of our operational semantics. In the following sections, we first introduce the necessary notation, before explaining the reduction rules in more detail.

3.2.2.1 Notation

We define our operational semantics in the context of the current machine state consisting of a *stack* Δ, a *heap* Ψ and a *contract store* Λ which tracks the aliasing contracts for each object. We use $\rightsquigarrow$ to denote reduction. The different components of our machine state are described in this section.

Key-value maps Our operational semantics tracks information about the current machine state, including objects and contracts. We organise this information into maps of keys and values. $M(k)$ represents a lookup of key k in map M. $M[k \mapsto v]$ represents an update of the value associated with key k in map M to value v. We define update semantics as usual: $M[k \mapsto v](k) = v$ and $M[k \mapsto v](k') = M(k')$ if $k' \neq k$.

We use angle brackets to indicate tuples and use $\downarrow_i$ to select the ith element of a tuple. Thus $\langle x_1, \ldots, x_n \rangle \downarrow_i = x_i$.

Types and values We use τ to denote types, either the primitive type `bool` or a class type. A variable of type `bool` contains a boolean value b, which may be either `true` or `false`. A variable of any other type contains an address a, which must 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 an address a.

For convenience, we define the function $init(\tau)$ to give the initial value for a variable of type τ: `false` for variables of type `bool` and `null` for all other types.

Syntactic contracts We use ϕ to refer to *syntactic contracts* which can either consist of two contract expressions, e^r and e^w, or a contract parameter, cp. If ϕ contains two contract expressions, $\phi = \langle e^r, e^w \rangle$. When ϕ is parameterised, we record the name of the contract parameter surrounded by angle brackets to distinguish it from non-parameterised contracts: $\phi = {<}cp{>}$.

Compile-time dictionaries In order to simplify the lookup of field, method, contract and contract parameter information in our operational semantics, we define the compile-time dictionaries $\mathcal{F}$, $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{CP}$. The information in these dictionaries can easily be determined statically from a program's source code.

For each class C, $\mathcal{F}^C$ maps each field name f onto the field's type τ: $\mathcal{F}^C = \{f \mapsto \tau\}$. Similarly, for each class C, $\mathcal{M}^C$ is a mapping from method m to a tuple containing the method's parameters x and their types σ, the method body E, the method's local variables

y and their types τ, and the method's purity π (`true` or `false`): $\mathcal{M}^C = \{m \mapsto \langle\{x \mapsto \sigma\}, E, \{y \mapsto \tau\}, \pi\rangle\}$.[6]

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

For each variable (field, parameter or local variable) x in class C, $\mathcal{C}^C$ contains the syntactic contract ϕ associated with the variable declaration: $\mathcal{C}^C = \{x \mapsto \phi\}$. For simplicity, we assume here that each variable name is unique inside a class.

For class C, $\mathcal{CP}^C$ contains a set of the class' formal contract parameter names: $\{cp_1, \ldots, cp_n\}$.

The stack Δ The stack Δ stores the contents of local variables. For a local non-boolean variable, Δ records the heap address ι of an object on the heap or the special value `null`. For local variables of the primitive type `bool`, Δ directly stores the variable's value, either `true` or `false`.

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

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

Note that since our simple syntax does not allow inner classes or nested methods and functions, the values of local variables and parameters can always be found in the top frame of the stack. Thus, variable reads and writes for Δ are defined as $\Delta(x) = \delta(x)$ and $\Delta[x \mapsto v] = \Delta_1 \bullet \delta[x \mapsto v]$ where $\Delta = \Delta_1 \bullet \delta$.

The heap Ψ Objects are stored on the heap Ψ, a partial function mapping addresses ι to objects o. Our semantics does 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, Ψ records its type τ and a dictionary mapping each field name f to its current value v. It also stores metadata about each object's contract parameters, recording the actual contract parameter instantiations that were provided 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 a triple consisting of 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, $\Psi(\iota) = \langle\tau, \{f \mapsto v\}, \{cp \mapsto \langle\iota', e^r, e^w\rangle\}\rangle$.

Contract closures 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 ϕ 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.

[6]We often use ':' instead of '$\mapsto$' for types, for example $\{x : \tau\}$.

Given a syntactic contract ϕ ($\langle e^r, e^w\rangle$ or $<cp>$), an object with address ι and its contract parameter dictionary ρ[7], the *contract closure* ψ is given by $\mathit{closure}(\phi, \iota, \rho)$ defined by:

$$\begin{aligned}\mathit{closure}(\langle e^r, e^w\rangle, \iota, \rho) &= \langle \iota, e^r, e^w\rangle \\ \mathit{closure}(<cp>, \iota, \rho) &= \rho(cp)\end{aligned}$$

The contract store Λ While the program is executing, we need to track which aliasing contracts apply to which objects. To this end, we define contract store Λ, a partial function which, for an object with heap address ι, records the set S of aliasing contracts which currently apply to the object. S is a set of triples $\langle \iota', e_r, e_w\rangle$ containing the address ι' of contract's declaring object and the read and write contract expressions e_r and e_w.

Parameters and local variables exist only while the method which defines them executes; their contracts must therefore 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: $\Lambda = \lambda_0 \bullet \ldots \bullet \lambda_n$.

Frame λ_0 holds the contracts associated with fields; these are not bound to a particular method execution but persist throughout program execution. When a method is called, we add a new stack frame λ_n which holds the contracts declared for the method's parameters and local variables. This frame is removed when the method returns, automatically removing the contracts of the method's parameters and local variables. This stack approach also ensures that contracts for the same local variable in different stack frames are recorded separately.

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, $\Lambda(\iota) = \lambda_0(\iota) \cup \cdots \cup \lambda_n(\iota)$ where $\Lambda = \lambda_0 \bullet \ldots \bullet \lambda_n$.

Whenever a field is assigned a new value (that is, pointed to a new object), its contract needs to be removed from the object to which it previously pointed and added to the object referenced after the assignment; to this end, we update the information in frame λ_0. Assignments of local variables and method parameters require updating the information in the top stack frame λ_n.

At first sight, Δ and Λ appear very similar. However, the semantics of lookup differ significantly: a lookup in Δ requires a lookup in the top frame of the stack only; 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.

Other operators To define the semantics of the `instanceof` operator, we require a boolean operator which determines if a type σ is the subtype of another type τ. We define the compile-time operator $<:$ to return `true` if a subtype relationship exists between σ and τ (as determined by the program's source code) and `false` otherwise. Note that the subtype relationship is both reflexive and transitive: type τ is a subtype of itself. If type τ_1 is a subtype of τ_2 and τ_2 is a subtype of τ_3 then τ_1 is also a subtype of τ_3.

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.

[7]This can be looked up on the heap as $\Psi(\iota)\downarrow_3$.

In our semantics, we denote $G^*(\iota, g, \Psi)$ the g-reachable set of the object at address ι in heap Ψ. An encapsulation group contains paths which at run time evaluate to objects reached by following these paths in Ψ starting from the object at address ι. If other encapsulation groups appear within g then these are recursively followed, allowing specification of any regular set of paths.

$$\frac{}{\begin{array}{c}\iota' \texttt{ canread } \iota'', \Delta, \Psi, \Lambda \rightsquigarrow \mathit{eval}(\iota_1, \iota', \iota'', e_1^r) \;\&\&\; \ldots \\ \&\&\; \mathit{eval}(\iota_n, \iota', \iota'', e_n^r), \Delta, \Psi, \Lambda\end{array}} \qquad \text{(canread)}$$

$$\text{where } \{\langle \iota_1, e_1^r, e_1^w \rangle, \ldots, \langle \iota_n, e_n^r, e_n^w \rangle\} = \Lambda(\iota'')$$

$$\frac{}{\begin{array}{c}\iota' \texttt{ canwrite } \iota'', \Delta, \Psi, \Lambda \rightsquigarrow \mathit{eval}(\iota_1, \iota', \iota'', e_1^w) \;\&\&\; \ldots \\ \&\&\; \mathit{eval}(\iota_n, \iota', \iota'', e_n^w), \Delta, \Psi, \Lambda\end{array}} \qquad \text{(canwrite)}$$

$$\text{where } \{\langle \iota_1, e_1^r, e_1^w, \ldots, \langle \iota_n, e_n^r, e_n^w \rangle\} = \Lambda(\iota'')$$

$$\frac{e, \Delta \bullet \delta, \Psi, \Lambda \rightsquigarrow e', \Delta \bullet \delta, \Psi, \Lambda}{\mathit{eval}(\iota, \iota', \iota'', e), \Delta, \Psi, \Lambda \rightsquigarrow \mathit{eval}(\iota, \iota', \iota'', e'), \Delta, \Psi, \Lambda} \qquad \text{(eval-contract)}$$

$$\text{where } \delta = \{\texttt{this} \mapsto \iota, \texttt{accessor} \mapsto \iota', \texttt{accessed} \mapsto \iota''\}$$

$$\frac{}{\mathit{eval}(\iota, \iota', \iota'', b), \Delta, \Psi, \Lambda \rightsquigarrow b, \Delta, \Psi, \Lambda} \qquad \text{(eval-done)}$$

$$\frac{}{\iota.f, \Delta, \Psi, \Lambda \rightsquigarrow \mathit{assert}(\texttt{this canread } \iota); \iota \odot f, \Delta, \Psi, \Lambda} \qquad \text{(field-get)}$$

$$\frac{}{\iota \odot f, \Delta, \Psi, \Lambda \rightsquigarrow v, \Delta, \Psi, \Lambda} \qquad \text{(field-get-}\odot\text{)}$$

$$\text{where } v = \Psi(\iota) \downarrow_2 (f)$$

$$\frac{}{\iota.f = v, \Delta, \Psi, \Lambda \rightsquigarrow \mathit{assert}(\texttt{this canwrite } \iota); \iota \odot f = v, \Delta, \Psi, \Lambda} \qquad \text{(field-put)}$$

$$\frac{}{\iota \odot f = b, \Delta, \Psi, \Lambda \rightsquigarrow b, \Delta, \Psi', \Lambda} \qquad \text{(field-put-}\odot\text{-bool)}$$

$$\text{where } \Psi' = \Psi[\iota \mapsto \langle \Psi(\iota) \downarrow_1, \Psi(\iota) \downarrow_2 [f \mapsto b], \Psi(\iota) \downarrow_3 \rangle]$$

Figure 3.2: Operational semantics for aliasing contracts

3.2.2.2 Contract evaluation

Contracts are declared on variables (fields, parameters and local variables) and at run time apply to the objects to which the variables point. Whenever on object is accessed, 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 object reads and writes:

- Field reads in rule (*field-get*) and calls to pure methods in rule (*method-call-pure*) constitute object reads.

$$\frac{}{\iota \odot f = a, \Delta, \Psi, \lambda_0 \bullet \Lambda \rightsquigarrow a, \Delta, \Psi', \lambda_0' \bullet \Lambda} \quad \text{(field-put-}\odot\text{)}$$

$$\begin{gathered}\text{where } \Psi' = \Psi[\iota \mapsto \langle \Psi(\iota)\downarrow_1, \Psi(\iota)\downarrow_2 [f \mapsto a], \Psi(\iota)\downarrow_3 \rangle] \\ \text{and } a' = \Psi(\iota)\downarrow_2 (f) \\ \text{and } \tau = \Psi(\iota)\downarrow_1 \\ \text{and } \psi = \mathit{closure}(\mathcal{C}^\tau(f), \iota, \Psi(\iota)\downarrow_3) \\ \text{and } \lambda_0' = \lambda_0[a' \mapsto \lambda_0(a') - \{\psi\}] \text{ (if } a' \neq \texttt{null}) \\ [a \mapsto \lambda_0(a) \cup \{\psi\}] \text{ (if } a \neq \texttt{null})\end{gathered}$$

$$\frac{}{x, \Delta, \Psi, \Lambda \rightsquigarrow \Delta(x), \Delta, \Psi, \Lambda} \quad \text{(local-get)}$$

$$\frac{}{x = b, \Delta, \Psi, \Lambda \rightsquigarrow b, \Delta[x \mapsto b], \Psi, \Lambda} \quad \text{(local-put-bool)}$$

$$\frac{}{x = a, \Delta, \Psi, \Lambda \bullet \lambda_n \rightsquigarrow a, \Delta[x \mapsto a], \Psi, \Lambda \bullet \lambda_n'} \quad \text{(local-put)}$$

$$\begin{gathered}\text{where } a' = \Delta(x) \\ \text{and } \iota = \Delta(\texttt{this}) \\ \text{and } \tau = \Psi(\iota)\downarrow_1 \\ \text{and } \psi = \mathit{closure}(\mathcal{C}^\tau(x), \iota, \Psi(\iota)\downarrow_3) \\ \text{and } \lambda_n' = \lambda_n[a' \mapsto \lambda_n(a') - \{\psi\}] \text{ (if } a' \neq \texttt{null}) \\ [a \mapsto \lambda_n(a) \cup \{\psi\}] \text{ (if } a \neq \texttt{null})\end{gathered}$$

$$\frac{}{\begin{gathered}\iota.m(v_1, \ldots, v_n), \Delta, \Psi, \Lambda \rightsquigarrow \mathit{assert}(\texttt{this canread } \iota); \\ \iota \odot m(v_1, \ldots, v_n), \Delta, \Psi, \Lambda\end{gathered}} \quad \text{(method-call-pure)}$$

$$\begin{gathered}\text{where } \tau = \Psi(\iota)\downarrow_1 \\ \text{and } \mathcal{M}^\tau(m) = (\_, \_, \_, \texttt{true})\end{gathered}$$

$$\frac{}{\begin{gathered}\iota.m(v_1, \ldots, v_n), \Delta, \Psi, \Lambda \rightsquigarrow \mathit{assert}(\texttt{this canread } \iota \ \&\& \\ \texttt{this canwrite } \iota);\ \iota \odot m(v_1, \ldots, v_n), \Delta, \Psi, \Lambda\end{gathered}} \quad \text{(method-call-impure)}$$

$$\begin{gathered}\text{where } \tau = \Psi(\iota)\downarrow_1 \\ \text{and } \mathcal{M}^\tau(m) = (\_, \_, \_, \texttt{false})\end{gathered}$$

Figure 3.3: Operational semantics for aliasing contracts

$$\frac{}{\iota \odot m(v_1, \ldots, v_n), \Delta, \Psi, \Lambda \rightsquigarrow E, \Delta \bullet \delta, \Psi, \Lambda \bullet \lambda} \quad \text{(method-call-}\odot\text{)}$$

$$\begin{gathered}
\text{where } \tau = \Psi(\iota) \downarrow_1 \\
\text{and } \mathcal{M}^\tau(m) = (\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}, E, \{y_1 : \tau_1, \ldots, y_m : \tau_m\}, \pi) \\
\text{and } \delta = \{\texttt{this} \mapsto \iota, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \\
y_1 \mapsto \mathit{init}(\tau_1), \ldots, y_m \mapsto \mathit{init}(\tau_m)\} \\
\text{and } \psi_i = \mathit{closure}(\mathcal{C}^\tau(x_i), \iota, \Psi(\iota) \downarrow_3) \ \ (1 \le i \le n) \\
\text{and } \lambda = \{v_i \mapsto \psi_i \mid 1 \le i \le n \wedge v_i \notin \{\texttt{null}, \texttt{true}, \texttt{false}\}\}
\end{gathered}$$

$$\frac{}{\texttt{new}\ C\langle(\phi_1), \ldots, (\phi_m)\rangle(v_1, \ldots, v_n), \Delta, \Psi, \Lambda \rightsquigarrow E; \iota, \Delta \bullet \delta, \Psi', \Lambda \bullet \lambda} \quad \text{(new)}$$

$$\begin{gathered}
\text{where } \iota \text{ is new in } \Psi \\
\text{and } \iota' = \Delta(\texttt{this}) \\
\text{and } \mathcal{F}^C = \{f_1 : \tau_1, \ldots, f_k : \tau_k\} \\
\text{and } \mathcal{CP}^C = \{\mathit{cp}_1, \ldots, \mathit{cp}_m\} \\
\text{and } \psi_i = \mathit{closure}(\phi_i, \iota', \Psi(\iota') \downarrow_3) \ \ (1 \le i \le m) \\
\text{and } \Psi' = \Psi[\iota \mapsto \langle C, \{f_1 \mapsto \mathit{init}(\tau_1), \ldots, f_k \mapsto \mathit{init}(\tau_k)\}, \\
\{\mathit{cp}_1 \mapsto \psi_1, \ldots, \mathit{cp}_m \mapsto \psi_m\}\rangle] \\
\text{and } \mathcal{M}^C(C) = (\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}, E, \{y_1 : \tau'_1, \ldots, y_l : \tau'_l\}, \_) \\
\text{and } \delta = \{\texttt{this} \mapsto \iota, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \\
y_1 \mapsto \mathit{init}(\tau'_1), \ldots, y_l \mapsto \mathit{init}(\tau'_l)\} \\
\text{and } \psi'_j = \mathit{closure}(\mathcal{C}^C(x_j), \iota, \Psi(\iota) \downarrow_3) \ \ (1 \le j \le n) \\
\text{and } \lambda = \{v_j \mapsto \psi'_j \mid 1 \le j \le n \wedge v_j \notin \{\texttt{null}, \texttt{true}, \texttt{false}\}\}
\end{gathered}$$

$$\frac{}{\iota\ \texttt{in}\ \iota'.g, \Delta, \Psi, \Lambda \rightsquigarrow b, \Delta, \Psi, \Lambda} \quad \text{(in)}$$

$$\text{where } b = \begin{cases} \texttt{true} & \iota \in G^*(\iota', g, \Psi) \\ \texttt{false} & \iota \notin G^*(\iota', g, \Psi) \end{cases}$$

$$\frac{}{\texttt{if true then}\ E_1\ \texttt{else}\ E_2, \Delta, \Psi, \Lambda \rightsquigarrow E_1, \Delta, \Psi, \Lambda} \quad \text{(conditional-true)}$$

$$\frac{}{\texttt{if false then}\ E_1\ \texttt{else}\ E_2, \Delta, \Psi, \Lambda \rightsquigarrow E_2, \Delta, \Psi, \Lambda} \quad \text{(conditional-false)}$$

$$\frac{}{v; E, \Delta, \Psi, \Lambda \rightsquigarrow E, \Delta, \Psi, \Lambda} \quad \text{(sequence)}$$

Figure 3.4: Operational semantics for aliasing contracts

$$\frac{}{!b, \Delta, \Psi, \Lambda \rightsquigarrow b', \Delta, \Psi, \Lambda} \quad \text{(not)}$$

$$\text{where } b' = \begin{cases} \texttt{true} & b = \texttt{false} \\ \texttt{false} & b = \texttt{true} \end{cases}$$

$$\frac{}{v_1 == v_2, \Delta, \Psi, \Lambda \rightsquigarrow b, \Delta, \Psi, \Lambda} \quad \text{(equals)}$$

$$\text{where } b = \begin{cases} \texttt{true} & v_1 = v_2 \\ \texttt{false} & v_1 \neq v_2 \end{cases}$$

$$\frac{}{\iota \ \texttt{instanceof} \ \tau, \Delta, \Psi, \Lambda \rightsquigarrow b, \Delta, \Psi, \Lambda} \quad \text{(instance-of)}$$

$$\text{where } \sigma = \Psi(\iota) \downarrow_1$$
$$\text{and } b = \begin{cases} \texttt{true} & \sigma <: \tau \\ \texttt{false} & \neg\sigma <: \tau \end{cases}$$

$$\frac{}{\texttt{false} \ \&\& \ E, \Delta, \Psi, \Lambda \rightsquigarrow \texttt{false}, \Delta, \Psi, \Lambda} \quad \text{(and-false)}$$

$$\frac{}{\texttt{true} \ \&\& \ E, \Delta, \Psi, \Lambda \rightsquigarrow E, \Delta, \Psi, \Lambda} \quad \text{(and-true)}$$

$$\frac{}{\texttt{true} \ || \ E, \Delta, \Psi, \Lambda \rightsquigarrow \texttt{true}, \Delta, \Psi, \Lambda} \quad \text{(or-true)}$$

$$\frac{}{\texttt{false} \ || \ E, \Delta, \Psi, \Lambda \rightsquigarrow E, \Delta, \Psi, \Lambda} \quad \text{(or-false)}$$

$$\frac{E, \Delta, \Psi, \Lambda \rightsquigarrow E', \Delta', \Psi', \Lambda'}{C[E], \Delta, \Psi, \Lambda \rightsquigarrow C[E'], \Delta', \Psi', \Lambda'} \quad \text{(context-e)}$$

Figure 3.5: Operational semantics for aliasing contracts

- Field writes in rule (*field-put*) are object writes.
- Calls to impure methods in rule (*method-call-impure*) constitute both an object read *and* an object write.
- The operators `canread` in rule (*canread*) and `canwrite` in rule (*canwrite*) evaluate an object's read and write contracts respectively.

Reads and writes of local variables, including the special variables `this`, `accessor` and `accessed`, do not require contract evaluation, as explained previously and shown in rule (*local-get*). They represent reads and writes to the unaliased *stack* only; the heap is not accessed and thus no aliasing contracts need to be evaluated. As already explained above, constructor calls also do not require contract evaluation; they directly read or write only to a heap region which does not yet represent an object (and therefore has no associated contracts).

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

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

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

as appropriate – see rules (*field-get*), (*field-put*), (*method-call-pure*) and (*method-call-impure*). Here '$\odot$' represents a standard object access *without* contract evaluation.

The `canread` and `canwrite` operators, described in 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 is, the read or write contract expressions) that currently apply to the object at address ι'' (the `accessed` object), $\iota_1 \ \ldots \ \iota_n$ are the declaring objects of contracts $e_1 \ \ldots \ e_n$ and ι' is the address of the object performing the access ("`accessor`"). The `&&` operator is lazy, as shown in rule (*and-false*). This means that if any of the *eval* terms reduces to `false`, the entire clause also immediately reduces to `false`.

Each contract associated with an object is evaluated separately using the *eval* construct (see rules (*eval-contract*) and (*eval-done*)), which is given (by `canread`/`canwrite` in rules (*canread*) and (*canwrite*)) the address ι of the contract's declaring object (bound in a new stack frame δ as "`this`"), the address ι' of the object whose method is making the access (bound as "`accessor`") and the address ι'' of the accessed object (bound as "`accessed`").

Execution proceeds as shown in rule (*eval-contract*) until a boolean value is produced in rule (*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 reading or writing an object. Any sub-expressions are reduced before contracts are evaluated. For example, for a field write of the form $E_1.f = E_2$, E_1 and E_2 are reduced *before* the contract evaluation required for the field write is performed. This means that any side-effects in the expressions E_1 and E_2 can potentially influence the outcome of the contract evaluation.

We note here that if there are multiple contracts to evaluate, evaluation order is irrelevant. Expressions in contracts may not have side-effects; contracts cannot declare new variables or contracts, or change existing variables or contracts. This means that Δ, Ψ, and Λ must be the same before and after each contract evaluation and thus one contract evaluation cannot affect the result of other contract evaluations.[8]

3.2.2.3 Contract transfer

Whenever a variable assignment occurs, the contracts in contract store Λ must be updated: the contract 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 points after the assignment.

Such a contract transfer occurs for field writes and local variable writes, as shown in rules (*field-put-*$\odot$) and (*local-put*). Note that (*field-put-*$\odot$) modifies the contracts in λ_0, while (*local-put*) updates the contracts in λ_n.

When a parameterised contract needs to be stored in the contract store Λ, we look up the actual contract associated with the contract parameter in the metadata of the contract's declaring object on the heap. This gives us the actual contract expressions associated with the contract parameter, as well as the appropriate declaring object. We store this information in Λ, enabling parameterised contracts to be evaluated like any other contract. Contract evaluation thus needs to have no knowledge of parameterised contracts (and indeed the contract evaluation rules in our operational semantics do not).

Booleans are primitive types which are not subject to aliasing. Consequently, writes of boolean fields and local variables do not require updates to the contract store, as shown in rules (*field-put-*$\odot$*-bool*) and (*local-put-bool*).

3.2.2.4 Method and constructor calls

The semantics of method and constructor calls are quite similar to each other, as shown in rules (*method-call-*$\odot$) and (*new*).

Rule (*new*) modifies the heap to create a new object, recording the object's type C, its fields and their (default) values, and the contract parameters cp_i with their associated contract closures ψ_i. The syntactic contracts ϕ_i passed to the constructor as instantiations for the object's contract parameters may either be full syntactic contracts with two contract expressions or may themselves be parameterised contracts. The function *closure* converts the syntactic contracts ϕ_i to contract closures ψ_i.

To perform the actual method or constructor call (*method-call-*$\odot$) and (*new*) 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 newly created object. The values of the method's or constructor's parameters $x_1, \ldots, x_n$ are initialised to the values $v_1, \ldots, v_n$ (the method or constructor call arguments). The local variables $y_1, \ldots, y_m$ are initialised to their default values $w_1, \ldots, w_m$.

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

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` (or `false`) and thus do not yet point to objects, their contracts do not need to be added to the contract store. Parameters, on the other hand, may already point to heap objects; thus the contracts of non-boolean method parameters must be added to the contract store frame λ (provided the method argument corresponding to a parameter is not `null`).

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 E and then returns the address ι of the newly created object; this enables chaining, for example `new Object().method()`.

3.2.2.5 The `in` operator

The evaluation of `in` relies on the computation of the set of objects reachable from an encapsulation group g. For E `in` $E'.g$, we evaluate E' to give object address ι and then follow all paths in g starting from ι in heap Ψ. This is shown in rule (*in*), which uses the G^* operator to compute the set of g-reachable objects.

Calculating the reachable set of an encapsulation group is potentially expensive. In the worst case it could 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 cyclic (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 that have to record which encapsulation groups have already been evaluated; the formalisation given here keeps our reduction rules simple and concise.

3.2.2.6 Conditionals, sequences and boolean operators

Finally, we define some standard reduction rules for conditionals, sequences and boolean operators. The conditional rules (*conditional-true*) and (*conditional-false*) express that a `true` condition in an `if`-clause leads to the evaluation of the `then`-clause, while a `false` condition triggers the evaluation of the `else`-clause.

The (*sequence*) rule fixes the evaluation order of a sequence of expressions: the second expression can be evaluated only after the first one has been reduced to a value.

Our semantics also include rules for the standard boolean operators `!` in rule (*not*), `==` in rule (*equals*), `&&` in rules (*and-false*) and (*and-true*) and `||` in rules (*or-false*) and (*or-true*). These operators have the expected semantics; `&&` and `||` are both evaluated lazily.

We also define the evaluation of the `instanceof` operator in rule (*instance-of*) using the operator $<:$ defined in Section 3.2.2.1: an expression ι `instanceof` τ evaluates to `true` if type σ of the object at heap address ι is a subtype of τ.

3.2.2.7 Evaluation contexts

There are a number of contexts in which one expression E is reduced to another expression E'. Instead of many similar, separate rules, we define the reduction rule (*context-e*) using evaluation contexts to express this.

Possible evaluation contexts are shown below; we use C to denote the evaluation context. For compactness, we define *bin_op* to range over binary operators $==$, `canread` and `canwrite`.

$$
\begin{aligned}
C = {} & [-] \\
& \mid \texttt{if } C \texttt{ then } E \texttt{ else } E \\
& \mid C; E \\
& \mid C.f \\
& \mid C.f = E \\
& \mid \iota.f = C \\
& \mid x = C \\
& \mid C.m(E, \ldots, E) \\
& \mid \iota.m(C, E, \ldots, E) \\
& \mid \iota.m(v, \ldots, v, C, E, \ldots, E) \\
& \mid \texttt{new } c\langle(\phi), \ldots, (\phi)\rangle(C, E, \ldots, E) \\
& \mid \texttt{new } c\langle(\phi), \ldots, (\phi)\rangle(v, \ldots, v, C, E, \ldots, E) \\
& \mid !C \\
& \mid C \textit{ bin\_op } E \\
& \mid v \textit{ bin\_op } C \\
& \mid C \texttt{ instanceof } \tau \\
& \mid C \texttt{ in } E.g \\
& \mid \iota \texttt{ in } C.g \\
& \mid C \mid\mid E \\
& \mid C \mathbin{\&\&} E
\end{aligned}
$$

3.3 Case study

We now present a case study to demonstrate the flexibility of aliasing contracts, showing how they can be used to enforce a variety of aliasing conditions. We discuss how aliasing contracts can support encapsulation in the iterator design pattern [37]. We chose this case study because existing static alias protection schemes struggle to implement iterators; as a result, iterators are frequently discussed in literature about alias protection (and used to demonstrate the failings of existing work), including in Clarke et al.'s work on ownership types [27] and Dietl et al.'s work on universe types [32]. Appendix A contains three additional case studies: the binary tree data structure and the observer and memento design patterns.

Iterator is a behavioural design pattern proposed by Gamma et al. [37]. It allows traversal of a collection's elements in sequential order, without exposing the underlying representation of the collection. Thus, the client does not need to know if it is dealing with, for example, an array list, linked list or hash set.

An iterator can be implemented in a number of different ways. Here, we consider what Noble calls a *structure sharing iterator* [73]. This iterator implementation is similar to that used by Java's collections library. A UML class diagram of the structure sharing iterator pattern is shown in Figure 3.6; it deviates slightly from the diagram given by Gamma et al. [37] who present an *external iterator* design. We also use a slightly different terminology from Gamma et al.'s version, referring to the `Collection` and `ConcreteCollection`, instead of `Aggregate` and `ConcreteAggregate`.

The important details of the iterator pattern are as follows:

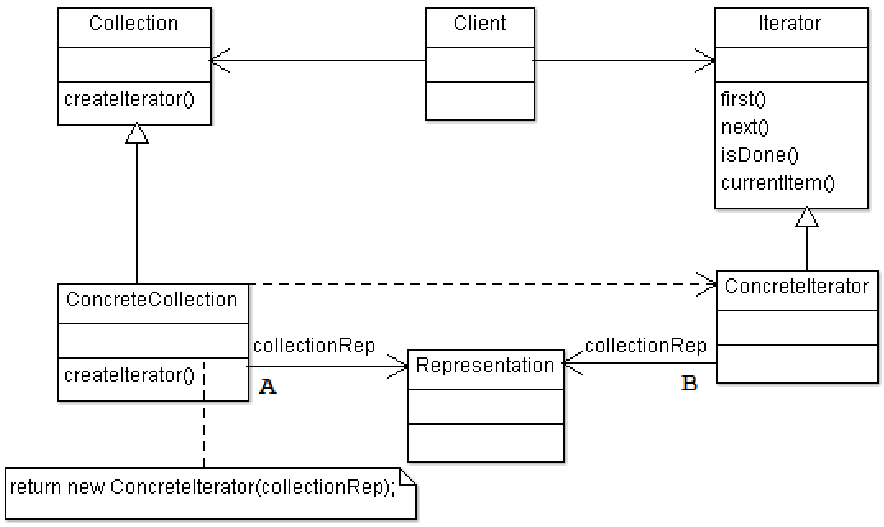


Figure 3.6: UML class diagram of the structure sharing iterator pattern

- The `createIterator` method in `ConcreteCollection` can be called by clients to obtain a reference to an iterator for the collection.
- `ConcreteIterator` contains the functionality for traversing the collection.
- `Representation` is a class representing the data of the collection. This may take a number of different forms, for example an array for array lists or a chain of nodes for a linked list. In our example, we represent all of these possibilities as a single class. We are not interested in the implementation details of the collection's representation here, only in how the representation is shared between the collection and the iterator.

Iterator is commonly discussed in the literature about alias protection because it requires the collection representation to be shared by the collection and one or more iterators. For this reason, single-ownership alias protection schemes such as Clarke-style ownership types struggle to implement structure sharing iterators [73] (although they may be able to implement other versions of the iterator pattern).

Using aliasing contracts, we can achieve sharing of the collection representation between the collection and the iterators. There are several different ways to achieve this. Depending on the exact usage context, different options may be preferable.

All of the options we present below use the rw-contract "`true`" for reference `B` in Figure 3.6. The reason for this is that the collection itself needs to protect its representation; iterators come and go and therefore cannot provide permanent alias protection. This places all of the protection burden on the collection itself.

There are a number of different ways that we can annotate reference `A` in Figure 3.6:

- The rw-contract

```
{accessor == accessed || accessor == this ||
    accessor instanceof ConcreteIterator}
```

allows the collection representation to be read and written by the representation itself (“`accessed`”), the collection (“`this`”) and any objects of type `ConcreteIterator`.

One problem with this approach is that it allows *all* iterators to access a collection’s representation. Thus, iterators could access the representation of all collection instances, not just of the one they are traversing.

- The rw-contract

  ```
  {accessor == accessed || accessor == this ||
      (accessor instanceof ConcreteIterator &&
      ((ConcreteIterator) accessor).getCollection() == this)}
  ```

  addresses the problem with the previous version by checking that iterators which access the collection are indeed associated with that particular collection instance. While slightly safer than the previous version, this contract requires the iterator to implement an additional method, `getCollection`, which returns the collection instance being traversed.

- The rw-contract

  ```
  {accessor == accessed || accessor == this ||
      accessor == currentIterator}
  ```

  works well when only one iterator should traverse the collection at a time, for example to avoid concurrency issues. In this case, the collection can record the `currentIterator` and give it access rights to its representation. If a second iterator attempts to traverse the collection at the same time, this will cause a contract violation.

- The rw-contract

  ```
  {accessor == accessed || accessor == this ||
      iterators.contains(accessor)}
  ```

  modifies the previous one to allow multiple iterators to traverse the collection at the same time. One issue with this version is that iterators which are no longer needed must be explicitly removed and destroyed[9]. On the other hand, this approach can implement quite subtle constraints; for example, we could easily restrict the number of concurrent iterators to two or three by limiting the size of the `iterators` collection.

- The contract

[9]This could, for example, be implemented in Java by storing iterators in a `WeakHashMap`, thus removing the need to explicitly de-register iterators. When an iterator is no longer used by the rest of the system, it can be finalised and automatically removed from the `WeakHashMap`. Storing iterators in a standard data structure would prevent them from being finalised in this way. However, with this implementation there is no way of knowing exactly when iterators will disappear from the `WeakHashMap`, as this depends on when (and if) they are finalised by the GC.

```
{accessor == accessed || accessor == this ||
    iterators.contains(accessor),
accessor == this || accessor == accessed ||
    accessor == writeIterator}
```

combines the previous two approaches. It allows many iterators to traverse the collection at the same time, while writes are limited to a single iterator (“`writeIterator`”).

Standard OO design advice says that behaviour should be placed with the data on which it acts. This is, for example, expressed by Riel’s heuristic “Keep related data and behaviour in one place” [85]. Separating behaviour from the relevant data causes coupling between the class containing the data and the class containing the behaviour.

Arguably, iterator does exactly this, separating the traversal behaviour from the collection and its items. This separation of data and behaviour requires the data to be shared by more than one object at run time, causing problems for single-ownership systems. The visitor and strategy patterns [37] include a similar separation; we can implement alias protection with aliasing contracts for these patterns, much in the same way as described for iterator here.

3.4 Summary

In this chapter, we introduced a dynamic alias protection scheme called aliasing contracts. We developed a formal operational semantics for aliasing contracts and demonstrated how they can be applied in practice to support encapsulation in the iterator design pattern.

Aliasing contracts include several powerful features: encapsulation groups which can be used to specify deep and transitive contracts and contract parameters which can be used to achieve contract polymorphism. The case study we presented here (and the additional case studies in Appendix A) show how flexible and expressive aliasing contracts are. They can be used to express a wide range of aliasing conditions, including some rather complex ones.

In Chapter 4, we continue with a detailed comparison between aliasing contracts and existing alias protection schemes.

CHAPTER 4

Comparison of aliasing contracts with existing alias protection schemes

In Chapter 3, we introduced aliasing contracts, a dynamic alias protection scheme. In this chapter, we present a detailed comparison of aliasing contracts with existing work. To facilitate this comparison, we consider the same categories we used to classify existing alias protection schemes in Chapter 2.

Aliasing contracts are highly flexible and expressive, as demonstrated with the case studies in Chapter 3 and Appendix A. Their high level of expressiveness enables them to model many alias protection policies used in existing work, including full encapsulation, module encapsulation and the three variants of owners-as-dominators and owners-as-modifiers (strict, peer and transitive). We thus argue that aliasing contracts subsume many of the existing models and can be used as a unifying approach to alias protection. We further demonstrate this in Section 4.11 by using aliasing contracts to compare two seemingly different aliasing protection systems: full encapsulation and Clarke-style ownership types.

4.1 References and object accesses

In Section 2.1.2.1, we noted that many existing alias protection schemes limit which references can exist (by restricting which variables can be assigned to each other), while other systems allow any references to occur but restrict how the references can be used; we say that some systems restrict *references*, while others restrict *object accesses* via these references. Despite the obvious differences between the two approaches, we argued in Section 2.1.2.1 that both provide the same encapsulation guarantees.

Aliasing contracts restrict *object accesses*. This increases their flexibility compared to systems which restrict references. Whether an object access is allowed (that is, a particular reference is usable) is determined by the program's aliasing structure at the time the object access is made; an unusable reference may become usable as the aliasing structure changes at run time and vice versa. This is not possible in systems which restrict references, as the introduction of the illegal (and perhaps later legal) reference would have been prohibited by the system. In such systems, either a reference is legal or it is illegal but this does not change as the program executes.

In the example below, the object referenced by variable `o` becomes inaccessible at program point `B`, when it is assigned to `o2` (that is, reference `o` becomes unusable). Any accesses to it now produce a contract violation of the rw-contract "`false`" of variable `o2`. Once that contract is removed at program point `D`, the object in `o` becomes accessible again (that is, reference `o` becomes usable). Thus, the object accesses at program points `A` and `E` succeed, but the access at program point `C` fails:

```
Object o {true};
Object o2 {false};
o.m();                              //A
o2 = o;                             //B
o.m();                              //C
o2 = new Object();                  //D
o.m();                              //E
```

This example also shows that it is possible for two variables with conflicting contracts to point to the same object without causing an error; a contract violation is reported only if the object is then accessed. This behaviour has an analogy in typing: some type systems may allow the definition of a valid type which turns out not to have any members; problems occur only when we try to create members of the type, not in the type definition itself.

4.2 Static and dynamic checking

In Section 2.1.2.2, we discussed the relative advantages and disadvantages of static and dynamic alias checking. We also noted that systems which restrict object accesses rather than references cannot usually be checked statically.

Aliasing contracts limit object accesses and thus require dynamic checking. This is due to the highly dynamic nature of aliasing contracts and their dependence on the aliasing structure of the program at the time an object access occurs at run time.

4.3 Temporal and spatial aliasing

Section 2.1.2.3 introduced a distinction between temporal and spatial aliasing. Like most other alias protection schemes, aliasing contracts are concerned with spatial aliasing only; that is, which spatial aliases to an object exist at the time of object access. Aliasing contracts can be used to enforce some temporal aliasing restrictions (as we discuss in Section 4.10.3 below, where we show how to model linearity with aliasing contracts), but they provide no *explicit* support for restricting temporal aliasing.

4.4 Static and dynamic aliasing

Section 2.1.2.4 introduced the difference between static and dynamic aliasing: static aliasing describes aliasing of objects through fields (that is, more permanent, *static* variables), while dynamic aliasing represents aliasing through (less permanent) local variables and method parameters. Some alias protection schemes (such as Hogg's islands [48] and Almeida's balloons [4]) loosen restrictions on dynamic aliasing to gain flexibility.

Aliasing contracts do not distinguish between static and dynamic aliases; the contracts of all references, whether fields, local variables or method parameters, are treated in the same way. However, contract suspension can be used to temporarily loosen restrictions on an object, for example for the duration of a method call. This allows us to model dynamic aliasing, as we explain in more detail in Section 4.6.

A distinction between static and dynamic aliasing could additionally be achieved by a simple change to the operational semantics of aliasing contracts presented in Chapter 3. Instead of checking contracts for *all* field reads, field writes and method calls as is done in the current semantics, we could perform such checks only when the read, write or method call is made through a field (that is, a static alias).

We currently have no plans to implement the modification described above. Allowing dynamic aliases across encapsulation boundaries would significantly weaken the encapsulation guarantees provided by our system. The aim of aliasing contracts is to allow objects to state assumptions about when the objects they reference may be read and written. Allowing some reads and writes without contract checks would defeat this purpose; programmers could no longer rely on the assumptions expressed in contracts, leading to potentially unexpected reads and writes.

4.5 Sharing

Many alias protection schemes include the concept of *shared* objects which can be accessed from anywhere in the system, as described in Section 2.1.2.5.

Aliasing contracts can model shared objects using the rw-contract "`true`". This contract allows reads and writes to the object from anywhere in the system, as it always evaluates to `true`.

There is, however, a subtle difference in semantics between objects annotated with the rw-contract "`true`" and shared objects in other systems. In other aliasing protection schemes, shared objects always remain shared. Objects stored in a variable of one protection level cannot usually be assigned to a variable of a different protection level. An object in a shared variable can never be assigned to a variable of a higher protection level and thus always remains shared.

Aliasing contracts, on the other hand, do not restrict assignments in this way and therefore allows objects to move between different encapsulation levels. If a shared object is moved to a higher protection level, it ceases to be shared. The validity of an object access depends on the conjunction of the aliasing contracts which currently apply to the accessed object. Even if one reference has the rw-contract "`true`", this does not necessarily mean that the object to which the variable points can be accessed freely from anywhere in the system, as the following example shows:

```
Object o {true};
Object o2 {accessor == accessed || accessor == this};
o = o2;                          //A
o2 = new Object();               //B
```

At the start, variable `o` points to a shared object. As soon as the more restrictive rw-contract "`accessor == accessed || accessor == this`" is added to the previously shared object at program point A of the example, the object in o can no longer be accessed freely from anywhere in the system; the more restrictive contract essentially overrides the weaker rw-contract "true". The reason for this lies in the conjunction of contracts: conjoining the contract "true" with another contract simply reduces to the evaluation result of the other contract. The object in o becomes shared again at program point B above when the more restrictive contract is removed.

This behaviour is similar to that of Boyland et al.'s capabilities [21]: various references to a shared object may exist, none of them asserting exclusive access rights. If another reference then asserts exclusive access rights, the base rights of the other references may be stripped away: the object is then no longer shared.

Static analysis can be used to detect cases where a shared object is assigned to a reference with a more restrictive contract. The static analysis we present in Chapter 6 reports such situations, alerting the programmer that an object which was intended to be shared may be blocked by another contract.

4.6 Borrowing

Borrowing, as described in Section 2.1.2.6, allows temporary aliasing (and thus accesses) to an otherwise encapsulated object, for example for the duration of a method call. A borrowed reference can exist for a limited time only and may not be stored permanently (for example in a field).

Aliasing contracts can model borrowing through contract suspension (which is desugared as explained in Section 3.1.3). The contract protecting an object can thus be removed temporarily, for example for the duration of a method call as shown in the example below. This allows other parts of the system to access the usually protected object for a limited time:

```
Object o {accessor == accessed || accessor == this};
suspend(this.o) {                   //A
  foo.bar(this.o);                  //B
}                                   //C
```

The object referenced by variable o is usually protected from accesses except those coming from itself ("accessed") and from the object which declares variable o ("this"). However, after the contract of variable o (which previously protected the object) is suspended at program point A, it becomes accessible to other parts of the system (for example in method foo.bar to which it is passed at program point B). The contract of variable o is reinstated at program point C at the end of the suspend-block and the object once again becomes encapsulated.

Suspending a contract achieves the same semantics as borrowing: the object becomes accessible to another part of the system for the duration of the method call, but not beyond that. Even if the callee retains a reference to the borrowed object beyond the method call (for example by storing it in a field) it cannot use that reference later, since the object will be protected by the reinstated contract.

We note that, as explained in Section 3.1.3, a contract can only be suspended by an object which has read and write access rights to the contract's declaring object. This is essential to ensure that contracts cannot be suspended arbitrarily to circumvent encapsulation.

Borrowing (with contract suspension) can be used to simulate dynamic aliasing. For example, suspending the contracts of method arguments for each method call in the system allows object accesses to occur through method parameters.

4.7 Immutability

In Section 2.1.2.7 we noted that immutability can signify three different things in the context of alias protection. In some cases (including C++'s `const`, Scala's `val` and Java's `final` keywords) immutability protects references, so that the value (that is, the address) stored in an immutable reference may not be modified (but the object to which it points may be). Alternatively, immutability may prevent a particular reference being used to write the object it contains (as for read-only references of universe types), although aliasing references may be used to modify the object. Finally, immutability may prohibit modifications to the object through any aliasing references (as for Hogg's `read` annotation and Scala's immutable objects). We argue that only the last of these three semantics is useful in the context of alias protection as it prevents all object writes, regardless of which reference they come through.

Aliasing contracts restrict object accesses, not references, and therefore can model only the last of the three semantics described above. The aliasing contract "`true, false`" ensures that the object to which the variable points cannot be written, as long as the variable points to it. Attempts to write the object will always result in a violation of the variable's write contract, "`false`".

The behaviour of contract "`false`" does not depend on the existence of other aliasing contracts (unlike contract "`true`" discussed in Section 4.5 above). Because contract evaluation is conjunctive, contract evaluation will always fail in the presence of the contract "`false`". This is demonstrated by the example below:

```
Object o {true, false};
Object o2 {true};
o.f = g;                //A
o = o2;                 //B
o.f = g;                //C
```

The object stored in variable `o` remains immutable (protected by the write contract "`false`") throughout the example. The addition of a second, looser contract at program point `B` has no effect on this. Attempts to write a field of `o` at program points `A` and `C` both fail.

4.8 Multiple ownership

Many important programming idioms such as iterators require multiple ownership, as discussed in Section 2.1.1.3. However, multiple ownership is not supported by many existing ownership-based alias protection schemes, as it is complex to check statically.

Aliasing contracts naturally support multiple ownership. For example, the rw-contract "`accessor == foo || accessor == bar`" ensures that the referenced object can be accessed by both `foo` and `bar` (provided there are no other interfering contracts).

Multiple ownership is also supported by encapsulation groups. For example, the rw-contract "`accessor in owners`" specifies that an object can be accessed by any of the objects in the `owners` group. This not only allows any number of owners to share an object, but supports a theoretically unlimited and changing number of actual owners at run time.

The approaches discussed above require an object's owners to be mutually aware of each other. If, for example, one of the owners specifies the rw-contract "`accessor == this`" for the multiply owned object (thinking it is the object's sole owner), the other owners would be blocked from accessing the object. A similar approach is taken by Östlund et al. in their work on ombudsmen-as-dominators [79], where the ombudsmen sharing a multiply owned object must cooperate in a "benevolent" [79] manner; that is, they must know about each other.

For aliasing contracts, it is in fact sufficient for one object to know about all of the owners; this is far more likely to occur in practice than all owners being aware of each other. This object can then specify a contract as above, for example rw-contract "`accessor == foo || accessor == bar`"; we call it the *coordinating object*. The remaining owners can then simply use the rw-contract "`true`", relying on the object to be protected (and correctly shared) by the coordinating object.

In other cases, owners may have some information about other owners, although they may not know their exact identity. For example, one owner may know the type of the other owner; in this case, it could specify the rw-contract "`accessor == this || accessor instanceof Foo`" where `Foo` is the type of the second owner. Although less safe than explicitly specifying all owners, this contract at least does not conflict with the other owner's contract. In the example below, the contracts of variables `o` in `Foo` and `o2` in `Bar` do not conflict; a `Foo` and a `Bar` object can simultaneously access (and own) the same object:

```
class Foo {
  Object o {accessor == this || accessor instanceof Bar};
}
class Bar {
  Object o2 {accessor == this || accessor instanceof Foo};
}
```

In their work on multiple ownership, Cameron et al. argue that "one owner often does not, or cannot, know the other potential owners" [22]. For this reason, they support an owner wildcard, `?`. As an example, consider:

```
Project<this & ?> prj = new Project<this & ?>();
```

This creates a new `Project` object which is owned by `this` and an additional unknown owner. This owner could essentially be any other object in the system. In the most extreme case, the other owner could be the root of the ownership tree, thus making the object in `prj` accessible from anywhere in the system.

We argue that such an approach to multiple ownership is unsafe. In the worst case, additional (unspecified) owners could allow the multiply owned object to be accessed from anywhere in the system. This provides no encapsulation guarantees at all, limiting its usefulness for alias protection.

4.9 Ownership transfer

As discussed in Section 2.1.1.3, many ownership-based alias protection schemes also struggle with ownership transfer. Using aliasing contracts, ownership transfer occurs naturally as the aliasing structure of the program changes at run time.

In the example below, the object stored in variable `o` is first accessible only to itself and the object which declares variable `o` (“`this`”), before becoming shared at program point `B`. This represents a transfer of ownership from `this` to the entire system:

```
Object o {accessor == accessed || accessor == this};
Object o2 {true};
o2 = o;                        //A
o = new Object();              //B
```

The example shows that transfer occurs in two steps. At program point `A`, a second contract is added to the object, in this case the rw-contract “`true`”. At program point `B`, the original contract is removed. This removal can be achieved by pointing `o` to a new object (as in the example) or by nullifying `o`.

4.10 Aliasing policies

We now show how aliasing contracts can be used to model the various aliasing policies described in Section 2.1.1.

4.10.1 Full encapsulation

Full encapsulation, as described in Section 2.1.1.1, is a rather restrictive aliasing policy. Objects are encapsulated inside a single object (which we call the *bridge* below, as in Hogg’s work [48]). Encapsulated objects can freely alias each other; unencapsulated objects can also freely alias each other. However, all aliasing between encapsulated and unencapsulated objects must pass through the bridge object. One major limitation of full encapsulation is that all objects referenced directly or transitively by an encapsulated object must also be encapsulated.

We can model full encapsulation using aliasing contracts with the help of encapsulation groups. In each class, we define an encapsulation group called `transitiveClosure` which will at run time contain all objects that can be directly or transitively reached through a series of field selections, starting from the group’s declaring object. For example, for a class with fields `f1`, ..., `fn`, we define `transitiveClosure` as follows:

```
group transitiveClosure = {f1, f1.transitiveClosure, ..., fn,
    fn.transitiveClosure};
```

Next, we identify the bridge classes in the program. These are classes whose instances act as bridges at run time, encapsulating all directly and indirectly referenced objects. In bridge classes, we give each field the rw-contract “`accessor == this || accessor in transitiveClosure`”. This contract ensures that all directly or transitively contained objects (that is, all objects encapsulated by the bridge) can be accessed only by the bridge (“`this`”) and other encapsulated objects (in group “`transitiveClosure`”).

For fields in all remaining classes, we use the contract “`accessor canread this, accessor canwrite this`”. This contract passes on contract evaluation to its declaring object until either a bridge object is found or if the *root* of the system is reached. The root of the system is the “main” object which is created by the operating system before program execution starts; it resides at the bottom of the execution stack and persists throughout program execution. Importantly, all other objects in the system can be reached through reference paths from the root object. This means that evaluation of the contract "`accessor canread this, accessor canwrite this`" must eventually reach either a bridge object or the system's root object[1]. If a bridge object is reached, the accessed object must be an encapsulated object and contract evaluation (of the rw-contract "`accessor == this || accessor in transitiveClosure`") will succeed only if the access came from the bridge itself or another encapsulated object. If contract evaluation reaches the root of the system, the accessed object was not encapsulated; in this case, contract evaluation always succeeds, as the root object of the system does not have any contracts associated with it (that is, there are no references to it).

This approach is demonstrated in the example below. Class `Bridge` defines a bridge class, while `Foo` and `Bar` are standard non-bridge classes (whose instances may be either encapsulated or unencapsulated):

```
class Bridge {
  Foo o1 {accessor == this || accessor in transitiveClosure};
  group transitiveClosure = {o1, o1.transitiveClosure};
}

class Foo {
  Bar o2 {accessor canread this, accessor canwrite this};
  group transitiveClosure = {o2, o2.transitiveClosure};
}

class Bar {
  ...
}
```

4.10.2 Owners-as-dominators and owners-as-modifiers

Owners-as-dominators and owners-as-modifiers are perhaps the best-known alias protection policies, implemented, for example, by Clarke et al.'s ownership types [27] and Müller et al.'s universe types [68]. In this section, we consider only single-ownership variants of these aliasing policies; multiple ownership was discussed in details in Section 4.8 above.

In Section 2.1.1.3, we identified and described three variants of owners-as-dominators and owners-as-modifiers: strict, peer and transitive owners-as-dominators and owners-as-modifiers. We discuss these variants in the next three sections and show how aliasing contracts can be used to model each of them.

Although most of the ownership-based systems described in Section 2.1.1.3 restrict references, we talk only about object accesses via these references here. (As we noted above, both approaches give equivalent encapsulation guarantees.) Owners-as-dominators restricts both object reads and writes (or references), while owners-as-modifiers limits only object writes (or references).

4.10.2.1 Strict owners-as-dominators and owners-as-modifiers

Strict owners-as-dominators and owners-as-modifiers is the simplest version of this aliasing policy. It allows an object to be accessed only by its direct owner (although it

[1]This assumes that the aliasing contract implementation eliminates cyclic contract evaluation; otherwise, contract evaluation may of course loop forever.

is usual practice to also allow an object to access itself). We can easily model strict owners-as-dominators by annotating fields with the rw-contract "`accessor == accessed || accessor == this`". Strict owners-as-modifiers is then represented by the contract "`true, accessor == accessed || accessor == this`". The condition "`accessor == accessed`" ensures that an object can access itself, while "`accessor == this`" gives access permissions to the object's owner.

4.10.2.2 Peer owners-as-dominators and owners-as-modifiers

Peer owners-as-dominators and owners-as-modifiers additionally allows objects with the same owner (that is, peers) to access each other. This is most easily modelled using encapsulation groups.

In each class, we define an encapsulation group `peers` and add all of the class' fields to the group definition. Thus, each object's `peers` group contains the object's directly owned objects:

```
group peers = {f1, f1.peers, ..., fn, fn.peers};
```

We then annotate each field with the rw-contract "`accessor == this || accessor in peers`" for owners-as-dominators or with the contract "`true, accessor == this || accessor in peers`" for owners-as-modifiers. This contract ensures that an object can be accessed only by its peers (including itself) and its owner.

We can also specify this contract without encapsulation groups. For a class with fields `f1` to `fn`, each field's contract becomes "`accessor == this || accessor == f1 || ... || accessor == fn`". Although semantically equivalent to the contract using encapsulation groups above, this contract is significantly more verbose. It is also less maintainable: adding a new field to a class requires modifications to each existing field contract, rather than just the encapsulation group definition.

4.10.2.3 Transitive owners-as-dominators and owners-as-modifiers

Transitive owners-as-dominators and owners-as-modifiers allows an object to be accessed by all objects below it in the ownership tree (that is, all objects it owns directly or transitively), in addition to its peers and its owner. This means that object accesses may go any number of levels up the ownership tree but at most one level down. We can model this aliasing policy with a simple modification to the `peers` encapsulation group from the previous section.

To highlight the difference between transitive and peer owners-as-dominators and owners-as-modifiers, we name the encapsulation group `repGroup` instead of `peers` in this section. Each object's `repGroup` group contains the object's representation – all objects it owns directly or transitively. For a class with fields `f1` to `fn`, we define `repGroup` as follows:

```
group repGroup = {f1, f1.repGroup, ..., fn, fn.repGroup};
```

This group contains all directly owned objects (`f1` to `fn`) and all objects owned transitively through these directly owned objects (`f1.repGroup` to `fn.repGroup`).

We can now use the same contracts as in the previous section: "`accessor == this || accessor in repGroup`" for owners-as-dominators and "`true, accessor == this || accessor in repGroup`" for owners-as-modifiers.

4.10.2.4 Clarke-style ownership types

We now look in detail at how aliasing contracts can model Clarke-style ownership types, arguably the most influential ownership-based alias protection scheme, which we described in detail in Section 2.1.1.3. Clarke-style ownership types is a transitive owners-as-dominators system and therefore the contracts required to model it are very similar to those discussed in the previous section. The main difference is the introduction of `norep` and `owner` annotations which produce different encapsulation semantics than the standard transitive owners-as-dominators `rep` annotation.

Clarke-style ownership types describe a system's ownership structure using three different annotations. The variable annotation `rep` indicates that the object stored in the variable is owned by the variable's declaring object (and that transitive owners-as-dominators semantics apply). Variables annotated with `norep` contain shared objects accessible anywhere in the system. Objects stored in `owner`-annotated variables have the same owner as the variable's declaring object.

In the previous section, we defined contracts and encapsulation groups to model the semantics of the `rep` annotation only. Here, we need to modify the encapsulation group structure to accommodate the additional `owner` annotation.

We define an encapsulation group called `repGroup` in each class as before; in addition, we define `ownGroup` which at run time contains all directly and transitively referenced objects that have the same owner (that is, objects that are reachable through `owner`-annotated references). For a class with `owner`-annotated fields `f1`, ..., `fn` and `rep`-annotated fields `h1`, ..., `hn` the group definitions are as follows:

```
group ownGroup = {f1, f1.ownGroup, ..., fn, fn.ownGroup};
group repGroup = {h1, h1.repGroup, h1.ownGroup, ...,
                  hn, hn.repGroup, hn.ownGroup};
```

The group `repGroup` contains all directly owned objects (`h1`, ..., `hn`), as well as all objects owned transitively through these directly owned objects (`h1.repGroup`, ..., `hn.repGroup` and `h1.ownGroup`, ..., `hn.ownGroup`), assuming that the owned objects' `repGroups` and `ownGroups` have been specified correctly.

A Clarke-style ownership annotation can then simply be replaced by the corresponding aliasing contract:

```
⟦rep⟧    =   {accessor == this || accessor in repGroup}
⟦norep⟧  =   {true}
⟦owner⟧  =   {accessor canread this, accessor canwrite this}
```

The contract ⟦`rep`⟧ gives transitive owners-as-dominators semantics, as explained in the previous section.

The contract ⟦`norep`⟧ evaluates to `true`, regardless of the value of `accessor`, making objects stored in `norep` variables accessible from anywhere in the system.

Finally, ⟦`owner`⟧ captures the semantics of the `owner` annotation, where an object has the same owner (and can be accessed by the same objects) as the contract's declaring object. When the object in an `owner`-annotated field is accessed, contract evaluation continues to the contract's declaring object ("`this`"), as explained in Chapter 3. For example, consider the following simple `LinkedList` which uses the `owner` annotation:

```
class LinkedList {
  Node head ⟦rep⟧;
}
class Node {
  Node next ⟦owner⟧;
}
```

When a `Node` in the list is accessed, evaluation of the ⟦`owner`⟧ contract causes contract evaluation for the previous `Node` in the list; this continues until the first `Node` in the list is reached. At this point, the ⟦`rep`⟧ contract of the `head` field of `LinkedList` is evaluated. This contract evaluation occurs no matter which node in the list is accessed, thus giving all nodes in the same list the same owner (the `LinkedList`).

Clarke-style ownership types additionally include *context parameters* (as explained in Section 2.1.1.3) which can be instantiated as `rep` or `norep`. Using monomorphisation, we can simply eliminate context parameters and model the resulting program as described above. Alternatively, we can directly model them using contract parameters; this is explained in detail in Appendix B.

4.10.3 Uniqueness and linearity

Uniqueness stipulates that only a single reference can exist to a *unique* object, as discussed in Section 2.1.1.2. This cannot be directly modelled using aliasing contracts, which are concerned only with the usage of references but not their existence.

Instead, aliasing contracts can express the constraint that accesses to an object may originate only within a single `accessor` object; in addition, it is usually useful to allow an object to access itself. This is in fact identical to strict owners-as-dominators as described in Section 4.10.2.1 above.

The rw-contract "`accessor == accessed || accessor == this`" ensures that accesses to an object can only come from a single `accessor` object (and from the object itself). The single `accessor` object may make accesses through any reference it holds (there may be multiple ones) but accesses from other objects would result in an error. Thus, this contract does not restrict accesses to a *single reference*, but to a *single object*.

Another difference between the semantics of standard uniqueness and those of aliasing contract is that annotating a variable with the rw-contract "`accessor == accessed || accessor == this`" does not necessarily guarantee access. Other references to the same object with conflicting contracts may exist; for example, several objects may hold a reference with rw-contract "`accessor == accessed || accessor == this`" to the same object. In this case, none of the variables could be used to access the object, since this would inevitably lead to a violation of one of the conflicting contracts. Annotating a reference with rw-contract "`accessor == accessed || accessor == this`" means that the associated object can be accessed by either one or zero objects; the contract prohibits accesses from multiple objects but does not guarantee that any access rights will exist at all.

Linearity, also described in Section 2.1.1.2, ensures that a variable is used exactly once. Again, we cannot model these semantics with aliasing contracts, since they concern references, not object accesses. However, we can use aliasing contracts to enforce the constraint that an object is accessed at most once. This may be useful for inherently linear objects, such as input streams, which are consumed when accessed.

To enforce linear access to an object, we store the object in a *protecting variable* with rw-contract "`false`" once it has been accessed, thus ensuring that it cannot be accessed again in the future. This variable must be a field; a local variable is therefore not sufficient for our purposes here. The reason for this is that a contract disappears when the associated reference goes out of scope and in this case we want the contract to persist as long as possible.

In the example below, we have a linear object stored in field `lin`. Method `foo` accesses this object at program point `A` by calling one of its methods, `bar`. As soon as this access has occurred, we assign the linear object to the protecting variable `protectingVar` with rw-contract "`false`" at program point `B`. This contract protects the linear object from any further accesses:

```
LinearObject lin {accessor == accessed || accessor == this};
Object protectingVar {false};
foo() {
  lin.bar();                                           //A
  protectingVar = lin;                                 //B
}
```

Note that we are actually modelling affine linearity, where the object must be accessed at most once, instead of standard linearity, which would ensure that the object is accessed exactly once. Boyland et al.'s capabilities [21] can also be used to model affine, but not standard, linearity.

Inserting assignments to protecting variables, such as `protectingVar` in the above example, directly after a linear object has been accessed can easily be done automatically at compile time.

4.10.4 Module encapsulation

Module encapsulation, as described in Section 2.1.1.4, encapsulates objects inside an entire module, instead of inside other objects. An encapsulation module could, for example, be a single type (making encapsulated objects accessible only to all instance of that type) or a program module such as a package in Java (making encapsulated objects accessible only to objects of types declared in that module).

Module encapsulation is primarily a static concept: it is easy to determine the module or class of a particular piece of code at compile time by looking at the class or module in which it is physically written. At run time, however, this does not work in the same way: we can easily determine the type of an object and the module in which this type is declared. However, the code being executed could physically reside in any of the supertypes and associated modules. Encapsulating an object inside a *single* type or module does not make sense at run time, since a single object may at runtime have many different types.

This requires an adjustment to the (inherently static) semantics of module encapsulation. Instead of encapsulating an object inside a single type, we encapsulate it inside a type and all of its subtypes. We use the `instanceof` operator: for example, the rw-contract "`accessor instanceof Foo`" ensures that the associated object can be read and written only by objects of type `Foo` or objects of a subtype of `Foo`.

We can encapsulate an object inside a module in a similar way. For simplicity, we assume the existence of a boolean `inmodule` operator, which is analogous to `instanceof`: `accessor inmodule M` is `true` if there is a type `T` in module `M` such that `accessor instanceof T`. Then, the rw-contract "`accessor inmodule M`" achieves module encapsulation: evaluation succeeds if the type of `accessor` or one of its supertypes is declared in module `M`.

This shows how easy it is to model module encapsulation using aliasing contracts (although the semantics of compile-time and run-time module membership necessarily differ). In contrast, module encapsulation in type universes [68] is modelled using complex typing rules. Confined types [94] enforce module encapsulation by placing a significant number of restrictions on method calls, assignment statements and visibility of classes and fields.

4.10.5 Capabilities

Capabilities are run-time values or access rights which one must quote to be able to indirect on a given reference. Boyland et al.'s work presented in their paper *Capabilities for sharing: A generalisation of uniqueness and read-only* [21], is arguably the aliasing protection scheme which is conceptually most similar to aliasing contracts. Like aliasing contracts, it can be used to model a range of other aliasing policies such as uniqueness and immutability. In this section, we compare and contrast the two approaches.

In Boyland et al.'s work, each reference has an associated capability. This capability specifies the reference's access rights for the object to which it points. For example, if a reference which points to object *o* holds a read capability, this means that the reference can be used to read *o*. If the reference does not have a write capability, it cannot be used to write *o*. Holding a particular capability *guarantees* access rights to an object; a reference with a read capability can always read the associated object. This is not the case for aliasing contracts, where a conflicting contract can interfere.

While holding a capability guarantees access rights in Boyland et al.'s work, not holding a capability does not imply that the associated object cannot be accessed through another reference. For example, if one reference does not hold a write capability for object *o*, this does not mean that *o* cannot be written via a different reference. This is similar to aliasing contracts; just because one part of the system cannot access an object (because of a contract violation), this does not mean that the object cannot be accessed from elsewhere.

These semantics imply that, in both Boyland et al.'s work and our aliasing contracts, we need to consider the union of all capabilities or contracts which currently apply to an object to determine its aliasing properties (for example, to establish whether it is immutable).

A major difference between Boyland et al's capabilities and aliasing contracts is that capabilities specify access rights for a given reference, while aliasing contracts specify access rights for the object holding the reference. For example, an exclusive read and write capability means that the *reference* can be used to read and write the object it contains; reads and writes through other references are not permitted. On the other hand, the rw-contract "`accessor == accessed || accessor == this`", means that only the object itself and the contract's declaring object, "`this`", can read and write the referenced object, but it does not matter which reference is used to do so. This difference is analogous to the difference between standard uniqueness semantics and uniqueness semantics as modelled by aliasing contracts (see Section 4.10.3 above). Thus, Boyland et al.'s capabilities can model standard uniqueness semantics, while aliasing contracts cannot.

Another difference is that Boyland et al. introduce three base capabilities, *read*, *write* and *identity*, while aliasing contracts distinguish only between reads and writes to an object, but do not deal with identity access separately. The reason for this is that aliasing contracts apply only to objects on the heap; contracts are evaluated when the heap region representing an object is read or written. Comparing the addresses stored in two references does not require access to the associated objects on heap and is therefore not currently covered by aliasing contracts. However, introducing the concept of identity access separately from object reads and writes would require only small changes to the operational semantics presented in Chapter 3.

Although Boyland et al. suggest that static checking of some programs using capabilities may be possible (and we similarly consider static verification of aliasing contracts in Chapter 6), the dynamic nature of capabilities and aliasing contracts means that neither can be checked statically in the general case. Any programs which cannot be proved to be correct by static analysis must be run in the presence of dynamic checks. Since the behaviour of both capabilities and aliasing contracts is determined by the conjunction of all capabilities and contracts applying to an object, such a run-time system needs to be able to retrieve the capabilities or contracts associated with each object in the system; this can either be done by storing backpointers or by using an external map recording the capabilites and contracts for each object. Our prototype implementation for aliasing contracts in Java (presented in Chapter 5) maintains such an external map.

Both Boyland et al.'s capabilities and aliasing contracts can be used to model various aliasing policies. However, aliasing contracts are arguably more expressive and high-level; they support arbitrary boolean conditions, while capabilities can specify only whether or not a particular reference has a fixed set of (potentially exclusive) read or write access rights for an object. This higher level of expressiveness makes aliasing contracts more suitable for modelling complex aliasing policies such as owners-as-dominators.

4.11 Using aliasing contracts to compare full encapsulation and Clarke-style ownership types

In this chapter, we have shown that aliasing contracts can model a wide range of existing alias protection schemes. This is useful when we want to compare two systems: we can translate them to aliasing contracts and perform a direct comparison.

For example, consider a comparison between Clarke-style ownership types and a full encapsulation scheme such as Hogg's islands. Both can be modelled using the same contracts, "`accessor in group`" and "`accessor canread this, accessor canwrite this`", suggesting that there is a close formal relationship between them.

For Hogg's islands, we use the contracts "`accessor in transitiveClosure`" for all fields in bridge classes; `transitiveClosure` is an encapsulation group containing all directly and transitively referenced objects. Objects referenced by encapsulated objects must also be encapsulated; we use contract "`accessor canread this, accessor canwrite this`" to achieve this.

Clarke-style ownership types use the rw-contract "`accessor in repGroup`" for owned (`rep`-annotated) objects, where `repGroup` contains all directly and transitively owned objects. Indirectly owned objects (annotated with `owner`) have the contract "`accessor can-
read this, accessor canwrite this`".

Clearly, there is a close correspondence between the two systems; in fact, the full encapsulation of Hogg's islands is a special case of owners-as-dominators as implemented by Clarke-style ownership types, where bridge classes annotate all their fields as `rep` and all other classes annotate their fields as `owner`. In this case, both the encapsulation group structure and the contracts used in the two systems match. From this, we can conclude that full encapsulation is actually a subset of transitive owners-as-dominators.

This example demonstrates that aliasing contracts provide a unifying way to compare seemingly different alias protection schemes.

4.12 Summary

In this chapter, we showed how aliasing contracts can be used to model various aliasing policies introduced by existing alias protection schemes, including full encapsulation, module encapsulation and the three variants of owners-as-dominators and owners-as-modifiers (strict, peer and transitive). This ability to model, and thus subsume, many existing aliasing policies demonstrates the expressiveness of aliasing contracts. It also shows their potential as a unifying scheme which can be used to understand and compare existing work in the area, as we demonstrated by comparing full encapsulation and Clarke-style ownership types.

CHAPTER 5

JaCon: a prototype implementation of aliasing contracts for Java

In this chapter, we present JaCon, a prototype implementation of aliasing contracts in Java. JaCon supports all of the features of aliasing contracts we proposed in Chapter 3, including the `canread` and `canwrite` operators, encapsulation groups, contract parameters and contract suspension.

The aliasing contracts proposed in Chapter 3 assumed very simple programming semantics. In Section 5.1 of this chapter, we start by extending this original proposal to real programming languages (and their more advanced features). In Section 5.2, we give an overview of JaCon, which includes a modified Java compiler and a run-time library implementing dynamic contract evaluation. Finally, in Section 5.3 we present the results of a performance evaluation of JaCon, including case studies of artificial programs designed to exhibit particular contract evaluation behaviour and case studies of large real-world Java programs. The results of the performance evaluation are very encouraging, as they show that JaCon's performance is comparable with existing debugging tools, such as Valgrind [72].

5.1 Extensions of aliasing contracts for real-life OO languages

The theory of aliasing contracts presented in Chapter 3 is clean and simple – every object access causes a contract evaluation – but real programming languages have more complex features that we need to account for.

In particular, `static` methods do not fit well with our object-based approach. For object accesses from `static` methods, there is no `accessor` object; in calls to `static` methods, there is no `accessed` object.

We address this problem by always allowing calls *to* `static` methods, since no `accessed` object means that there are no contracts which need to be evaluated; they do not represent actual object accesses. In accesses *from* `static` methods, we set `accessor` to `null`; this causes contracts such as "`accessor == this`" to fail, while contracts which do not use `accessor` (such as "`true`") behave as expected.

We allow contracts to be declared for `static` fields (which contain objects that we may want to encapsulate); however, we stipulate for obvious reasons that such contracts can refer only to `static` members of the enclosing class.

Many modern OO languages include variables of primitive types, which store values instead of references to objects. Values cannot be aliased and therefore accessing them does not require contract evaluations in JaCon.

Other language features, on the other hand, do not require special treatment due to the dynamic nature of aliasing contracts. For example, objects of inner classes can be treated just like other objects. Inheritance also fits naturally with aliasing contracts. Fields are inherited by subclasses, along with their contracts, but cannot be overridden; to fit with existing inheritance semantics, we similarly disallow the overriding of contracts in subclasses.

5.2 Description of JaCon

We have implemented a prototype, JaCon, which supports the definition of aliasing contracts in Java and performs contract evaluation at run time. The prototype consists of a modified Java compiler and a run-time library (which we call the *contract library* below). The compiler injects calls to the contract library into the source code, allowing the contract library to monitor contracts at run time.

As an alternative approach, we considered modifying the Java Virtual Machine to track contracts at run time. The advantage of such an approach is that it does not require a modified compiler, only a specialised virtual machine. On the other hand, code compiled with JaCon requires only the presence of the contract library to run on any standard Java Virtual Machine. Another advantage of our approach is that we can inspect the generated source code to see exactly what code will be executed by the virtual machine, making it easier for us to trace problems and understand programs' precise behaviour.

We chose Java as our base programming language since it is currently the most popular OO programming language [92] and is used in a large number of open-source systems. However, Java is a relatively complex language with many different features, making the prototype implementation non-trivial. For example, Java's non-linear execution flow, caused by exceptions, `break` and `continue` statements, complicates the tracking of contracts, as we discuss below.

Although not described in detail here, our contract library implementation can handle programs with multiple threads. Its methods and data structures are synchronised in such a way that addition, removal and evaluation of contracts work correctly, even when there are concurrent accesses from different threads.

5.2.1 Prototype features

For our prototype implementation, we modified the compiler *javac* of the OpenJDK 6 [78] to inject calls to the contract library into a program's source.

The syntax of contracts in JaCon is identical to the syntax proposed for aliasing contracts in Chapter 3. Syntactically, a contract consists of one or two boolean expressions[1],

[1] As in Chapter 3, one contract expression can be omitted when both contract expressions are identical.

separated by a comma and enclosed in braces; contracts can use the special variables `accessor` and `accessed`, as well as the special operators `canread`, `canwrite` and `in`.

Contracts are parsed by the JaCon compiler and converted into anonymous inner classes extending the abstract class `Contract`, as shown in the example below; one such `Contract` class is created for each syntactically distinct contract expression in a class. Each `Contract` class overrides the method `checkContract`, which can be called by the contract library to evaluate the contract at run time. Method `checkContract` takes two parameters, `accessor` and `accessed`, both of type `Object`; the contract expression becomes the method's boolean return statement. Revisiting the `LinkedList` example from previous chapters (see, for example, Section 3.1), the rw-contract "`accessor == accessed || accessor == this`" of the `head` field of `Node` is transformed into:

```
public Contract _contractLinkedList42 = new Contract() {
  public boolean checkContract(Object accessor, Object accessed) {
    return accessor == accessed || accessor == LinkedList.this;
  }
};
```

We note that any references to `this` in the contract expression must be transformed to `OuterClass.this` in order to refer not to the `Contract` object but to the contract's declaring object; in the example above, `this` becomes `LinkedList.this`.

Converting contracts into inner classes as shown above automatically enforces the restriction that contracts can only refer to fields and methods of `this`, but not to local variables and method parameters. In addition, the compiler automatically checks that the contract expression is `boolean` (as the return type of `checkContract` is `boolean`).

To lessen the annotation burden on programmers, JaCon allows contracts to be omitted completely; it uses default contracts when no contract is specified for a variable: "`true, accessor == accessed || accessor == this`" for non-public fields and "`true`" for `public` fields, all `static` variables and local variables and method parameters. In addition, JaCon defines four "special contracts" and provides keywords to abbreviate them:

- `shared` – "`true`";
- `const` – "`true, false`";
- `owned` – "`accessor == accessed || accessor == this`"
- `writeowned` – "`true, accessor == accessed || accessor == this`".

For example, the declaration `private Node head` {`accessor == accessed || accessor == this`} can be abbreviated to `private owned Node head`.

Contracts are registered and de-registered when an assignment occurs: for example, assignment `head = newNode` (in method `addItem` of `LinkedList` – see Section 3.1) points `head` away from its current object and to the object currently also pointed to by `newNode`. This change to the program's aliasing structure requires modification of the set of contracts associated with these two objects: JaCon inserts two calls to the contract library, one to de-register the contract of variable `head` before the assignment and one to register the contract of `head` after the assignment. The assignment `head = newNode` becomes:

```
ContractLibrary.removeContract(head, _contractLinkedList42);
head = newNode;
ContractLibrary.addContract(head, _contractLinkedList42);
```

This corresponds to the reduction rules (*field-put-⊙*) for field writes and (*local-put*) for local variable writes in the operational semantics presented in Chapter 3.

We must be particularly careful with the first argument of `addContract` and `removeContract`, since this expression could contain method calls with side effects. For example, the assignment `myIterator.next().foo = newFoo` becomes:

```
ContractLibrary.removeContract(myIterator.next().foo, _contract0);
myIterator.next().foo = newFoo;
ContractLibrary.addContract(myIterator.next().foo, _contract0);
```

Assuming the `next` method of the iterator advances the iterator to the next position in the list, the modified code (with calls to `removeContract` and `addContract`) moves the iterator forward by three positions instead of just one. To avoid this problem, JaCon introduces an intermediate variable declaration whenever it encounters a method call in the left-hand-side expression of an assignment:

```
Object _o1 = myIterator.next();
ContractLibrary.removeContract(_o1.foo, _contractBar0);
_o1.foo = newFoo;
ContractLibrary.addContract(_o1.foo, _contractBar0);
```

This produces the desired semantics.

Registration and de-registration of contracts is complicated by Java's non-linear flow of execution caused by, for example, exceptions. Contracts of local variables have to be removed at the end of the block in which they are declared; at this point, local variables go out of scope and their contracts should not persist. Exceptions are problematic because an exception causes execution to leave a block prematurely; contracts of local variables must still be de-registered in this case. We therefore wrap each block in each method in a `try-finally`-block and remove local variable and method parameter contracts in the `finally`-block.

While contracts of local variables are deallocated at the end of a method's execution, field contracts must persist as long as their declaring objects persist. In Java, objects exist until there are no more references to them and they are eventually finalised by the GC (at an unspecified time). This inherent uncertainty about how long objects exist significantly influences the semantics of aliasing contracts and indeed Java finalisation itself. When should the associated contracts be removed? This is connected to the somewhat philosophical question about how long an object exists; does the object "die" when it becomes unreachable or when it is finalised by the GC?

In our operational semantics in Chapter 3, we side-stepped this issue and said that objects continue to exist until the program finishes executing; thus, contracts of fields persist as well. However, such an approach is clearly not feasible in practice.

We could remove the contracts of an object's fields when the object is no longer reachable. However, this is complex to implement, requiring sophisticated tracking of references beyond simple reference counting. Alternatively, contracts could persist (along with the object's references) until the object is collected by the GC.

We take the second approach in our implementation for practical reasons and because it fits better with Java's approach to references. An object's fields remain in memory until finalisation by the GC, and so do the fields' contracts. This means that the contract of an object waiting to be finalised can cause contract violations. Contract evaluation is monotonic with respect to GC: GC can remove contract violations but never introduce them.

Contract registration and de-registration as explained above allows us to track which contracts apply to each object at any point in the program's execution. This tracking of contracts is equivalent to tracking the references to each object, which in itself is potentially useful; this means that JaCon could also be used as a tool for investigating the topology of the heap, independent of aliasing contracts.

Calls to the contract library to check contracts are required before each object access. JaCon inserts such calls before field reads and writes (as in the operational semantics in Chapter 3). Reads and writes of local variables do not require contract evaluation; this is consistent with the reduction rules (*local-get*) and (*local-put*) of the operational semantics in Chapter 3.

For example, the assignment `x.f = y.g` contains a write to `x` and a read to `y`; it becomes:

```
ContractLibrary.checkWriteContracts(x);
ContractLibrary.checkReadContracts(y);
x.f = y.g;
```

As we noted in Chapter 3, the order in which contract evaluations are performed does not matter, since one contract evaluation cannot impact the result of another.

The `canread` and `canwrite` operators trigger read and write contract evaluations respectively, according to reduction rules (*canread*) and (*canwrite*) in the operational semantics in Chapter 3. JaCon translates them to calls to contract library methods `checkReadContracts` and `checkWriteContracts`.

Methods may be declared `pure` or `impure`. If no purity is specified for a method, JaCon automatically determines its purity. Calls to `pure` methods require read contract evaluations, while calls to `impure` methods need to trigger both read and write contract evaluations. Appropriate contract checks are inserted at the entry to each method's body (rather than in the caller).

As for calls to `addContract` and `removeContract` above, JaCon may need to declare intermediate variables to avoid methods with side-effects being invoked multiple times.

JaCon does not add contract evaluations for object accesses inside contracts. This is a departure from our operational semantics proposed in Chapter 3 which included contract checks for all object accesses, even those inside contracts. Not evaluating object accesses in contracts is more efficient and avoids the problem of potentially cyclic contract evaluation, making it a more practical approach for JaCon.

Conditionals and loops complicate contract registration, de-registration and evaluation. In Java, the conditions of `if`-statements, `switch`-statements and various loops may contain expressions (which require contract evaluations), assignments (which require contract registration and de-registration) and, in the case of `for`-loops, variable declarations. The situation is complicated further by the presence of `break` and `continue` statements in the loop or conditional body. Although we could insert calls to the various contract library methods at the appropriate points in the loop or conditional, this would create highly complex code, as we need to account for all possible execution flows.

A simpler solution involves putting the required calls to contract library methods directly in the condition of the loop or conditional. This ensures that they are executed exactly when the condition is also executed. For example, consider the following `if`-statement, where `y` is a local variable and `f` is a field:

```
if(y != null && y.f != null) {
  ...
}
```

JaCon converts this if-statement to:

```
if(y != null && (ContractLibrary.checkReadContracts(y)
    && y.f != null)) {
  ...
}
```

Because of Java's lazy evaluation of the && operator, this version executes the contract evaluation for the second part of the condition only if the first part succeeds.

This approach works for if-statements, while-loops and do-while-loops, but not for switch-statements, for-loops and for-each-loops, all of which have non-boolean expressions in their conditions that cannot be chained together with contract evaluation as above. This makes the implementation of aliasing contracts significantly more complex for these constructs[2].

JaCon includes support for encapsulation groups, with a syntax similar to that proposed in Chapter 3. However, we change the keywords group and in to encapsgroup and ingroup here, since we found that group and in are already frequently used as variable names in real-world programs.

Encapsulation groups are translated into boolean methods, which, given an object as an argument, will return true if the object is in the group and false otherwise. The ingroup operator can then simply be transformed into a call to the method representing the required encapsulation group.

JaCon also allows the definition and use of parameterised contracts. Unlike normal contracts, parameterised contracts are not translated into anonymous inner classes but are instead passed around as simple Strings. The contract library tracks the actual values of the contract parameters for each object, as supplied when the object is instantiated.

When the methods addContract and removeContract receive a parameterised contract as an argument, they simply look up the actual contract corresponding to the contract parameter name. The methods checkReadContracts and checkWriteContracts can then evaluate them like any other contract. This approach is consistent with our operational semantics in Chapter 3, where parameterised contracts are also resolved before being stored in the contract store.

Contracts can be temporarily suspended in JaCon using a suspend-block, as suggested in Chapter 3. JaCon replaces a suspend-block with calls to the contract library methods suspendContract and reinstateContract. For example, suspend(x) {...} becomes:

```
ContractLibrary.suspendContract(this, "x");
...
ContractLibrary.reinstateContract(this, "x");
```

[2]In this case, JaCon inserts the required contract library calls in various different places in the loop or conditional. For example, in a for-loop, calls to removeContract are added just before the loop condition is executed (that is, before the loop, at the end of the loop body and directly before continue statements). Calls to addContract are inserted immediately following the execution of the loop condition (that is, at the start of the loop body and directly after the loop – but only provided that no break occurred).

As in our operational semantics, we impose the additional restriction that variables whose contracts are currently suspended cannot appear on the left-hand side of assignments. If the contract library detects reassignment of a variable with a suspended contract, it throws an exception to report the error.

Finally, the contract library needs to keep track of which object is currently executing a method; this gives the value of `accessor` for contract evaluations. For this purpose, it maintains a call stack. The contract library is notified of context changes at the start and end of every method execution:

```
public void foo() {
  ContractLibrary.enterContext(this);
  ...
  ContractLibrary.leaveContext();
}
```

5.2.1.1 Optimisations

A naive implementation of aliasing contracts, as described above, performs many unnecessary contract evaluations. We have implemented optimisations to reduce the number of contract evaluations and improve the performance of JaCon.

We expect that the contract expression "`true`" will be used commonly in practice. It signifies that the variable declaring the contract places no restrictions on accesses to the referenced object. Since this contract expression obviously always evaluates to `true`, there is no need for the contract library to store or evaluate it[3].

Evaluating all contracts every time an object is accessed is inefficient. JaCon includes an optimisation which allows it to skip many contract evaluations; it divides contracts into three categories:

- Contracts whose result does not change for different `accessor` objects and is not affected by changes to an object's state. This includes, for example, the contracts "`true`" and "`false`". They need to be evaluated only *once* and can return the same result for subsequent evaluations.

- Contracts whose result changes for different `accessor` objects but which are not affected by changes to an object's state. This includes the contracts "`accessor == accessed`" and "`accessor == this`". These contracts need to be evaluated only *once for each distinct* `accessor` *object.*

- Contracts which depend on values of fields, call methods or refer to encapsulation groups, such as "`accessor == this.f`", "`accessor == getFoo()`" or "`accessor in myGroup`". These contracts need to be re-evaluated every time an object is accessed, since the value of fields (and thus encapsulation group membership) and the return value of methods may have changed since the previous evaluation.

JaCon's contract library classifies contracts as above and uses this information to decide which contracts need to be evaluated when an object is accessed and which contract evaluations can be skipped.

[3]Thus, JaCon does not create `Contract` classes to represent the contract expression "`true`" and does not inject calls to `addContract` and `removeContract` where the variable on the left-hand-side of the assignment has the rw-contract "`true`".

The optimisations above do not fit well with contract suspension: we cannot assume that a contract has been previously evaluated, if it could have been suspended. Instead, the optimisations above are applied only when none of the contracts for the accessed object are suspended. From an efficiency standpoint, it therefore makes sense to limit the use of contract suspension in programs where performance is important.

5.2.2 Known issues

There are two known issues with the current implementation of JaCon which stop valid Java programs from compiling correctly. The first concerns the use of labelled `break` statements which allow breaking directly out of an inner loop or conditional. Our compiler does not currently deal with such `break` statements, as they are quite infrequent in real-world code. We encountered them in two programs we tested and simply restructured the code to remove them.

A second issue is that JaCon occasionally requires renaming of variables. This occurs because JaCon moves around variable declaration when it insers `try-finally`-blocks into methods. Variables declared inside the `try`-block need to be accessible in the `finally`-block as well so that their contracts can be de-registered; therefore, JaCon places variable declarations directly before the `try-finally`-block. This can lead to name clashes when two variables with the same name exist inside previously separately scoped sub-blocks.

This issue could easily be addressed by automatically renaming variables when required. We decided that this was not needed for our prototype implementation given that such name clashes occur very infrequently. We tested JaCon on five real-world Java programs each containing thousands of lines of code (LoC) and found that renaming was usually required in less than ten places per program; in addition, the renaming process is quick and mechanical. We also argue that having two variables with the same name in the same method is bad programming practice – two identical names representing different variables and possibly different concepts is confusing. Therefore, the required renaming may in fact be beneficial for software engineering purposes.

In addition to the issues described above, JaCon's compiler could benefit from better error reporting. In some cases, error messages are cryptic, making JaCon somewhat difficult to use for developers with little knowledge of the `javac` compiler. However, at this stage, JaCon is primarily intended as a proof-of-concept and for this reason we have not expended additional effort on improving error reporting.

5.3 Performance evaluation of JaCon

Using JaCon, we now quantify the performance overhead of aliasing contracts by conducting three empirical studies. First, we use a simple example program to show how the time required for contract evaluation increases with the number of contracts associated with the accessed object. We then investigate the performance of encapsulation groups, measuring the performance of the `ingroup` operator as the number of objects in the encapsulation group increases. Finally, we apply JaCon to five open-source programs totalling almost 500,000 LoC and measure the impact of aliasing contracts on their performance. This gives us a good indication of how JaCon can be expected to perform on real software.

All performance measurements were conducted on a Windows 7 laptop with 8GB of RAM and a 2.5GHz Intel Core i5 processor.

5.3.1 Performance of a single object access

Whenever an object is accessed, all contracts associated with the object must be evaluated. We thus expect the time required for the object access to increase with the number of contracts associated with the accessed object (assuming that the contracts cannot be optimised and all need to be evaluated by JaCon). If we assume that each contract is a simple boolean condition which can be evaluated in constant time, contract evaluation time should increase linearly with the number of contracts evaluated.

In Section 5.2.1.1, we presented optimisations which allow JaCon to skip some contract evaluations. We expect to see a significant performance gain when these optimisations can be applied.

We perform a simple experiment to test these two hypotheses; the experiment characterises the time required for contract evaluation for varying numbers of contracts and varying contract expressions.

5.3.1.1 Method

To measure the time required to perform an object access, we construct a simple test program, shown below:

```
public class SimpleTest {
  public void run() {
    Bar b = new Bar();
    Foo[] foos = new Foo[NUM_OBJECTS];
    for(int i = 0; i < NUM_OBJECTS; i++) {
      foos[i] = new Foo();
      foos[i].bar = b;                    //A
      b.obj = new Object();               //B
    }
  }
}
class Foo {
  public Bar bar {CONTRACT};
}
class Bar {
  public Object obj;
}
```

The program executes a loop, adding one reference (and hence one contract) per iteration to the `Bar` object stored in variable `b`. This happens in the assignment `foos[i].bar = b` at program point `A` in the above program. The assignment (field update) `b.obj = new Object()` at program point `B` then performs a write to the `Bar` object in `b`, causing all contracts associated with it to be evaluated. By measuring the time taken for this access on every iteration, we can measure how the performance varies with the number of contracts associated with the accessed object.

We also vary the contracts of the accessed `Bar` object (marked as `CONTRACT` in the code above) to see how different contract expressions influence object access time.

n	Object access time (in ms)		
	`alwaysTrue()` (always succeeds)	`alwaysFalse()` (always fails)	`accessor == accessor` (always succeeds)
0	0	0	0
5,000	0.84	0.00099	0.00011
15,000	3.93	0.00038	0.00032
25,000	6.82	0.00084	0.00074
35,000	8.93	0.00060	0.00038
65,000	15.11	0.00040	0.00068
95,000	19.37	0.00054	0.00056

Table 5.1: Time in milliseconds per object access for varying number of contracts n and varying contract expressions

We use the rw-contracts "`alwaysTrue()`" (which calls a method that returns `true`) and "`alwaysFalse()`" (which calls a method that returns `false`); they involve method calls and their evaluation can therefore not be optimised by JaCon. Rw-contract "`accessor == accessor`", on the other hand, only needs to be evaluated once for each `accessor`; due to JaCon's optimisations, we expect it to perform significantly better.

5.3.1.2 Results

Table 5.1 presents the results of our measurements. It shows the number of milliseconds required for a single object access depending on the number of contracts associated with the accessed object. The table shows results for three different contracts as outlined above.

JaCon cannot optimise the evaluation of rw-contracts "`alwaysTrue()`" and "`always-False()`" since they involve a method call. These contracts need to be re-evaluated for each object access. For rw-contract "`alwaysTrue()`", evaluation time thus increases linearly with the number of contracts[4], adding around two milliseconds for every 10,000 contracts. When using rw-contract "`alwaysFalse()`" the very first contract evaluates to `false`, making it unnecessary to check the remaining contracts due to the lazy evaluation of contract conjunction. Thus, the time required for each object access is very low and does not change as the number of contracts increases.

The rw-contract "`accessor == accessor`" clearly always evaluates to `true`. The contract depends on the value of `accessor` but is not affected by changes to the object's state. Thus, each contract needs to be evaluated only once per `accessor`. Since `accessor` is always the same in our example (the `SimpleExample` object running the loop), each contract needs to be evaluated exactly once. Therefore, only one contract is evaluated for every iteration – the newly added contract. Time taken to access the object therefore matches the "`alwaysFalse()`" case and is not affected by the number of contracts associated with the object.

Our measurements for the above example show that the time taken to access an object increases linearly with the number of contracts for contracts whose evaluation cannot be optimised and where all contracts evaluate to `true`; evaluation time can be constant in

[4]Some fluctuations between measurements are of course expected; we try to decrease such fluctuations by reporting averages of several measurements, both here and in the other case studies below.

the best case, where contract evaluation can be optimised by JaCon or where a contract evaluates to `false`. Contract evaluation appears feasible despite the linear worst case, as long as the number of contracts per object remains low. We believe it is unlikely for more than 10,000 references to point to the same object at once, even in a large program.

Even if there are many references to a single object, the performance presented above is unlikely to occur. In practice, it is difficult to construct a case where all contracts evaluate to `true` but none of them can be optimised. All of the contracts which we expect to be most commonly used can be optimised, including "`true`", "`false`", "`accessor instanceof Foo`" and "`accessor == accessed || accessor == this`" (which JaCon uses as a default contract for non-public non-static fields). This makes contract evaluation efficient even when many contracts are associated with a single object.

5.3.2 Performance of encapsulation groups

The contracts considered in the above experiment were simple boolean conditions which can be evaluated in constant time. Contracts involving encapsulation groups are more complex to evaluate, as an encapsulation group can in theory contain an unlimited number of objects. In the worst case, this could be every object in the system.

We expect the time taken to determine whether an object is a member of an encapsulation group to be linear in the number of objects in the group. Encapsulation group evaluation proceeds by visiting each object in the group and checking whether it is the required object; the linear worst case occurs when the required object is either the last member of the encapsulation group to be visited or where the object is not in the group. In the best case, where the required object is the first object which is visited, we expect evaluation time to be constant. We now present an experiment to test these hypotheses.

5.3.2.1 Method

The test program used for this experiment is shown below:

```
public class GroupTest {
  public void run() {
    LinkedList list = new LinkedList();
    Node current = new Node();
    list.head = current;
    for(int i = 1; i < NUM_ITERATIONS; i++) {
      list.dummyObject.innerDummyObject = new Object();  //A
      Node newNode = new Node();
      current.next = newNode;
      current = newNode;
    }
  }
}

public class Node {
  Node next {true};
  encapsgroup nextNodes = {next, next.nextNodes};
}

public class LinkedList {
  Node head {true};
  Foo dummyObject {accessor ingroup allNodes};
  encapsgroup allNodes = {head, head.nextNodes};
  public LinkedList() {
    dummyObject = new Foo();
  }
}

public class Foo {
  Object innerDummyObject {true};
}
```

The test method `run()` in class `GroupTest` iterates to build up a `LinkedList` of `Nodes`. Every iteration, at program point `A`, it accesses a dummy object in the `LinkedList`; this dummy object's contract ("`accessor ingroup allNodes`") triggers an evaluation of the encapsulation group called `allNodes` which contains all nodes in the `LinkedList`. The size of the `LinkedList`, and thus the size of the `allNodes` group, grows by one per iteration. By measuring the time taken to perform the access to the dummy object, we can measure the time taken to search the encapsulation group for the `accessor` object, as its size increases.

To avoid other contract evaluations from affecting the measurements, we give all other variables in the program the rw-contract "`true`".

In the test program shown above, the object we search for in the encapsulation group (`accessor` – an instance of `GroupTest`) is never found, producing the worst-case behaviour, where every object in the group is visited.

To produce the best-case behaviour, we use a second version of the test program in which the required object is visited first when `allNodes` is searched. In this version of the test program, we add the instance of `GroupTest` to the encapsulation group `allNodes` in `LinkedList`; the `LinkedList` class from above becomes:

```
  public class LinkedList {
    Node head {true};
    Foo dummyObject {accessor ingroup allNodes};
    GroupTest test {true};
    encapsgroup allNodes = {test, head, head.nextNodes};
    public LinkedList(GroupTest test) {
      dummyObject = new Foo();
      this.test = test;
    }
  }
```

When constructing the `LinkedList`, `GroupTest` simply passes itself as an argument to the `LinkedList` constructor: `LinkedList list = new LinkedList(this)`.

5.3.2.2 Results

Table 5.2 shows the results of our measurements. We can see that, as expected, the time taken to evaluate group membership grows linearly in the worst case. In our experiment, we found that for every 10,000 objects in an encapsulation group around five additional milliseconds were required.

Number of group objects	Worst case (in ms)	Best case (in ms)
0	0	0
5,000	2.03	0.0032
15,000	6.19	0.0087
25,000	11.19	0.0039
35,000	15.99	0.0028
45,000	20.61	0.0033
55,000	28.75	0.0038
65,000	33.17	0.0025
75,000	37.70	0.0027
85,000	42.34	0.0057
95,000	48.39	0.0027

Table 5.2: Time in milliseconds per encapsulation group evaluation for varying numbers of objects in the group

The best-case performance results also confirm our expectations. In this case, group evaluation time is not influenced by the size of the group and remains roughly constant (at between 0.0025 and 0.0087 ms).

Our results from Section 5.3.1 showed that contract evaluation time increases linearly with the number of contracts. If every contract for an object involves an encapsulation group, this would result in quadratic performance in the worst case. However, for this to be really problematic, the number of contracts and the number of objects in each encapsulation group must be quite large. This appears unrealistic in practice. We therefore argue that this is a theoretical worst case which is unlikely to occur in practical situations.

5.3.3 Empirical study with open-source Java programs

The two empirical studies above give an indication of the expected best-case and worst-case behaviours of various contracts. However, these results are not very useful for estimating the performance of real-world programs with aliasing contracts.

We conduct two case studies involving five open-source Java programs: FindBugs[5], JGraphT[6], JUnit[7], NekoHTML[8] and Trove[9]. Table 5.3 shows information about these programs, including version, size and number of test cases.

All of the selected programs are still being actively developed and have been updated in the past two years. In addition, all programs include extensive JUnit test suites with large test cases, making them suitable for performance evaluation.

The five programs we selected come from different domains:

- FindBugs is a static analysis tool which identifies potential bugs in code by analysing a program's Java bytecode. The algorithms it uses are complex and build significant

[5]`http://findbugs.sourceforge.net/`
[6]`http://jgrapht.org`
[7]`http://junit.org/`
[8]`http://nekohtml.sourceforge.net`
[9]`http://trove.starlight-systems.com`

Program	Version	Date	Source files	Classes	LoC	Test cases
FindBugs	2.0.2	10/12/2012	1,060	1,689	192,259	161
JGraphT	0.8.3	19/01/2012	181	262	32,899	152
JUnit	4.10	29/09/2011	162	219	11,994	524
NekoHTML	1.9.18	27/02/2013	26	53	11,482	222
Trove	3.0.3	15/02/2013	691	1,572	239,266	548

Table 5.3: Version, size and number of test cases of the test programs

data structures. We therefore expect the performance of FindBugs to be strongly influenced by aliasing contracts.

- JGraphT is a graph library which implements various graph data structures (such as directed and undirected, weighted and unweighted graphs) and associated graph algorithms (including shortest path, vertex cover and chromatic number algorithms). Like FindBugs, it involves large data structures and runs algorithms with high asymptotic complexities. We therefore also expect its performance to be severely impacted by aliasing contracts.

- JUnit is a well-known tool for unit testing in Java. It does not involve large data structures and complex algorithms and we therefore expect JUnit's performance to degrade only slightly in the presence of aliasing contracts.

- NekoHTML is an HTML scanner and tag balancer. As parsing involves a lot of comparatively slow input and output, we do not expect the performance of NekoHTML to be strongly influenced by aliasing contracts.

- The Trove library provides high performance collections for Java. The large data structures and collections built by Trove are likely to result in significant performance degradation in the presence of aliasing contracts.

We now present the details of our two case studies. First, we evaluate the performance of the five test programs by running their unit tests with and without aliasing contracts. This is described in Section 5.3.3.1.

Secondly, we measure the performance of FindBugs for real usage scenarios, as described in Section 5.3.3.2. FindBugs takes Java bytecode as input; we execute FindBugs with bytecode of the above five test programs as input. This represents a real application of FindBugs, giving us reliable performance data.

In addition to providing a good indication of the expected real-world performance of aliasing contracts, these two case studies have given us the opportunity to thoroughly test JaCon. Our modified compiler has generated more than 800,000 LoC. Our contract library has executed many test cases, some of them enormous in size. This demonstrates the robustness of our prototype and its ability to handle real-world software.

5.3.3.1 Case study 1: unit test measurements

In our first case study, we compile and execute the unit tests of the five test programs with and without aliasing contracts and compare their performance.

Method

First, we compile each of the programs four times, using the standard Java compiler (called `javac0` below) and three different versions of the JaCon compiler.

The programs we chose include thousands of LoC, making it impossible to manually annotate them with aliasing contracts. Instead, we measure performance with three different versions of the JaCon compiler, each using different default contracts for variables.

Compiler `javac1` shows what happens when all aliasing contracts are "`true`"; that is, when aliasing is not at all controlled. (This corresponds to standard reference semantics without alias protection.) As the contract "`true`" is not stored or evaluated by the contract library, programs compiled with `javac1` perform no contract additions, removals or evaluations.

Compiler `javac2` uses the rw-contract "`true`" for local variables, method parameters, `public` fields and all `static` variables, and contract "`true, accessor == accessed || accessor == this`" for non-public fields. These contracts are based on the assumption that objects stored in non-public fields are intended to be encapsulated and should therefore not be written by other parts of the system. The contracts added by `javac2` cause relatively few contract violations in most of our test programs, as shown in our results below. This indicates that they give a good approximation of the encapsulation used in the test programs (and therefore probably of any manually added aliasing contracts).

Compiler `javac3` uses the rw-contract "`accessor == accessor`" (which always evaluates to `true`) for all variables. Programs compiled with `javac1` and `javac2` contain many "`true`" contracts which can be ignored by the contract library; they are neither stored nor evaluated. In programs compiled with `javac3`, on the other hand, all contracts must be stored and evaluated. In addition, contract evaluation can never be aborted prematurely when a contract evaluates to `false`.

We argue that the measurements for programs compiled with `javac2` give a good indication of expected performance with manually added contracts; our measurements below show that few contract violations occur when executing the test programs with these default contracts, which thus appear to be a good approximation of the test programs' encapsulation structure. Measurements with `javac1` and `javac3` give a lower and upper performance bound (when only simple, optimisable contracts without encapsulation groups are used).

For each of the four compilations, we record a range of compilation data: compilation time, LoC generated, number of bytes generated and the number of various contract library calls that are injected by the compiler.

LoC (measured at the end of the compilation process) and number of bytes are both measures of the generated program's size. Our size measurements include only the program's source code, but exclude unit tests, sample classes and the contract library code.

Next, we execute the test suites of each test program without aliasing contracts (as compiled by `javac0`) and with aliasing contracts (as compiled by `javac1`, `javac2` and `javac3`), recording various performance measurements.

For each test suite, we measure the time required to run the unit tests. Some unit tests include random elements, leading to slightly different execution every time and causing small variations in measurements.

We also record statistics about the Java GC to get an indication of the memory usage of the programs, looking at the maximum heap size encountered by the GC as well as the total amount of time spent in GC.

Program	Test cases	Code coverage
FindBugs	161	12.6%
JGraphT	152	70.2%
JUnit	524	74.1%
NekoHTML	222	69.5%
Trove	548	7.1%

Table 5.4: Size and code coverage of the test suites of our test programs

During program execution we also count the number of contract additions, removals and evaluations performed by the contract library and the number of failed read or write contract evaluation.

Above, we mentioned that JaCon can be used as a reference tracking tool independently of aliasing contracts. To determine how feasible this is, we also run the unit tests with a modified contract library which skips contract evaluations and performs only contract registration and de-registration. This is equivalent to tracking the references to each object. For this measurement, we use the `javac3` compiler which does not insert any "`true`" contracts; thus the contracts of all variables must be stored by the contract library, tracking all references.

Using unit tests for performance measurements is arguably not ideal since in some cases unit tests will be far less complex than real usage scenarios. However, the level of code coverage and size of the test suites of the five test programs we use (shown in Table 5.4) mitigates this risk. All test suites consist of at least 100 tests. JUnit and Trove have the biggest test suites with more than 500 tests each.

JGraphT, JUnit and NekoHTML all have excellent code coverage of around 70 percent. Given the size and code coverage of these test suites, it is reasonable to assume that performance of the unit tests is indicative of real program performance.

Trove has a large test suite but low code coverage at only around 7 percent. The reason for this low code coverage, however, is the fact that Trove automatically generates a large number of collections. Many of the collections have similar code; for example, Trove generates seven different array list implementations, one for each primitive type in Java. Since these implementations are all generated from the same template, the unit tests only test one of them. Taking this into account, the actual code coverage of Trove's unit tests is significantly higher.

The unit tests of FindBugs also have low code coverage at 12.5 percent and, given its size of almost 200,000 LoC, its test suite includes comparatively few test cases. We should therefore be somewhat careful about drawing conclusions from the performance measurements of these unit tests. A more reliable performance evaluation of FindBugs is presented in Section 5.3.3.2.

Results

From the measurements, we make the following observations:

- Both compilation time and program size (see Table 5.5) increase in the presence of aliasing contracts, This is to be expected, since JaCon's modified compiler injects calls to the contract library, requiring more time and producing a larger program

Program	Compilation time (in s)						
	`javac0`	`javac1`	Ratio	`javac2`	Ratio	`javac3`	Ratio
FindBugs	15.08	20.43	1.35	26.17	1.73	29.31	1.94
JGraphT	4.64	6.82	1.47	6.97	1.50	8.15	1.76
JUnit	2.17	3.46	1.60	3.51	1.62	4.00	1.84
NekoHTML	2.32	3.53	1.52	3.61	1.55	3.77	1.63
Trove	17.21	21.29	1.24	22.95	1.33	25.68	1.49

Program	LoC						
	`javac0`	`javac1`	Ratio	`javac2`	Ratio	`javac3`	Ratio
FindBugs	113,748	231,575	2.04	248,542	2.19	301,209	2.65
JGraphT	11,436	23,031	2.01	25,390	2.22	29,544	2.58
JUnit	7,508	16,074	2.14	17,513	2.33	21,368	2.85
NekoHTML	6,319	12,149	1.92	12,799	2.03	15,633	2.47
Trove	150,810	330,499	2.19	339,987	2.25	375,303	2.49

Program	Kilobytes generated (in kB)						
	`javac0`	`javac1`	Ratio	`javac2`	Ratio	`javac3`	Ratio
FindBugs	6,071	7,230	1.19	8,307	1.37	10,308	1.70
JGraphT	639	791	1.24	950	1.49	1,184	1.85
JUnit	429	522	1.22	625	1.46	824	1.92
NekoHTML	227	279	1.23	310	1.37	402	1.77
Trove	5,040	6,546	1.30	6,840	1.36	7,555	1.50

Table 5.5: Compilation measurements - compilation time and output program size

as output. Our measurements show that the increase in program size is roughly proportional to the increase in compilation time.

We can see that compilation time and program size are lowest when using `javac1` and highest when using `javac3`. Compiler `javac1` gives all variables the rw-contract "`true`"; `javac2` uses non-true contracts for some fields, while `javac3` gives all variables non-true contracts. Since rw-contracts "`true`" are not stored or evaluated by the contract library, compilation is faster and program size smaller the more "`true`" contracts there are in the input program.

The measurements for compilation time show that using JaCon is feasible even for large programs. Compilation of large programs with more than 100,000 LoC, such as FindBugs and Trove, using JaCon adds only 10 to 15 seconds to the compilation process; in addition, the size of the generated programs is very manageable at up to about 300,000 LoC or 10MB – less than three times the size of the program without contracts.

- The number of calls to `addContract` and `removeContract` varies for `javac1`, `javac2` and `javac3`, while the number of calls to `checkReadContracts` and `checkWriteContracts` remains the same; this is shown in Table 5.6. The change in calls to `addContract` and `removeContract` are explained by the varying number of "`true`" contracts for programs compiled with `javac1`, `javac2` and `javac3`.

  In all five programs, around 0.8 calls to the contract library are inserted by `javac3` for every line of original source code. This number is lower for `javac1` and `javac2`, between 0.4 and 0.6, reflecting the smaller number of calls to `addContract` and `remove-`
`Contract`.

- Execution time (see Table 5.7) increases from `javac0` to `javac3`. As hypothesised above, both JGraphT and Trove are significantly affected by the presence of aliasing contracts. JGraphT, for example, runs around 19 times more slowly when compiled with `javac2` and 147 times more slowly when compiled with `javac3`. Even when compiled with `javac1`, JGraphT is slowed down by a factor of 11. Although this version of the program does not store or evaluate contracts, calls to the contract evaluation methods are still performed.

  If we look more closely at the individual tests in the test suites of JGraphT and Trove, we find that the performance degradation is caused by only a handful of tests. The majority of tests run almost as quickly with contracts as without. For example, the worst-performing test suite in JGraphT is called `FibonacciHeapTest`. It builds up a Fibonacci heap, performing 20,000 insertions followed by 10,000 removals, and runs 112 times more slowly when compiled with `javac2` and 330 times more slowly with `javac3`.

  The remaining programs are less strongly affected by the presence of aliasing contracts and are slowed down by a factor of less than 1.4 for `javac1`, less than 1.6 for `javac2` and less than 2.6 for `javac3`. This is expected for NekoHTML and JUnit, but is somewhat surprising for FindBugs which we expected to be strongly affected by aliasing contracts. This result can be explained by the program's relatively small test suite. The case study of FindBugs presented in Section 5.3.3.2 gives more reliable results.

Program	Contract library calls			Contract evaluation calls			% of contract library calls		
	`javac1`	`javac2`	`javac3`	`javac1`	`javac2`	`javac3`	`javac1`	`javac2`	`javac3`
FindBugs	46,336	52,279	92,104	27,210	27,210	27,210	58.72%	52.05%	29.54%
JGraphT	5,321	6,125	8,560	3,111	3,111	3,111	58.47%	50.79%	36.34%
JUnit	3,331	3,632	5,963	1,377	1,377	1,377	41.34%	37.91%	23.09%
NekoHTML	3,387	3,659	5,845	2,515	2,515	2,515	74.25%	68.73%	43.03%
Trove	82,599	85,885	110,445	46,097	46,097	46,097	55.81%	53.67%	41.74%

Program	Add and remove calls			% of contract library calls			Library calls per LoC		
	`javac1`	`javac2`	`javac3`	`javac1`	`javac2`	`javac3`	`javac1`	`javac2`	`javac3`
FindBugs	0	4,963	43,371	0%	9.49%	47.09%	0.41	0.46	0.81
JGraphT	0	664	3,043	0%	10.84%	35.55%	0.47	0.54	0.75
JUnit	0	257	2,554	0%	7.08%	42.83%	0.44	0.48	0.79
NekoHTML	0	188	2,168	0%	5.14%	37.09%	0.54	0.58	0.92
Trove	0	2,726	27,266	0%	3.17%	24.69%	0.55	0.57	0.73

Table 5.6: Compilation measurements - contract library calls injected into the source code

Program	Execution time (in s)								
	javac0	javac1	Ratio	javac2	Ratio	javac3	Ratio	Ref tracking	Ratio
FindBugs	47.32	49.72	1.05	55.22	1.17	125.80	2.66	109.68	2.32
JGraphT	4.640	50.16	10.82	87.92	18.96	680.93	146.82	15.75	3.39
JUnit	14.67	17.60	1.20	19.84	1.35	29.45	2.01	23.29	1.59
NekoHTML	2.14	3.08	1.44	3.50	1.63	5.58	2.61	4.33	2.03
Trove	6.54	44.98	6.87	51.52	9.31	78.63	12.02	28.53	4.36

Program	Maximum heap size (in MB)								
	javac0	javac1	Ratio	javac2	Ratio	javac3	Ratio	Ref tracking	Ratio
FindBugs	37.00	37.01	1.00	76.66	2.07	595.63	16.10	594.56	16.07
JGraphT	34.62	34.06	0.98	639.42	18.47	1,824.56	52.70	2,028.73	58.59
JUnit	33.80	33.85	1.00	66.25	1.96	119.08	3.52	120.58	3.57
NekoHTML	32.64	32.64	1.00	42.46	1.30	91.12	2.79	101.99	3.12
Trove	1,021.97	1,021.90	1.00	1,127.99	1.10	1,140.25	1.12	1,124.85	1.10

Program	Time in GC (in s)								
	javac0	javac1	Ratio	javac2	Ratio	javac3	Ratio	Ref tracking	Ratio
FindBugs	0.23	0.24	1.04	0.54	2.34	2.54	10.97	1.99	8.57
JGraphT	0.032	0.13	3.93	26.22	815.49	103.10	3206.58	111.60	3470.95
JUnit	0.004	0.017	4.20	0.024	5.98	0.83	210.49	0.81	204.87
NekoHTML	0.0037	0.0042	1.15	0.025	6.88	0.10	27.54	0.11	30.50
Trove	0.044	0.19	4.32	2.71	61.43	10.91	247.38	12.31	279.13

Table 5.7: Run-time performance measurements – execution time and memory usage

Tracking only references but not evaluating contracts is significantly more efficient than full contract evaluation for all of our test programs. We record slow-down factors of between 1.6 and 4.4. This shows that the decrease in performance is caused mostly by contract evaluation, not contract tracking. In addition, this demonstrates that JaCon is also useful as a tool for reference tracking.

The wide range of behaviour we observe here, where the performance of some programs is barely affected by aliasing contracts, while others experience severe decreases in performance, can be explained by the different aliasing properties of the test programs. The main performance issue with aliasing contracts occurs when many variables refer to the same object and re-assignment of variables is frequent. Such conditions occur, for example, in programs building complex data structures (as we can see in the performance of JGraphT's unit tests).

While the performance overhead would clearly make many programs unusable in practice, executing unit tests with aliasing contracts still appears feasible for all our test programs, demonstrating that it is indeed possible to use aliasing contracts as a testing and debugging tool.

- Maximum heap size (see Table 5.7) does not significantly increase for programs compiled with `javac1`, compared to programs compiled with `javac0`. This observation is consistent with the fact that programs compiled with `javac1` do not store any contracts.

  Maximum heap size increases significantly for programs compiled with `javac2` and `javac3`. The absolute increase in heap size is particularly high for JGraphT and Trove. This is consistent with the large increase in execution time we observe for JGraphT and Trove; the remaining programs exhibit a significantly smaller increase in the maximum heap size.

  We see a similar effect in the amount of time required by the GC, where JGraphT and Trove show the largest increase in GC time.

  Somewhat surprisingly, we see an increase in GC time from `javac0` to `javac1` of up to a factor of 4.3; however, we observe no associated increase in maximum heap size, as explained above. Closer investigation shows that this increase in GC time is caused by the call stack tracking in the contract library. Although the absolute size of the call stack is small (and thus does not significantly impact the maximum heap size), it constantly grows and shrinks, creating work for the GC.

  When using JaCon for reference tracking, both maximum heap size and GC time are comparable to execution with `javac3`. This makes sense since both programs register and de-register the same number of contracts, thus requiring the same amount of memory and GC time.

- JGraphT and Trove trigger by far the largest number of contract additions (up to 200 million for JGraphT – see Table 5.8), leading to large memory and execution overheads.

  If we compare the number of contract additions for each program to the number of contract removals, we find a significant gap. Most contracts are deallocated by the GC when their declaring objects are finalised; these contract removals do not trigger calls to the `removeContract` method of the contract library and (since we

Program	Additions				Removals			
	javac1	javac2	javac3	Ref tracking	javac1	javac2	javac3	Ref tracking
FindBugs	0	15,503	1,130,863	1,130,863	0	5,441	921,866	921,866
JGraphT	0	8,416,061	209,651,980	209,923,196	0	5,295,815	49,280,350	49,306,924
JUnit	0	47,646	1,026,146	988,308	0	60	473,650	455,646
NekoHTML	0	11,085	229,374	229,374	0	467	170,916	170,900
Trove	0	2,719,679	15,756,822	15,740,664	0	2,471,682	13,152,880	13,136,680

Program	Read evaluations				Write evaluations			
	javac1	javac2	javac3	Ref tracking	javac1	javac2	javac3	Ref tracking
FindBugs	1,083,689	1,083,514	1,083,552	0	52,337	52,335	52,335	0
JGraphT	206,882,381	206,943,918	206,994,930	0	9,181,740	9,183,227	9,190,091	0
JUnit	618,093	644,204	677,691	0	14,982	16,492	18,689	0
NekoHTML	965,178	965,178	965,178	0	91,950	91,950	91,950	0
Trove	276,027,883	276,041,447	276,043,045	0	6,971,226	6,971,183	6,971,532	0

Program	Failures (read)				Failures (write)				% Failed
	javac1	javac2	javac3	Ref tracking	javac1	javac2	javac3	Ref tracking	javac2
FindBugs	0	0	0	0	0	339	0	0	0.03%
JGraphT	0	0	0	0	0	77,015	0	0	0.04%
JUnit	0	0	0	0	0	789	0	0	0.13%
NekoHTML	0	0	0	0	0	38,163	0	0	3.61%
Trove	0	0	0	0	0	306,205	0	0	0.11%

Table 5.8: Run-time performance measurements – contract library calls during program execution

only count explicit contract removals caused by calls to `removeContract` here) these removals are not counted.

The number of read and write contract evaluations are the same for programs compiled with `javac1`, `javac2` and `javac3`[10]. When tracking only references, no contract evaluations are performed.

The percentage of failed contract evaluations is very low. Clearly, the rw-contracts "`true`" and "`accessor == accessor`" used by `javac1` and `javac3` respectively cannot fail; only the write contract used by `javac2` for non-public fields, "`accessor == accessed || accessor == this`", can cause contract violations. We can see that the failure rate is very low, even for this contract. NekoHTML exhibits the largest failure rate at 3.6 percent, while the failure rate is below 0.2 percent for all other programs. This shows that, although default contracts clearly cannot fully capture the intended encapsulation structure of programs, the default contracts used by `javac2` provide a good starting place.

- The above results show that the performance of aliasing contracts is comparable to existing debugging tools such as Valgrind [72]. Nethercote et al. present an empirical study of Valgrind [72], showing that some programs run up to 58 times more slowly using Valgrind, while the slow-down factor is below 20 for most programs.

  Gordon et al. propose dynamic ownership types, a dynamically-checked alias protection scheme (like aliasing contracts). The authors perform a simple performance evaluation, reporting a slow-down of around 50%, significantly lower than what we have measured here (although comparable to our JUnit and NekoHTML test cases). Dynamic ownership types are much simpler to check than aliasing contracts. Each object access requires a single check to see if the access originated in the object's owner. This may explain the performance gap between Gordon et al.'s measurements and our own. In addition, the benchmark used by Gordon et al. totals only 1,700 LoC, compared to our benchmark of almost 500,000 LoC. Given the huge variation in performance we observed in our measurements here, we are doubtful that Gordon et al.'s performance measurements will generalise to larger programs.

5.3.3.2 Case study 2: A real usage scenario for FindBugs

We also conduct a case study FindBugs with real inputs. This test represents a real usage scenario and we therefore suggest that the performance measurements collected reliably characterise the performance of FindBugs in the presence of aliasing contracts.

Method

FindBugs analyses Java bytecode to report possible errors and bugs. We compile FindBugs twice, once with `javac0` to get `findbugs0` and again with `javac2` to get `findbugs2`. We select `javac2` for this experiment rather than `javac1` or `javac3` as we expect its default contracts to give the best approximation of manually added contracts. We then run the unit tests of each of our test programs with both `junit0` and `junit2`.

It is important to note that we run `findbugs0` and `findbugs2` on programs which are themselves compiled with `javac0`. This means that the test programs include no aliasing

[10]Except for small variations caused by random unit tests.

contracts. Any contract tracking or evaluation (and the associated performance overhead) is confined to `findbugs2` and is not caused by the test programs.

Results

Table 5.9 presents the results of the FindBugs performance evaluation. It shows a significant increase in execution time. The worst performance degradation occurs for Trove (55 times slower with `findbugs2` that `findbugs0`) and FindBugs (48 times slower). For the remaining programs, FindBugs executes less than 15 times more slowly.

These results are caused by the high complexity of FindBugs. This becomes clear when looking at the number of contract addition, contract removals and contract evaluations performed. When analysing itself, FindBugs performs more than 210 million contract additions, 170 million contract removals, 9 billion read contract evaluations and 500 million write contract evaluations. The contract failure rate is again low at around 0.5 percent.

The large number of contracts involved in executing FindBugs is also reflected in the memory usage data: although the maximum heap size required increased by a factor of only between 2.5 and 6.2, the time spent in GC increased by a factor of up to 352. This indicates that the number of contracts stored at any point in time is significant. Furthermore, the large increase in GC time suggests that contracts are created and destroyed frequently.

5.4 Summary

In this chapter, we presented JaCon, a prototype implementation of aliasing contracts for Java. Our performance evaluation shows that using JaCon as a testing tool is certainly feasible. It can handle even large programs and its performance is comparable to other debugging tools such as Valgrind. Future development of JaCon could address some of its current limitations, including fixing problems with labelled `break` statements and developing better error reporting.

JaCon was built as a proof-of-concept and we suspect more attention to low-level implementation would expose further performance improvement in contract tracking and evaluation. In addition, we believe that implementing aliasing contracts as part of the Java Virtual Machine would further decrease performance overheads.

Program	Execution time (in s)			Maximum heap size (in MB)			Time in GC (in s)		
	`findbugs0`	`findbugs2`	Ratio	`findbugs0`	`findbugs2`	Ratio	`findbugs0`	`findbugs2`	Ratio
FindBugs	78.76	3798.38	48.22	2,116	5,282	2.50	7.30	1161.30	159.13
JGraphT	19.47	260.19	13.36	483	1,775	3.68	0.80	61.67	76.66
JUnit	14.62	164.02	11.22	287	1,763	6.15	0.13	45.00	351.50
NekoHTML	14.40	156.99	10.90	289	1,791	6.20	0.18	36.76	203.33
Trove	47.28	2598.11	54.96	1,528	5,129	3.36	3.32	942.30	283.51

Program	Additions	Removals	Read evaluations	Write evaluations	Failures (read)	Failures (write)	% Failed
FindBugs	211,203,920	170,100,573	9,697,190,017	559,826,988	0	41,507,208	0.40%
JGraphT	16,129,716	12,903,824	688,766,458	45,655,146	0	3,329,906	0.45%
JUnit	10,705,950	8,474,153	403,062,190	28,167,954	0	2,031,393	0.47%
NekoHTML	9,419,810	7,608,511	404,982,966	26,231,785	0	1,938,143	0.45%
Trove	156,854,263	124,606,331	5,800,083,820	414,023,842	0	33,039,360	0.53%

Table 5.9: FindBugs run-time performance measurements

CHAPTER 6

Static verification of aliasing contracts

In previous chapters, we proposed aliasing contracts, a highly expressive and flexible alias protection scheme. The dynamic checking of aliasing contracts (and the associated performance overhead) is their main disadvantage compared to existing static alias protection schemes. In this chapter, we present a static analysis that can verify many simple aliasing contracts at compile time. To distinguish between compile-time and run-time concepts, we use the term *contract evaluation* when talking about run-time checking of contracts, while *contract verification* refers to compile-time contract checking.

Aliasing contracts can be arbitrary boolean conditions. It is well known that the result of such conditions cannot in general be determined without executing the program. This implies that there are some aliasing contracts which cannot be verified statically and thus we cannot build a tool to verify *all* possible aliasing contracts. Furthermore, exact verification of aliasing contracts requires a full alias analysis of the program to determine which variables are aliased at each point in the program's execution (and thus which contracts need to be verified); this problem is also undecidable in the general case.

Instead of attempting to verify all possible aliasing contracts, we propose a static analysis which is able to verify some common aliasing contracts, but makes no attempt to verify more complex conditions. Section 6.1 gives an overview of how the analysis works and Section 6.2 formalises the analysis.

Our static analysis can fully verify many existing aliasing policies, including owners-as-dominators (as in Clarke-style ownership types [27]), owners-as-modifiers and full encapsulation, as we show in Section 6.3.

Like many existing alias and pointer analyses, our static analysis is conservative; an error is reported whenever the analysis cannot conclusively show that no contract violations will occur at run time. Thus, programs which pass our analysis are guaranteed to execute without contract violations at run time, but the converse does not hold.

Using our static analysis, aliasing contracts can be partially verified at compile time, while more complex conditions require run-time evaluation. This part-static, part-dynamic approach is analogous to gradual typing [87]. Contracts which can be verified at compile time can subsequently be removed from the program during compilation, reducing the number of contracts which need to be evaluated at run time and improving performance. In Section 6.4, we present StatCon, a prototype implementation of the analysis we Java.

We use StatCon to show that combining static and dynamic contract checking in this way is feasible and produces the expected performance improvements.

6.1 Overview of the static analysis

Our static analysis works in two stages. First, it performs alias analysis, looking at all assignments in a program's source code to determine which variables may be aliased with each other during the program's execution. We choose an Andersen-style may-alias analysis which is context-insensitive, flow-insensitive and subset-based. Such an analysis is sufficient for our purposes of modelling existing alias protection schemes such as Clarke-style ownership types [27].

The flow-insensitivity and context-insensitivity of our analysis means that if two variables x and y are assigned to each other *at any point* in the program's source code, they are *always* considered to be possible aliases. In addition, our analysis derives may-aliasing information: it assumes that two variables are aliased if it cannot prove that the two variables will definitely be disjoint at run time. These conservative assumptions ensure that programs which pass our static analysis will never cause contract violations at run time.

Static typing used to enforce alias protection in existing static alias protection schemes such as Clarke-style ownership types is also context-insensitive and flow-insensitive. It works by restricting assignment statements to ensure that two references with different aliasing properties cannot be assigned to each other, thus avoiding objects being switched from one encapsulation context (or owner) to another. Therefore, our simple flow-insensitive and context-insensitive analysis is a good match with existing statically-checked schemes.

The second stage of the analysis consists of verifying each object access in the program's source code, including field reads, field writes and method calls, as proposed in our operational semantics in Chapter 3. For each source-code object access, our analysis verifies the contracts of all variables which may point to the accessed object (according to the may-aliasing information from the first stage of the analysis). If it cannot prove that a contract will evaluate to `true` at run time, it reports an error.

6.2 Formalisation: aliasing graphs and contract verification

In this section, we describe our static analysis in detail. We start by proposing a simple syntax for our analysis, as shown in Figure 6.1. This syntax is similar to (but slightly simpler than) the syntax for aliasing contracts introduced in Chapter 3. C ranges over class names (C' may also range over the pre-defined empty class `Object`), m over method names, f over field names, x, y, z over local variable names, g over encapsulation group names, v over all variable names (local variables, method parameters and fields) and ξ over special contract variables. Letter p ranges over *access paths*, a sequence of field selections starting from a local variable or `this`. As in the syntax in Chapter 3, we abbreviate, e.g., $\mathit{fieldDef}_1 \ldots \mathit{fieldDef}_n$ to $\overline{\mathit{fieldDef}}$. As our analysis is flow-insensitive, we abstract all control flow with syntax $\overline{\mathit{stmt}}$.

To simplify the analysis explanation below we introduce several further syntactic restrictions. Assignments $p = p'$ require p not to be `this` and are restricted to ANF form;

$$
\begin{array}{rcl}
\mathit{program} & = & \overline{\mathit{classDef}} \\
\mathit{classDef} & = & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{\mathit{fieldDef}}\ \overline{\mathit{groupDef}}\ \overline{\mathit{methodDef}}\ \} \\
\mathit{fieldDef} & = & C\ f\ \{\ \mathit{contractDef}\ \}; \\
\mathit{groupDef} & = & \texttt{group}\ g = \{\ \overline{\mathit{groupMember}}\ \} \\
\mathit{groupMember} & = & f \mid f.g \\
\mathit{methodDef} & = & (\texttt{pure} \mid \texttt{impure})\ C\ m\ \texttt{(}\ \overline{\mathit{varDef}}\ \texttt{)}\ \{\ \overline{\mathit{varDef}};\ \overline{\mathit{stmt}};\ \texttt{return}\ v;\ \} \\
\mathit{varDef} & = & C\ x\ \{\ \mathit{contractDef}\ \} \\
\mathit{contractDef} & = & \phi, \phi \\
\mathit{stmt} & = & p = p \mid x = y.m\texttt{(}\ \overline{z}\ \texttt{)} \mid x = \texttt{new}\ C\texttt{()} \\
p & = & x \mid \texttt{this} \mid p.f \\
\phi & = & \xi\ \texttt{canread}\ \xi \mid \xi\ \texttt{canwrite}\ \xi \mid \xi{==}\xi \mid \xi\ \texttt{in}\ g \mid \phi\ \texttt{\&\&}\ \phi \mid \phi\ \texttt{||}\ \phi \mid \texttt{!}\phi \\
 & & \xi\ \texttt{instanceof}\ C \\
\xi & = & \texttt{accessor} \mid \texttt{accessed} \mid \texttt{this}
\end{array}
$$

Figure 6.1: A simple syntax for static analysis of aliasing contracts

that is, p and p' may contain at most one field selection operator between them (although we relax this requirement in examples). We also simplify method calls to have no parameters and no result by treating $x = y.m\texttt{(}z_1, \ldots, z_n\texttt{)}$ as a sequence of expressions $y.\texttt{par}_1^m = z_1; \ldots; y.\texttt{par}_n^m = z_n; y.m(); x = y.\texttt{ret}^m$ where $\texttt{par}_i^m$ (parameters) and $\texttt{ret}^m$ (result) are fields of the class which defines m[1].

The parameterless syntax for `new` assumes it merely allocates storage. The idiom of *constructor* is achieved by following calls to `new` with a method call appropriately initialising the new object; this call is then also treated as parameterless by the technique above.

For analysis purposes we can also replace local variables x with fields f_x; as a result, the program we are analysing will contain only fields, but no method parameters or local variables. This means that all access paths p start with `this` so below we drop the "`this`" uniformly. The result of the restrictions is that our analyser may consider the program to be a set of methods each consisting of a set of simple field assignments and parameterless calls to methods and `new`.

6.2.1 Definitions

We define two compile-time operators: we write $D <: C$ (as in Chapter 3) to mean D is a subclass of C. We write $C \bowtie D$ if $C <: D$ or $D <: C$; this is useful since, in an OO language, two variables may only alias if one variable's (declared) type is a subtype of the other's (as only then a value common to both types may exist).

We also write $\tau(C\!:\!v)$ to give the *declared* type of variable v in class C; at run time v may contain values of any subclass. We extend this notation to write $\tau(C\!:\!p)$ for the type of a path expression p.

For simplicity we assume that variable names are unique within a class (that is, no two distinct methods or fields in a given class can have the same name). The notation $C\!:\!v$ (*rooted variable*) means that v is a field of class C; this uniquely identifies it. Due to inheritance, the same variable may be described as $D\!:\!v$ with $D <: C$.

[1]If m is overridden we place these fields in the definition of m highest in the class hierarchy.

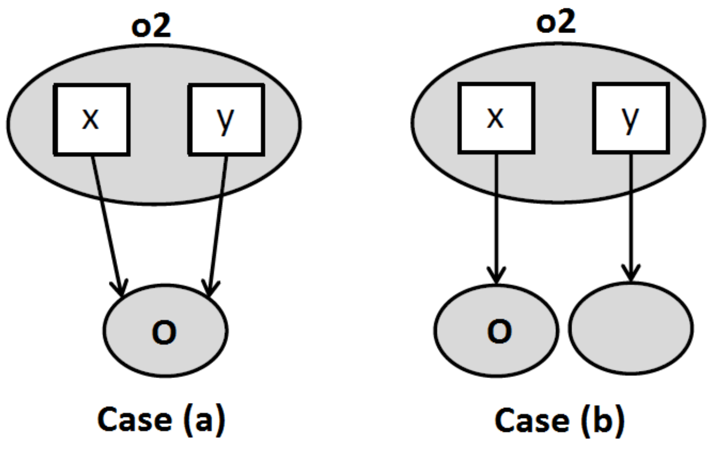


Figure 6.2: Two simple object graphs.

Access paths p, q, r and s are (as usual) sequences $v_1.v_2.\cdots.v_n$ which follow a chain of field selections. A *rooted path* $C\!:\!p$ is a correctly typed access path; it is a pair $C_1\!:\!v_1.\cdots.v_n$ such that either $n = 1$ and $C_1\!:\!v_1$ is a rooted variable, or $n > 1$ and $C_1\!:\!v_1$ is a rooted variable and $C_2\!:\!v_2.\cdots.v_n$ is a rooted path where $C_2 = \tau(C_1\!:\!v_1)$. We call C the class or *context* of the rooted path. The pseudo-variable `this` appearing in class C is represented as the empty path $C\!:\!\epsilon$; this automatically identifies `this`.f with f.

For practical analysis purposes Section 6.2.4 later extends the idea of a rooted path to allow it to represent an infinite number of paths with some common repeated sequence of variables, but we start with a simple, though possibly infinitary, mathematical definition of aliasing.

Given a program whose aliasing contracts we wish to verify statically, we first use its statements *stmt* to create an *aliasing graph*, and then use this to attempt to verify the program's contracts. We start by constructing the aliasing graph.

6.2.2 Static rooted paths and dynamic objects

Rooted paths are static constructs which do not exist at run time. Although we are developing a static analysis here, we are ultimately interested in the relationships between run-time objects, not static rooted paths.

The correspondence between rooted paths and run-time objects is as follows: in one particular state of execution (which can be reached from the initial program state), rooted path $C\!:\!p$ corresponds to objects which are reachable by starting at any object of type C and performing a chain of field selections p.

In order to verify a program's contracts, we must determine what aliasing between objects is possible at run time. Figure 6.2 shows two simple object graphs; objects are represented by ellipses, while variables are represented by squares. The important difference between the two graphs is that when object o is accessed in *Case (a)*, this requires evaluation of the contracts of variables x and y; in *Case (b)*, on the other hand, an access to o requires evaluation of the contract of x only.

To ensure safety, our analysis must derive an aliasing relationship between two rooted paths $C\!:\!x$ and $C\!:\!y$ if there are any possible execution states that have an object graph as in *Case (a)*; that is, where there may exist an object o_2, such that $o_2.x$ and $o_2.y$ point to the same object o. If this is the case, we say that rooted paths $C{:}x$ and $C{:}y$ *may-alias*.

Our analysis is conservative and derives an *over-approximation* of may-aliasing: it assumes aliasing (that is, *Case (a)*) is possible unless it can prove otherwise; even if there is no possible object graph that satisfies *Case (a)*, our analysis may still derive may-aliasing between $C{:}x$ and $C{:}y$.

6.2.3 Aliasing graph

An *aliasing graph* has rooted paths for vertices, and its edges are first seeded by the program statements, before a closure operation is performed. Performing *alias analysis* on the program results in an aliasing graph. We use an Andersen-style analysis [5] which is context-insensitive, flow-insensitive and subset-based (and thus takes into account the direction of value flow in assignments, rather than an equality-based analysis which would consider assignments as bi-directional).

The aliasing graph has a directed *flow edge* from $C{:}p.v_1$ to $C{:}q.v_2$ (written $C{:}p \rightarrow C{:}q$), if there is a run-time possibility that data in an object o of type C may flow from $o.p.v_1$ to $o.q.v_2$. For example the assignment $x = y.f$ appearing in a method of class C results in an edge $C{:}y.f \rightarrow C{:}x$. As usual our notion of 'may flow' is conservative – there may be no feasible execution which witnesses this possibility. Flow is a preorder relation which is reflexive and transitive but not necessarily symmetric (although in some instances it may be).

The aliasing graph also has undirected *aliasing edges*; we write $C{:}p \leftrightarrow C{:}q$ to denote *may-aliasing* between rooted paths $C{:}p$ and $C{:}q$, where there may at run time exist an object o of class C such that $o.p$ and $o.q$ point to the same object (that is, contain the same value). Aliasing edges are defined in terms of flow edges as shown below. We note that although flow implies may-aliasing, the converse does not hold. Thus, aliasing edges are a superset of flow edges in an aliasing graph.

$$\frac{C{:}p \rightarrow C{:}q}{C{:}p \leftrightarrow C{:}q} \qquad (\leftrightarrow\ \mathit{1})$$

$$\frac{C{:}q \rightarrow C{:}p}{C{:}p \leftrightarrow C{:}q} \qquad (\leftrightarrow\ \mathit{2})$$

$$\frac{C{:}r \rightarrow C{:}p \quad C{:}r \rightarrow C{:}q}{C{:}p \leftrightarrow C{:}q} \qquad (\leftrightarrow\ \mathit{3})$$

Two rooted paths $C{:}p$ and $C{:}q$ may-alias if there is flow of data from $C{:}p$ to $C{:}q$ ($C{:}p \rightarrow C{:}q$) or vice versa ($C{:}q \rightarrow C{:}p$). Alternatively, if there is a third rooted path $C{:}r$, such that the value of $C{:}r$ flows into both $C{:}p$ and $C{:}q$ ($C{:}r \rightarrow C{:}p$ and $C{:}r \rightarrow C{:}q$), then may-aliasing is also possible between $C{:}p$ and $C{:}q$.

The above rules show that aliasing edges are symmetric but not necessarily transitive; they are additionally reflexive[2]. Although the lack of transitivity is perhaps surprising, it is caused by the distinction between flow edges and aliasing edges which allows the analysis to discount impossible may-aliasing cases. Consider two assignment statements $y = x$ and $y = z$ in class C. We have $C{:}x \rightarrow C{:}y$ and $C{:}z \rightarrow C{:}y$ and thus $C{:}x \leftrightarrow C{:}y$

[2]Given the definition of flow edges below (which make flow edges reflexive).

and $C\!:\!z \leftrightarrow C\!:\!y$ (by rule ($\leftrightarrow$ *1*) above). However, we do not want to derive $C\!:\!x \leftrightarrow C\!:\!z$; given only these two assignment statements, it is impossible for an object o of type C to exist at run time, where $o.x$ and $o.z$ point to the same object; that is, *Case (a)* in Figure 6.2 above cannot occur and therefore may-aliasing should not be transitive. This illustrates that our analysis is subset-based; an equality-based analysis does not take into account direction of data flow and therefore could not eliminate this case.

Both flow and aliasing edges only connect paths rooted in the same class C, for example $C\!:\!p$ with $C\!:\!q$. We say class C witnesses the flow or aliasing edge.

Given a program Π, the set of flow edges of its aliasing graph is defined by the following axioms and inference rules:

Seeding: For each assignment $p = q$ appearing in class C of program Π:

$$\frac{}{C\!:\!q \rightarrow C\!:\!p} \quad (\textit{Seeding})$$

Calls to `new` and method calls do not generate flow edges because method arguments and results have been reduced to assignments, and `new` is unaliased[3].

Reflexivity: For each rooted path $C\!:\!p$:

$$\frac{}{C\!:\!p \rightarrow C\!:\!p} \quad (\textit{Reflexivity})$$

This rule makes the flow relation reflexive. Given the above rules defining aliasing edges, we also have $C\!:\!p \leftrightarrow C\!:\!p$ for all rooted paths $C\!:\!p$.

We only need to require $C\!:\!\epsilon \rightarrow C\!:\!\epsilon$ (given the next rule, (*Aliased reflexivity*)) but we find that giving the above more general rule is clearer.

Aliased reflexivity If C witnesses p and q being may-aliases then it witnesses flow from $p.r$ to $q.r$. This encodes as the following rule:

$$\frac{C\!:\!p \leftrightarrow C\!:\!q}{C\!:\!p.r \rightarrow C\!:\!q.r} \quad (\textit{Aliased reflexivity})$$

If $C\!:\!p \leftrightarrow C\!:\!q$, this means that there may at run time exist an object o such that $o.p$ and $o.q$ point to the same object. If this is indeed the case, then $o.p.r$ and $o.q.r$ must necessarily point to the same object as well. As for the reflexivity rule above, this causes flow and gives us $C\!:\!p.r \rightarrow C\!:\!q.r$.

As the $\leftrightarrow$ relation is symmetric, the above rule will derive both $C\!:\!p.r \rightarrow C\!:\!q.r$ and $C\!:\!q.r \rightarrow C\!:\!p.r$. Note that this is a stronger relationship than $C\!:\!p.r \leftrightarrow C\!:\!q.r$ which does not necessarily imply flow between $C\!:\!p.r$ and $C\!:\!q.r$.

Context transfer and transitivity: This rule describes how two flow edges are combined to deduce a transitive edge. Such an edge may have a different context from the original edges, thus transferring may-aliasing information from one context to another.

Suppose class D writes to variable v (perhaps a field in class F) and class E reads from it. Suppose also class C witnesses a may-alias between D's path to v and E's

[3]That is, a run-time object is unaliased directly following its creation.

path to v (for example based on C having paths s to D and s' to E). Then there is possible flow from the variable written in D to that read in E and this possible flow is witnessed by C. This encodes as the following rule:

$$\frac{\begin{array}{c} C\!:\!s.q \leftrightarrow C\!:\!s'.q' \\ D\!:\!p \rightarrow D\!:\!q.v \quad E\!:\!q'.v \rightarrow E\!:\!r \end{array}}{C\!:\!s.p \rightarrow C\!:\!s'.r} \quad (\textit{Context transfer and transitivity})$$

The above definitions export the concept of aliasing edges. As we explained in Section 6.2.2, may-aliasing between rooted paths is important for contract verification, where we need to verify the contracts of all variables that may-alias. Contract verification is explained in detail in Section 6.2.5.

Flow edges are important for inferring aliasing edges but are not separately required for contract verification. The reason for this is that the flow edges of an aliasing graph are a *subset* of the aliasing edges; thus they do not need to be considered separately.

The above rules define a system similar to that of Andersen or Steensgaard for calculating the points-to relation; it is a form of dynamic (or online) transitive closure algorithm. However, the resulting graph may have an infinite number of vertices and edges because rooted paths are not simply restricted to those appearing in the source code[4]. We return to the problem of infinite aliasing graphs in Section 6.2.4.

6.2.3.1 Examples

We now consider several simple examples to show how the above rules work. First, suppose a class C has the statements $x = y$ and $y = z$ (flow insensitivity means the order of the statements does not matter). We seed the aliasing graph with edges $C\!:\!y \rightarrow C\!:\!x$ and $C\!:\!z \rightarrow C\!:\!y$ according to (*Seeding*) above. We then expect our analysis to deduce the obvious transitive aliasing relationship $C\!:\!z \rightarrow C\!:\!x$. This relationship is deduced by (*Context transfer and transitivity*): given $C\!:\!y \rightarrow C\!:\!x$ and $C\!:\!z \rightarrow C\!:\!y$ and $C\!:\!\epsilon \rightarrow C\!:\!\epsilon$, it infers $C\!:\!z \rightarrow C\!:\!x$ as expected (setting $p = z$, $q.v = q'.v = y$, $r = x$, $s = s' = \epsilon$).

This also works if the two assignment statements occur in different classes. For example, suppose a class D has statement $a.x = b.y$, class E has statement $d.z = c.y$ and class C has statement $e.b = f.c$, then we start by seeding the aliasing graph with edges $D\!:\!b.y \rightarrow D\!:\!a.x$, $E\!:\!d.z \rightarrow E\!:\!c.y$ and $C\!:\!f.c \rightarrow C\!:\!e.d$. Applying (*Context transfer and transitivity*) with $p = d.z$, $q = c$, $q' = b$, $r = a.x$, $v = y$, $s = f$ and $s' = e$ we get $C\!:\!f.d.z \rightarrow C\!:\!e.a.x$ as expected. This demonstrates context transfer: the new edge has a different context from the two edges that we used to deduce it.

Finally, we give an example to demonstrate the application of (*Aliased reflexivity*). Suppose class C has statements $y = x.a$ and $z = y.b$. These give edges $C\!:\!x.a \rightarrow C\!:\!y$ and $C\!:\!y.b \rightarrow C\!:\!z$. Putting these two assignments together, we expect our analysis to deduce an aliasing relationship between $x.a.b$ and z. From the initial edges, (*Aliased reflexivity*) deduces $C\!:\!x.a.b \rightarrow C\!:\!y.b$. This in turn enables the application of (*Context transfer and transitivity*) (with $p = x.a.b$, $q.v = q'.v = y.b$, $r = z$, $s = s' = \epsilon$), generating edge $C\!:\!x.a.b \rightarrow C\!:\!z$ as expected.

[4]Rule (*Context transfer and transitivity*) introduces rooted paths which may not exist in the original source code.

6.2.3.2 Soundness

In this section, we demonstrate the soundness of the above rules by showing that they capture all aliasing relationships which might occur during run-time execution of a program's source code.

We model run-time behaviour using a *heap* as a state and (because of our flow-insensitive "program is a set of statements" formulation) we assume program statements are repeatedly executed to give successive new heaps.

As usual a heap Ψ is a partial mapping from heap addresses ι to objects. An object is represented as a tuple containing the object's type C and a mapping from the field names f of C to heap addresses. These two definitions combine to $\Psi(\iota) = \langle C, \{f \mapsto \iota\}\rangle$. This is similar to the heap definition used in the operational semantics in Chapter 3.

Below, we refer to objects only by their heap address ι which uniquely identifies them. We use p, q, r and s to represent sequences of field selections. For conciseness, we write $\Psi(\iota, p)$ to represent the lookup of p in heap Ψ starting at the object with address ι.

Given a heap Ψ, we write $\langle\iota_1, f_1\rangle \sim_\Psi \langle\iota_2, f_2\rangle$ if field f_1 of object ι_1 and field f_2 of object ι_2 contain the same value (that is, are aliased) in heap Ψ; thus $\langle\iota_1, f_1\rangle \sim_\Psi \langle\iota_2, f_2\rangle$ if and only if $\Psi(\iota_1, f_1) = \Psi(\iota_2, f_2)$. The difference between $\sim_\Psi$ and $\leftrightarrow$ is that $\sim_\Psi$ is an aliasing relationship in the run-time heap, while $\leftrightarrow$ represents static may-aliasing. We note that $\sim_\Psi$ is an equivalence relation which is reflexive, symmetric and transitive.

Execution consists of non-deterministically choosing an object of type C with address ι in the heap Ψ and non-deterministically executing a statement s of C (or a subtype of C); this produces a new heap Ψ'. We write $\Psi \overset{\iota:s}{\to} \Psi'$. The initial heap has one object of distinguished class `main`. Statements that would cause an exception (for example $x = y.z$ when y is `null`) do not generate a successor state and so are effectively avoided in the non-deterministic selection of statements to execute.

Consider the assignment statement $p.x = q.y$ executed on the object with heap address ι, where $\Psi(\iota, p) = \iota_1$ and $\Psi(\iota, q) = \iota_2$. Then, after executing the assignment, we must have $\langle\iota_1, x\rangle \sim_{\Psi'} \langle\iota_3, z\rangle$ for all $\langle\iota_3, z\rangle$ where $\langle\iota_2, y\rangle \sim_\Psi \langle\iota_3, z\rangle$. Note that this covers the obvious relationship $\langle\iota_1, x\rangle \sim_{\Psi'} \langle\iota_2, y\rangle$ due to the reflexivity of $\sim_\Psi$.

Soundness is now a matter of showing that the aliasing graph's ($\leftrightarrow$) relation satisfies $\sim_\Psi$ (as defined below) for each possible execution heap Ψ having evolved from the initial heap. For example, if the initial heap can evolve to a heap containing an object of class C with two fields f and h which both contain identical (non-`null`) values, then the aliasing graph must contain $C\!:\!f \leftrightarrow C\!:\!h$.

We say that ($\leftrightarrow$) satisfies $\sim_\Psi$ if

$$\langle\iota_1, f_1\rangle \sim_\Psi \langle\iota_2, f_2\rangle \implies C\!:\!p.f_1 \leftrightarrow C\!:\!q.f_2 \; \forall C\!:\!p.f_1, C\!:\!q.f_2$$

Every aliasing relationship in the heap must be represented in the aliasing graph (but not necessarily vice versa).

We now prove soundness by induction[5]: the initial heap state contains no aliasing and therefore gives the base case. The induction step requires us to show that if the aliasing graph satisfies heap Ψ and a single statement is executed, then the aliasing graph also satisfies the resulting heap Ψ'.

We only need to consider execution of assignment statements here, as they are the only statements which change the program's aliasing structure. Execution of any other

[5]Often called *preservation* in proofs of progress and preservation.

statements (such as `new`) may create a heap Ψ' which is different from the previous heap Ψ, but Ψ' must trivially satisfy the aliasing graph, if Ψ did.

In addition, we only need to consider the $\sim_{\Psi'}$ relationships which do not already exist in $\sim_\Psi$; any relationships which are common to $\sim_\Psi$ and $\sim_{\Psi'}$ are trivially satisfied, given that the aliasing graph satisfies Ψ.

Proposition 1. *If* $(\leftrightarrow)$ *satisfies* $\sim_\Psi$ *and* $\Psi \overset{\iota:s.x=s'.y}{\rightarrow} \Psi'$, *then* $(\leftrightarrow)$ *satisfies* $\sim_{\Psi'}$.

Proof. We consider the assignment statement $s.x = s'.y$ in the source code of class D, where $\Psi(\iota, s) = \iota_1$ and $\Psi(\iota, s') = \iota_2$.

We know that we have $\langle \iota_1, y\rangle \sim_{\Psi'} \langle \iota_3, z\rangle$ for each $\langle \iota_3, z\rangle$ such that $\langle \iota_2, y\rangle \sim_\Psi \langle \iota_3, z\rangle$. Then, we need to show that we also have $C{:}p.x \leftrightarrow C{:}r.z$ for all rooted paths $C{:}r.z$ such that $C{:}q.y \leftrightarrow C{:}r.z$.

Note that we only need to consider rooted paths $C{:}p$ and $C{:}q$ such that the run-time objects with address ι_1 and ι_2 are reachable starting from a run-time object o of class C and performing field selections p and q respectively. This means that we need to consider only rooted paths $C{:}p$ and $C{:}q$ which may-alias rooted paths $C{:}p'.s$ and $C{:}q'.s'$ ending in s and s' respectively. Below we can therefore assume that we have $C{:}p \leftrightarrow C{:}p'.s$ and $C{:}q \leftrightarrow C{:}q'.s'$. Then:

1. Given the assignment statement $s.x = s'.y$ in class D we get $D{:}s'.y \rightarrow D{:}s.x$ by (*Seeding*).

2. We get $D{:}s.x \rightarrow D{:}s.x$ by (*Reflexivity*).

3. Given $D{:}s'.y \rightarrow D{:}s.x$ and $D{:}s.x \rightarrow D{:}s.x$ we get $C{:}q'.s'.y \rightarrow C{:}p'.s.x$ for all rooted paths $C{:}p'$ and $C{:}q'$ by (*Context transfer and transitivity*).

4. Given $C{:}p \leftrightarrow C{:}p'.s$ and $C{:}q \leftrightarrow C{:}q'.s'$ we derive $C{:}p'.s.x \rightarrow C{:}p.x$ and $C{:}q.y \rightarrow C{:}q'.s'.y$ by (*Aliased reflexivity*).

5. Given $C{:}q.y \rightarrow C{:}q'.s'.y$, $C{:}q'.s'.y \rightarrow C{:}p'.s.x$ and $C{:}p'.s.x \rightarrow C{:}p.x$ we get $C{:}q.y \rightarrow C{:}p.x$ by two applications of (*Context transfer and transitivity*).

6. Next, we consider all rooted paths $C{:}r.z$ such that $C{:}q.y \leftrightarrow C{:}r.z$. For each such path, we need to show that the aliasing graph contains $C{:}p.x \leftrightarrow C{:}r.z$. There are three distinct cases:

   (a) We may have $C{:}q.y \leftrightarrow C{:}r.z$ because $C{:}q.y \rightarrow C{:}r.z$ according to ($\leftrightarrow$ *1*). Then:

      i. Given $C{:}q.y \rightarrow C{:}r.z$ and $C{:}q.y \rightarrow C{:}p.x$, we get $C{:}p.x \leftrightarrow C{:}r.z$ as required by ($\leftrightarrow$ *3*).

   (b) We may have $C{:}q.y \leftrightarrow C{:}r.z$ because $C{:}r.z \rightarrow C{:}q.y$ by ($\leftrightarrow$ *2*). Then:

      i. Given $C{:}r.z \rightarrow C{:}q.y$ and $C{:}q.y \rightarrow C{:}p.x$ we derive $C{:}r.z \rightarrow C{:}p.x$ by (*Context transfer and transitivity*).

      ii. Given $C{:}r.z \rightarrow C{:}p.x$ we get $C{:}p.x \leftrightarrow C{:}r.z$ as required by ($\leftrightarrow$ *2*).

   (c) We may have $C{:}q.y \leftrightarrow C{:}r.z$ because there exists a rooted path $C{:}s$ such that $C{:}s \rightarrow C{:}q.y$ and $C{:}s \rightarrow C{:}r.z$ by ($\leftrightarrow$ *3*). Then:

i. Given $C{:}s \to C{:}q.y$ and $C{:}q.y \to C{:}p.x$ we derive $C{:}s \to C{:}p.x$ by (*Context transfer and transitivity*).

ii. Given $C{:}s \to C{:}p.x$ and $C{:}s \to C{:}r.z$ we get $C{:}p.x \leftrightarrow C{:}r.z$ are required by ($\leftrightarrow$ *3*).

Thus, if $(\leftrightarrow)$ satisfies $\sim_\Psi$ and $\Psi \overset{\iota:s.x=s'.y}{\to} \Psi'$, then $(\leftrightarrow)$ satisfies $\sim_{\Psi'}$. This shows that the aliasing rules we presented above correctly infer aliasing relationships in heap Ψ'. $\square$

6.2.4 Effectively computing the aliasing graph

While the axioms and rules given above to specify the aliasing graph are mathematically well defined as the least model of the rules, they are computationally problematic as there can be an infinite number of nodes and edges. For example, consider an extract from the `LinkedList` example used in previous chapters:

```
class LinkedList {
  ...
  public pure boolean member(Object data {true}) {
    for(Node n {true} = head; n != null; n = n.next) {  //A
      ...
    }
  }
}
```

Assignment `n = n.next` at program point `A` seeds the aliasing graph with `Node:n.next` $\to$ `Node:n`. Rule (*Aliased reflexivity*) derives `Node:n.next.next` $\to$ `Node:n.next` and (*Context transfer and transitivity*) combines the two edges to give `Node:n.next.next` $\to$ `Node:n`. Subsequently, the analysis derives `Node:n.next.next.next` $\to$ `Node:n` in the same way and so on forever. This behaviour is somewhat expected as we need to account for all possible run-time depths of exploration of the linked list. However, due to the infinite number of nodes created we cannot solve the problem merely by constructing the graph on demand.

Our implementation sidesteps this problem by representing a Kleene-star-like "repeated traversal of some set of edges" explicitly in a modified aliasing graph. We redefine the notion of vertices of the aliasing graph, so that the rooted paths $C{:}r$ specifically exclude paths of the form $C{:}p.f.q.f$ in which a field name f occurs more than once. When such a vertex is about to be required, we instead replace the (necessarily) existing vertex $C{:}p.f$ with a new form of vertex $C{:}p.f.(q.f)$. We call $C{:}p.f.(q.f)$ a *cyclic access path* and say that $C{:}p.f$, $C{:}p.f.q.f$, $C{:}p.f.q.f.q.f$ and so on are possible *expansions* of $C{:}p.f.(q.f)$. The new vertex $C{:}p.f.(q.f)$ we create is a *summary vertex* representing all vertices which are expansions of $C{:}p.f.(q.f)$.

The introduction of cyclic access paths to the aliasing graph means that the number of vertices in the graph is always finite, and determined by the program size.

Our cyclic access paths are similar to *symbolic access paths* proposed by Deutsch [30]. These also aim to reduce an infinite number of possible access paths to a finite number, but additionally describe how many times a cycle in an access path can occur; for example, the access path `a.b.a.b.c` is shortened to $(a \to b \to)^2 c$. Notation for *compressed access paths* by Komondoor [56] is similar to our work, but the semantics are different: in Komondoor's work, $p.a.*$ represents access path $p.a$ followed by any field selections, for example $p.a.a$ or

$p.a.b$; this is different from our cyclic access paths, where $p.(a)$ may represent only $p.a.a$ but not $p.a.b$.

6.2.5 Contract verification

At run time, an object access requires evaluation of the aliasing contracts of all variables currently pointing to the accessed object. The aliasing graph constructed in Section 6.2.3 overestimates run-time aliasing. We can use this to *statically verify* certain aliasing contracts and therefore eliminate *dynamic checks*. A program which can be verified is then guaranteed to not cause contract violations at run time but the converse does not hold (analogous to static versus dynamic typing).

Verification consists of the following: for a class C, we consider each source-code object access to variable v[6] (that is, field reads $y = v.f$, field writes $v.f = y$ and method calls $v.m()$[7]). For a possible run-time object o of type C, this source-code object access corresponds to a run-time access to $o.v$. Such an access requires the evaluation of the contract of variable v in object o, as well as evaluation of the contract of variable v_2 in object o_2 if $o.v = o_2.v_2$. Therefore, in order to statically verify the above source-code object access, we must verify the contracts for all rooted paths $C{:}p$ such that $o.v$ or $o_2.v_2$ can be reached starting at an object o_3 of type C and performing field selections p; we explain below exactly which rooted paths need to be considered.

We write $[\![C{:}v]\!]$ for the contract in the declaration of variable v visible in class C; the contract of v then appears in class C or one of its superclasses, attached to the variable declaration for v. For conciseness, we similarly write $[\![C{:}p.v]\!]$ for the contract of variable v visible in class $\tau(C{:}p)$. Depending on the type of object access, our analysis verifies the read and/or write contracts as appropriate.

To verify a contract, we must provide access paths representing `accessor`, `accessed` and `this`. We write $\mathit{verify}(\phi, C{:}p_{\text{accessor}}, C{:}p_{\text{accessed}}, C{:}p_{\text{this}})$ for the verification of the contract of variable v of class C, where $\phi = [\![C{:}v]\!]$. We explain the cases of *verify* for different contracts ϕ later in this section.

Due to our ANF-style syntax, there are only two kinds of source-code object accesses to consider:

Access to v, with $v = f$: A run-time object o may access another object o_2 stored in its own field f, for example by reading or writing a field $f.h$ or calling a method $f.m()$.

To verify such an object access, we first find all rooted paths in the aliasing graph which contain f as their last term: $D{:}p.f$, where $\tau(D{:}p) \bowtie C$. We select these rooted paths, because the accessed object o_2 may be reachable at run time from an instance of D following field selections $p.f$. For each such vertex we require $\mathit{verify}([\![D{:}p.f]\!], D{:}p, D{:}p.f, D{:}p)$.

At run time, an access to o_2 also requires the evaluation of the contract of variable h, if there is another object o_3 such that $o_3.h$ is aliased with $o.f$. Thus, we find all rooted paths $D{:}q.h$ such that $D{:}p.f \leftrightarrow D{:}q.h$. For each such $D{:}q.h$ we require $\mathit{verify}([\![D{:}q.h]\!], D{:}p, D{:}q.h, D{:}q)$.

[6]Where v is either a field f or `this`.

[7]No other source-code object accesses are possible due to the restrictions of our ANF-style syntax.

We can equivalently require *verify*$([\![D{:}q.h]\!], D{:}p, D{:}p.f, D{:}q)$, where `accessed` is described by $D{:}p.f$ instead of $D{:}q.h$. An access to $D{:}p.f$ requires a contract verification for $D{:}q.h$ because the run-time objects $o.p.f$ and $o.q.h$ may be aliased (*Case (a)* in Figure 6.2); contract verification would not be required if the two run-time objects were unrelated (*Case (b)* in Figure 6.2). Therefore, $o.p.f$ and $o.q.h$ must be aliased when the contract of $o.q.h$ is evaluated. It follows that we can treat $D{:}p.f$ and $D{:}q.h$ as equivalent during contract verification (or otherwise verification would not be needed); it is then sufficient for just *one* of these two alternative verifications to succeed in order to prove that the contract will evaluate to `true` at run time.

We now use a simple example program to show how object accesses with $v = f$ are verified:

```
class C {
  C f {φ};
  C g {φ′};
  m() {
    f = g;            //A
    f.m();            //B
  }
}
```

The statement `f.m()` at program point `B` represents an access from an instance of type C (or a subtype of C) to the object stored in its field f. Thus, we require *verify*$(\phi, C{:}\epsilon, C{:}f, C{:}\epsilon)$. In addition, the assignment at program point `A` above creates aliasing between fields f and g in class C; we have $C{:}f \leftrightarrow C{:}g$. We therefore also require *verify*$(\phi', C{:}\epsilon, C{:}g, C{:}\epsilon)$ or equivalently *verify*$(\phi', C{:}\epsilon, C{:}f, C{:}\epsilon)$.

Access to v, with $v =$ `this`: An object of type C may access itself, for example by reading or writing its own field `this`.h or calling its own method `this`.m. At run time, the accessed object may be any object of type C or a subtype of C. Therefore, for each rooted path $D{:}p.f$ where $\tau(D{:}p.f) \bowtie C$ we require *verify*$([\![D{:}p.f]\!], D{:}p.f, D{:}p.f, D{:}p)$.

We do not need to separately verify the contracts of aliased rooted paths $D{:}q.h$ with $D{:}p.f \leftrightarrow D{:}q.h$ as the previous case required. $D{:}p.f \leftrightarrow D{:}q.h$ is possible only if $\tau(D{:}p.f) \bowtie \tau(D{:}q.h)$; thus, the contracts of $D{:}q.h$ will already be verified by the previous paragraph.

As an example of how we verify object accesses with $v =$ `this`, consider the following program:

```
class C {
  D f {φ};
}
class D {
  m() {
    m();              //A
  }
}
```

The statement `m()` at program point `A` represents an access from an instance of type D to itself. To verify the correctness of the access, we require *verify*$(\phi, C{:}f, C{:}f, C{:}\epsilon)$, since $\tau(C{:}f) \bowtie D$.

These two cases show that a single source-code object access may require several contract verifications, analogous to the conjunctive evaluation of several contracts at run time. In some cases, the steps described above may cause the same contract ϕ to be verified multiple times for the same values of $C{:}p_{\text{accessor}}$, $C{:}p_{\text{accessed}}$ and $C{:}p_{\text{this}}$. For efficiency reasons, implementations may choose to avoid such repeated verifications by caching.

We now define when *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}p_{\text{accessed}}, C{:}p_{\text{this}})$ holds for different syntactic contracts ϕ. We only consider syntactic contracts ϕ which are required to model existing alias protection schemes such as Clarke-style ownership types and module encapsulation. We first explain verification of these contracts with simple rooted paths representing $C{:}p_{\text{accessor}}$, $C{:}p_{\text{accessed}}$ and $C{:}p_{\text{this}}$; verification with cyclic access paths (introduced in Section 6.2.4 above) is covered in Section 6.2.5.1 below.

Case ϕ of `true` always holds.

Case ϕ of `accessor == this` holds if, for every run-time object o of type C, $o.p_{\text{accessor}}$ and $o.p_{\text{this}}$ must be aliased. We write $C{:}p_{\text{accessor}} \equiv C{:}p_{\text{this}}$ to represent this and call it *must-equality*.

Our aliasing graph contains only may-aliasing information, which cannot be used to deduce must-equality. For example, consider two paths $C{:}p$ and $C{:}q$; if the aliasing graph contains the relationship $C{:}p \leftrightarrow C{:}q$, we know only that, for a run-time object o of type C, $o.p$ and $o.q$ may be aliased (or they may not be).

The only safe way to guarantee must-equality of two rooted paths is when we the two paths are syntactically equal: $C{:}\epsilon \equiv C{:}\epsilon$ and $C{:}p_1.v \equiv C{:}p_2.v$ if and only if $C{:}p_1 \equiv C{:}p_2$. For every run-time object o of type C, $o.p$ and $o.p$ must clearly be aliased.

Case ϕ of `accessor == accessed` holds if $C{:}p_{\text{accessor}} \equiv C{:}p_{\text{accessed}}$.

Case ϕ of `accessor instanceof D` holds if $\tau(C{:}p_{\text{accessor}}) <: D$.

Case ϕ of `accessor in` g holds if, for every run-time object o of class C, $o.p_{\text{accessor}}$ is in group g of $o.p_{\text{this}}$.

We define the compile-time function *path* to return a set of all rooted paths in group g in $C{:}p$, where the definition of g contains fields $f_1 \dots f_n$ and nested encapsulation groups $g_1 \dots g_m$ of fields $h_1 \dots h_m$ $(h_1.g_1 \dots h_m.g_m)$[8].

$$\mathit{path}(g, C{:}p) = \bigcup_{i=1}^{n} \{C{:}p.f_i\} \cup \bigcup_{j=1}^{m} \mathit{path}(g_j, C{:}p.h_j)$$

Then, using this definition, *verify*(`accessor in` $g, C{:}p_{\text{accessor}}, C{:}p_{\text{accessed}}, C{:}p_{\text{this}}$) holds if $C{:}p_{\text{accessor}} \in \mathit{path}(g, C{:}p_{\text{this}})$.

[8]We can use cyclic access paths to ensure that the number of rooted paths returned by *path* is finite.

For example, $C{:}p.v_1$ is in group g of $C{:}p$ if the definition of group g in type $\tau(C{:}p)$ contains field v_1. $C{:}p.v_1.v_2$ is in group g of $C{:}p$ if g in type $\tau(C{:}p)$ contains a nested group $v_1.g_2$ and group g_2 in type $\tau(C{:}p.v_1)$ contains field v_2.

We note here that it is not possible to guarantee that $C{:}p.q$ is in group g of $C{:}r$, unless we have $C{:}p \equiv C{:}r$; as we noted above, our analysis deduces may-aliasing relationships only and we can therefore never be certain that, for every run-time object o of class C, $o.p$ and $o.r$ are definitely aliased unless we have $C{:}p \equiv C{:}r$.

Cases ϕ of `accessor canread this` and `accessor canwrite this`: This contract requires *indirect* contract verifications: the verification $\mathit{verify}(\phi, C{:}p.f, C{:}q.g.h, C{:}q.g)$ instead requires us to verify the contract $\phi' = [\![C{:}q.g]\!]$ of the original contract's declaring object $C{:}q.g$; thus, we require $\mathit{verify}(\phi', C{:}p.f, C{:}q.g, C{:}q)$. This is analogous to run-time contract evaluation; to evaluate the contract `accessor canread this` of variable g of run-time object $o.q$, we instead evaluate the contracts of $o.q$, as explained in Chapter 3.

We verify the contract of the declaring object of ϕ by removing the last term from $C{:}p_{\text{accessed}}$ and $C{:}p_{\text{this}}$, while keeping $C{:}p_{\text{accessor}}$ the same.

It is possible for $C{:}p_{\text{this}}$ to become empty if we continue to remove its last term; at this point, we cannot keep removing terms. For $\mathit{verify}(\phi, C{:}p.f, C{:}q, C{:}\epsilon)$, we instead require $\mathit{verify}(\phi', D{:}p'.f'.p.f, D{:}p'.f', D{:}p')$ where $\phi' = [\![D{:}p'.f']\!]$ for all rooted paths $D{:}p'.f'$ such that $\tau(D{:}p'.f') \bowtie C$. This is similar to verifying an object access to v where $v = \mathit{this}$: all rooted paths $D{:}p'.f'$ with matching type must be considered. Consider the following example program, based on a `LinkedList` data structure:

```
class Node {
  Node next {accessor canread this, accessor canwrite this};
  pure m() {
    next.next.m();    //A
  }
}
```

The object access at program point `A` requires $\mathit{verify}(\texttt{accessor can}-\texttt{read this}, \texttt{Node}{:}\epsilon, \texttt{Node}{:}\texttt{next.next}, \texttt{Node}{:}\texttt{next})$; as described above, for this to hold, we instead require $\mathit{verify}(\texttt{accessor canread this}, \texttt{Node}{:}\epsilon, \texttt{Node}{:}\texttt{next}, \texttt{Node}{:}\epsilon)$, where we simply remove the last term from $C{:}p_{\text{accessed}}$ and $C{:}p_{\text{this}}$, while keeping $C{:}p_{\text{accessor}}$ the same.

Now, because $C{:}p_{\text{this}}$ is empty (and we cannot remove its last term), we find all rooted paths $C{:}p$ such that $\tau(C{:}p) \bowtie \texttt{Node}$. For example, we set $C{:}p = \texttt{Node}{:}\texttt{next}$ and require $\mathit{verify}(\texttt{accessor canread this}, \texttt{Node}{:}\texttt{next}, \texttt{Node}{:}\texttt{next}, \texttt{Node}{:}\epsilon)$.

Verification of the contracts `accessor canread this` and `accessor canwrite this` can cause repeated contract verifications; that is, multiple verifications of the same contract ϕ for the same values of $C{:}p_{\text{accessor}}$, $C{:}p_{\text{accessed}}$ and $C{:}p_{\text{this}}$. As already noted above, such recursively repeated verifications can (and should) be avoided by caching in implementations of our analysis, thus ensuring that verification of `accessor canread this` and `accessor canwrite this` results in a finite number of contract verifications.

In the example above, $\textit{verify}(\texttt{accessor canread this}, \texttt{Node:next}, \texttt{Node:next}, \texttt{Node:}\epsilon)$ in turn requires $\textit{verify}(\texttt{accessor canread this}, \texttt{Node:next.next}, \texttt{Node:next}, \texttt{Node:}\epsilon)$ (if we again set $C{:}p = \texttt{Node:next}$) which requires $\textit{verify}(\texttt{accessor canread this}$, $\texttt{Node:next.next.next}, \texttt{Node:next}, \texttt{Node:}\epsilon)$ and so on. By using cyclic access paths and writing $\textit{verify}(\texttt{accessor canread this}, \texttt{Node:(next)}, \texttt{Node:next}, \texttt{Node:}\epsilon)$ we can avoid infinitely repeating contract verifications.

Other cases of ϕ do not hold for the purposes of this analysis; they do not arise when modelling existing alias protection systems such as Clarke-style ownership types.

6.2.5.1 Contract verification with cyclic access paths

Section 6.2.4 introduced cyclic access paths $C{:}p.(q).f$ ensure that the aliasing graph remains finite; each cyclic access path summarises an infinite number of vertices. In this section, we look at how contract verification works in the presence of cyclic access paths: we define when $\textit{verify}(\phi, C{:}p_{\text{accessor}}, C{:}p_{\text{accessed}}, C{:}p_{\text{this}})$ holds, where $C{:}p_{\text{accessor}}$, $C{:}p_{\text{accessed}}$ and $C{:}p_{\text{this}}$ may be cyclic access paths:

Case ϕ of `true` always holds.

Case ϕ of `accessor == this` holds if $C{:}p_{\text{accessor}} \equiv C{:}p_{\text{this}}$ and neither $C{:}p_{\text{accessor}}$ nor $C{:}p_{\text{this}}$ are cyclic. Since cyclic access paths represent many actual access path expansions, equality of such paths can never be proven; we cannot know for certain which expansions of the cyclic access paths we are comparing for equality. For example, we must reject $C{:}a.(b.c) \equiv C{:}a.(b.c)$ as this may in fact be a comparison between two different expansions of $C{:}a.(b.c)$ (for example $C{:}a.b.c \equiv C{:}a.b.c.b.c$ which is clearly false).

Case ϕ of `accessor == accessed` holds if $C{:}p_{\text{accessor}} \equiv C{:}p_{\text{accessed}}$ and $C{:}p_{\text{accessor}}$ and $C{:}p_{\text{accessed}}$ are not cyclic.

Case ϕ of `accessor instanceof D` holds for cyclic access path $C{:}p_{\text{accessor}} = C{:}p.(q.f)$, if $\tau(C{:}p) <: D$ and $\tau(C{:}p.q.f) <: D$. Testing the first two expansions of $C{:}p.(q.f)$ is sufficient, since all later expansions ($C{:}p.q.f.q.f$, $C{:}p.q.f.q.f.q.f$ and so on) must have the same declared type as $C{:}p.q.f$, as they all end with the same variable f; their declared type is the type of field f in $\tau(C{:}p.(q.f).q)$.

We must check the types of the first two expansions, $C{:}p$ and $C{:}p.q.f$, separately as their declared types $\tau(C{:}p)$ and $\tau(C{:}p.q.f)$ may differ (as long as they share a common supertype C' such that $C'{:}q.f$ is a valid rooted path). Therefore, $\tau(C{:}p) <: D$ and $\tau(C{:}p.q.f) <: D$ may give different results and both conditions need to be verified to show that the contract holds for all possible expansions of $C{:}p_{\text{accessor}}$.

Case ϕ of `accessor in` g : First, we consider $\textit{verify}(\texttt{accessor in}\ g, C{:}p.(q).r, C{:}p_{\text{accessed}}, C{:}p_{\text{this}})$ where $C{:}p_{\text{accessor}}$ is cyclic; this holds if there is another group g_2 in $D = \tau(C{:}p)$ such that:

- $C{:}p.g_2 \in \textit{path}(g, C{:}p_{\text{this}})$; and
- $C{:}p.q.g_2 \in \textit{path}(g, C{:}p_{\text{this}})$; and
- $C{:}p.r \in \textit{path}(g_2, C{:}p)$.

These conditions simply check for the existence of another encapsulation group g_2 which is reachable from group g in $C{:}p_{\text{this}}$ by zero or more expansions of $C{:}p.(q)$. For this, it is sufficient to check that $C{:}p.g_2$ and $C{:}p.q.g_2$ are reachable by following paths in group g starting at $C{:}p_{\text{this}}$; the remaining expansions of the cycle (for example $C{:}p.q.q.g_2$) must then necessarily also be reachable from g starting at $C{:}p_{\text{this}}$. Secondly, we require that g_2 in $C{:}p$ contains $C{:}p.r$; this ensures that the remainder of $C{:}p_{\text{accessor}}$ after the cycle is also reachable. These three conditions together ensure that all expansions of $C{:}p.(q).r$ must be reachable by following paths in g starting at $C{:}p_{\text{this}}$: any number of expansions of $C{:}p.(q)$ take us from group g to group g_2 and g_2 contains the rest of the rooted path we are looking for.

For example, we can use encapsulation groups to model aliasing in a `LinkedList` (as we did in Chapter 3):

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

To check *verify*(`accessor in allNodes`, `LinkedList`:`head`.(`next`), `LinkedList`:`head`, `LinkedList`:ϵ), we first set $g_2 =$ `nextNodes`. Then, as required above, we have:

- `LinkedList`:`head.nextNodes` $\in$ *path*(`allNodes`, `LinkedList`:ϵ); and
- `LinkedList`:`head.next.nextNodes` $\in$ *path*(`allNodes`, `LinkedList`:ϵ); and
- `LinkedList`:`head` $\in$ *path*(`nextNodes`, `LinkedList`:`head`).

As expected, we conclude that the contract holds, having shown that all possible expansions of `LinkedList`:`head`.(`next`) must be in group `allNodes` of `LinkedList`:ϵ.

When $C{:}p_{\text{this}}$ is cyclic, contract verification must fail. We noted above that it is not possible for $C{:}p.q$ to be in group g of $C{:}p_{\text{this}}$, unless we have $C{:}p \equiv C{:}p_{\text{this}}$. As discussed above, must-equality cannot be shown to hold for cyclic paths; thus, if $C{:}p_{\text{this}}$ is cyclic, we can never have $C{:}p \equiv C{:}p_{\text{this}}$, meaning that the contract is not verifiable.

Cases ϕ of `accessor canread this` and `accessor canwrite this`: Contract verification of these contracts can be extended to cyclic access paths in a straightforward way. If $C{:}p_{\text{accessor}}$ is cyclic, this has no effect here, since contract verification is simply passed on to the contract's declaring object, while the value of $C{:}p_{\text{accessor}}$ stays the same; this contract can then be verified as explained in this section.

If we have $C{:}p_{\text{this}} = C{:}p.(q).f$, this requires verification of the contracts of $C{:}p.(q)$ instead. We need this verification to cover all possible expansions of $C{:}p.(q)$ (including zero expansions of the cycle). Therefore, for *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r.h, C{:}p.(q).f)$ we require *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r, C{:}p)$ and *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r, C{:}p.(q).q)$.

We note that if all variables in q of $C{:}p.(q)$ have the contract `accessor canread this` or `accessor canwrite this`, *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r.g, C{:}p.(q))$ simply reduces to *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r, C{:}p)$. This happens because *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r.g, C{:}p.(q))$ reduces to *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r, C{:}p)$ and *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r, C{:}p.(q).q)$ as described above. Then, *verify*$(\phi, C{:}p_{\text{accessor}}, C{:}r, C{:}p.(q).q)$ will eventually cause repeated verification of $\mathit{verify}(\phi, C\!:\!p_{\text{accessor}}, C\!:\!r.g, C\!:\!p.(q))$[9] which implementations should omit by caching.

Other cases of ϕ do not hold for the purposes of this analysis.

6.2.6 Equivalence edges

Unfortunately, the analysis we have presented so far is over-pessimistic and, as a result, fails to verify many simple programs (including some programs modelling Clarke-style ownership types with aliasing contracts). This over-pessimism pertains to relationships between two rooted paths $C\!:\!p.v$ and $C\!:\!q.v$ (ending in the same variable v) introduced by the (*Aliased reflexivity*) rule presented above. We give an example of a very simple program which fails to verify below.

In this section, we refine the relationships in our aliasing graph and introduce *equivalence edges*, denoted $\cong$, to make our analysis less conservative, allowing it to verify more contracts (including those modelling Clarke-style ownership types as discussed in Section 6.3.1).

Consider the following simple example; the contracts in this example appear straightforward to verify, but verification fails as we explain below:

```
class C {
  D x {true};
  D y {true};
  ...
  x = y;        //A
}
class D {
  E z {accessor == this};
  ...
  z.m();        //B
}
```

Our analysis uses the assignment at program point `A` to derive $C\!:\!x \leftrightarrow C\!:\!y$ and then applies (*Aliased reflexivity*) to deduce $C\!:\!x.z \to C\!:\!y.z$. This flow edge is a spurious over-approximation; it occurs despite the fact that variable z is never actually assigned to in the above code.

Now consider the object access `z.m()` at program point `B`. This object access is legal and this should be easy to show statically.

To verify this object access, our analysis attempts to show that $\mathit{verify}(\texttt{accessor} == \texttt{this}, C\!:\!x, C\!:\!x.z, C\!:\!x)$ holds; this is straightforward, since the rooted paths representing `accessor` and `this` are identical here (both are $C\!:\!x$).

Next, the analysis considers all rooted paths which are aliased with $C\!:\!x.z$. Since we have $C\!:\!x.z \leftrightarrow C\!:\!y.z$, we also require $\mathit{verify}(\texttt{accessor} == \texttt{this}, C\!:\!x, C\!:\!x.z, C\!:\!y)$ to hold[10]. This verification clearly does not hold, since the rooted paths representing `accessor` and `this` are not identical here ($C\!:\!x$ and $C\!:\!y$ respectively). Thus, verification of this simple object access fails.

[9]Both $C\!:\!p_{\text{this}}$ and $C\!:\!p_{\text{accessed}}$ eventually start repeating in this way, while $C\!:\!p_{\text{accessor}}$ stays the same.

[10]Or alternatively, $\mathit{verify}(\texttt{accessor} == \texttt{this}, C\!:\!x, C\!:\!y.z, C\!:\!y)$ with an equivalent value for `accessed` as explained in Section 6.2.5.

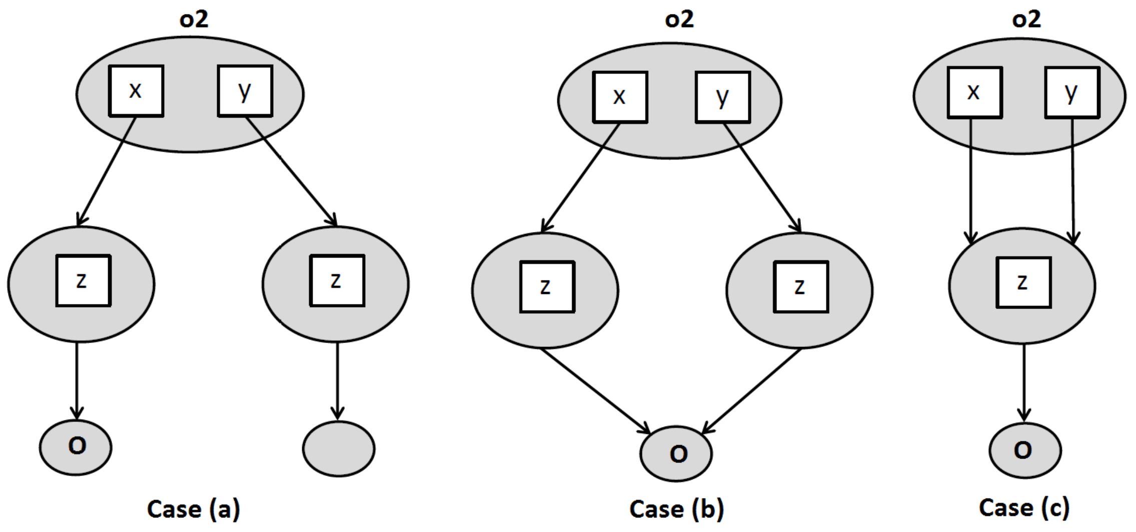


Figure 6.3: Possible run-time object graphs for $C{:}x.z \leftrightarrow C{:}y.z$

The problem is that the assumptions made by our analysis are over-pessimistic. May-aliasing between $C{:}x.z$ and $C{:}y.z$ ($C{:}x.z \leftrightarrow C{:}y.z$) correctly reflects the relationships in three possible run-time object graphs. These are shown in Figure 6.3.

However, given the above example code (without any additional uses of x, y and z in "...") *Case (b)* in Figure 6.3 is impossible. This object graph can be created by the assignment `x.z = y.z` but not by `x = y` only. This case is the cornerstone of the analysis refinement presented here.

The implications for contract verification are important. When object o is accessed, *Case (a)* requires evaluation of the contract of $o_2.x.z$ only; *Case (b)* requires evaluation of the contracts of $o_2.x.z$ *and* $o_2.y.z$; *Case (c)* requires evaluation of the contract of either $o_2.x.z$ or $o_2.y.z$. In *Case (c)*, evaluating the contracts of both $o_2.x.z$ and $o_2.y.z$ is unnecessary, since they are an evaluation of the same contract of the same object.

In the example above the second contract verification $\mathit{verify}(\texttt{accessor} == \texttt{this}, C{:}x, C{:}y.z, C{:}y)$ (which fails) is thus completely unnecessary; there is no possible run-time object graph that requires it (because *Case (b)* is unnecessary). By skipping this unnecessary verification, the above example becomes easy to verify.

The relationship between $C{:}x.z$ and $C{:}y.z$ includes fewer possible object graphs than simple may-aliasing. It is "all-or-nothing" aliasing; if we take an object o of class C, then $o.x.z$ and $o.y.z$ are aliased if $o.x$ and $o.y$ are aliased; $o.x.z$ and $o.y.z$ are unrelated if $o.x$ and $o.y$ are unrelated; it is not possible for $o.x.z$ and $o.y.z$ to be aliased if $o.x$ and $o.y$ are not.

We call this relationship *may-equivalence* since, for the purposes of contract verification, $C{:}x.z$ and $C{:}y.z$ can be considered to be equivalent. May-equivalence can produce object graphs as in *Case (a)* and *Case (c)* in Figure 6.3 only, while may-aliasing can produce all three object graphs. We note that may-equivalence is only possible between two rooted paths which finish with the same variable v; otherwise, *Case (b)* above remains possible.

We introduce equivalence edges, denoted $\cong$, to represent may-equivalence in the aliasing graph. Equivalence edges, as defined by the rules below, give an equivalence relation which is reflexive, symmetric and transitive.

We re-state the inference rules given in Section 6.2.3 above to include equivalence edges:

Aliasing edges: The rules describing aliasing edges are unchanged from the previous definition:

$$\frac{C\!:\!p \rightarrow C\!:\!q}{C\!:\!p \leftrightarrow C\!:\!q} \quad (\leftrightarrow\ \mathit{1})$$

$$\frac{C\!:\!q \rightarrow C\!:\!p}{C\!:\!p \leftrightarrow C\!:\!q} \quad (\leftrightarrow\ \mathit{2})$$

$$\frac{C\!:\!r \rightarrow C\!:\!p \quad C\!:\!r \rightarrow C\!:\!q}{C\!:\!p \leftrightarrow C\!:\!q} \quad (\leftrightarrow\ \mathit{3})$$

Seeding: As before, for each assignment $p = q$ appearing in class C of program Π:

$$\frac{}{C\!:\!q \rightarrow C\!:\!p} \quad (\mathit{Seeding})$$

Reflexivity: The relationship between a rooted path and itself is may-equivalence, rather than may-aliasing as before; clearly, *Case (b)* in Figure 6.3 is impossible in this situation:

$$\frac{}{C\!:\!p \cong C\!:\!p} \quad (\mathit{Reflexivity})$$

This is a change from the previous version of the rule which added a flow edge from each rooted path to itself; clearly, this rule makes the equivalence relation reflexive.

Aliased reflexivity: Similarly, $C\!:\!p \leftrightarrow C\!:\!q$ now implies may-equivalence ($\cong$) between $C\!:\!p.r$ and $C\!:\!q.r$ rather than just flow, as explained in the above example:

$$\frac{C\!:\!p \leftrightarrow C\!:\!q}{C\!:\!p.r \cong C\!:\!q.r} \quad (\mathit{Aliased\ reflexivity})$$

Since aliasing edges are symmetric, this rule results in symmetry of equivalence edges.

Context transfer and transitivity: This rule describes how edges of different types are combined:

$$\frac{\begin{array}{c} C\!:\!s.q \leftrightarrow C\!:\!s'.q' \\ D\!:\!p \overset{0,1}{\rightarrow} D\!:\!q.v \quad E\!:\!q'.v \overset{0,1}{\rightarrow} E\!:\!r \end{array}}{C\!:\!s.p \rightarrow C\!:\!s'.r} \quad (\mathit{Context\ transfer\ and\ transitivity\ 1})$$

(provided $C\!:\!s.p \neq C\!:\!s'.r$, i.e. the paths are syntactically distinct)

Here $\overset{0,1}{\rightarrow}$ represents use of either $\rightarrow$ or $\cong$ edges, but at most one can be $\cong$.

The rule includes the additional restriction that flow edges cannot be introduced between two identical rooted paths (since *Case (b)* in Figure 6.3 is impossible in that situation).

We give a second rule to describe how two equivalence edges are combined; this rule makes the equivalence relation transitive:

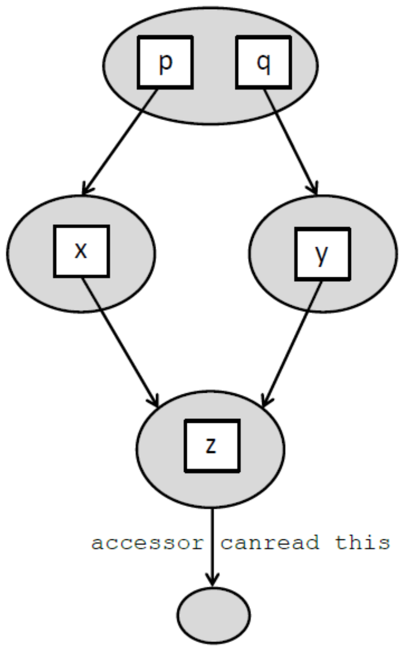


Figure 6.4: One possible run-time object graph for $C{:}p.x.z \cong C{:}q.y.z$

$$\frac{\begin{array}{c} C{:}s.q \leftrightarrow C{:}s'.q' \\ D{:}p \cong D{:}q.v \quad E{:}q'.v \cong E{:}r \end{array}}{C{:}s.p \cong C{:}s'.r} \quad (\textit{Context transfer and transitivity 2})$$

As already mentioned above, we note that equivalence edges as defined by these inference rules can only exist between two rooted paths $C{:}p.v$ and $C{:}q.v$ which share at least the last term v.

We already briefly discussed the impact of may-equivalence on contract verifications in the example above. If $C{:}p.v$ and $C{:}q.v$ are aliased, this would normally require verification of $C{:}q.v$ when the contract of $C{:}p.v$ is verified. However, if we have $C{:}p.v \cong C{:}q.v$ and we have already verified the contract for $C{:}p.v$, we can skip verification of the contract of $C{:}q.v$. More specifically, we can skip $\mathit{verify}([\![C{:}q.v]\!], C{:}p_{\text{accessor}}, C{:}q.v, C{:}q)$ where $\mathit{verify}([\![C{:}p.v]\!], C{:}p_{\text{accessor}}, C{:}p.v, C{:}p)$ holds, if $C{:}p.v \cong C{:}q.v$.

It is of course possible to have both $C{:}p.v \cong C{:}q.v$ and $C{:}p.v \leftrightarrow C{:}q.v$ at the same time; if this is the case, we must consider the may-aliasing relationship, as it includes more possible object graphs than may-equivalence. This then requires us to verify the contract of $C{:}q.v$ when verifying the contract of $C{:}p.v$ (as we did previously in Section 6.2.5); this is necessary to ensure the continued safety of our analysis.

We note that skipping contract verification by exploiting equivalence ($\cong$) edges works for all contract constructs allowed by our syntax in Figure 6.1 and considered in Section 6.2.5, but does *not* apply to contracts involving the `canread` and `canwrite` operators. Contracts involving these operators lead to *indirect* contract verifications; in order to verify the contract "`accessor canread this`" of $C{:}q.v$, for example, our analysis must show that the contract for $C{:}q$ holds.

For example, assume that we have $C{:}p.x.z \cong C{:}q.y.z$, where the contract of variable z is "`accessor canread this`". Then, the object graph in Figure 6.4 is possible at run time. In this situation, it is insufficient to verify only the contract of $C{:}p.x.z$; this verification will indirectly verify the contract of $C{:}p.x$ but not the contract of $C{:}q.y$ (as required to verify the contract of $C{:}q.y.z$). Therefore, in order to maintain correctness for all possible run-time object graphs, our analysis verifies the contract of $C{:}q.y.z$ as usual where $[\![C{:}q.y.z]\!]$ is `accessor canread this` or `accessor canwrite this`, even when $C{:}p.x.z \cong C{:}q.y.z$.

6.3 Static analysis for existing alias protection schemes

In Chapter 4, we showed how aliasing contracts can be used to model existing alias protection schemes. In this section, we prove that the static analysis presented in this chapter can verify the aliasing contracts of programs translated from such existing static alias protection systems, including transitive owners-as-dominators (as implemented by Clarke-style ownership types), transitive owners-as-modifiers and full encapsulation.

6.3.1 Verifying Clarke-style ownership types

In this section, we show how our static analysis can verify programs translated from Clarke-style ownership types to aliasing contracts; this translation was explained in detail in Section 4.10.2.4. Here, we briefly repeat the important details.

We can model ownership types using aliasing contracts by simply replacing Clarke-style ownership type annotations with aliasing contracts as follows:

```
⟦rep⟧    =   {accessor == this || accessor in repGroup}
⟦norep⟧  =   {true}
⟦owner⟧  =   {accessor canread this, accessor canwrite this}
```

For a class with `owner`-annotated fields `f1`, ..., `fn` and `rep`-annotated fields `h1`, ..., `hn` we define `repGroup` (and the auxiliary group `ownGroup`) as follows:

```
group ownGroup = {f1, f1.ownGroup, ..., fn, fn.ownGroup};
group repGroup = {h1, h1.repGroup, h1.ownGroup, ...,
                  hn, hn.repGroup, hn.ownGroup};
```

Clarke-style ownership types also include *context parameters* which allow objects of the same type to be parameterised with different annotations. We do not consider these here, as they can simply be modelled using monomorphisation.

The big picture of this section is that we consider the shape of the aliasing graph produced by programs with Clarke-style ownership types; the edges which can exist in this graph are greatly restricted as ownership types limit which assignments are legal. We then demonstrate that all possible source-code object accesses can be verified, given the produced aliasing graph, concluding that our static analysis can verify and eliminate all contracts in a program encoding Clarke-style ownership types as aliasing contracts.

In this section, we augment rooted paths (and hence vertices of the aliasing graph) by explicitly adding ownership annotations to variables in the paths; this allows a more direct expression of the various constraints on the aliasing graph produced by programs using Clarke-style ownership types. For example, rooted path $C{:}p.v[\texttt{rep}]$ finishes with a variable v annotated with `rep`; nothing is known about the annotations of variables in path p. Path $C{:}p.q[\texttt{rep.owner}*]$ finishes with a path q consisting of a `rep`-annotated variable followed by zero or more `owner`-annotated variables: $C{:}p.q[\texttt{rep.owner}*]$ may, for example, be $C{:}p.v_1[\texttt{rep}].v_2[\texttt{owner}].v_3[\texttt{owner}]$ or $C{:}p.v[\texttt{rep}]$.

Below, we divide rooted paths in three categories according to their annotations so that each path fits unambiguously into a single category: we distinguish between rooted paths $C{:}p.q[\texttt{rep.owner}*]$ ending with one `rep` annotation followed by any number of `owner` annotations, rooted paths $C{:}p.q[\texttt{norep.owner}*]$ ending with one `norep` annotation followed by any number of `owner` annotations and rooted paths $C{:}p[\texttt{owner}*]$ which contain only `owner` annotations[11].

We now consider the aliasing graph produced by programs using Clarke-style ownership types. As our static analysis does not allow connections between vertices with different contexts, we only consider a single context C here. The same constraints as for context C apply to the structure of the aliasing graph for all other contexts; thus, if we can show that our analysis works for one context C, this implies that it will also work for all other contexts.

Only a small range of assignments are allowed by Clarke-style ownership types to ensure that an object cannot change ownership context; these can be summarised as follows:

- $C{:}p.q_1[\texttt{rep.owner}*] = C{:}p.q_2[\texttt{rep.owner}*]$[12]
- $C{:}p_1.q_1[\texttt{norep.owner}*] = C{:}p_2.q_2[\texttt{norep.owner}*]$[13]
- $C{:}p[\texttt{owner}*] = C{:}q[\texttt{owner}*]$

Applying our inference rules, we deduce the following edges:

- We seed our aliasing graph as before, turning assignments of the above form into $\rightarrow$ edges, according to (*Seeding*).
- Applying (*Aliased reflexivity*) to this initial aliasing graph, we deduce additional edges of the form $C{:}p_1.q[\texttt{rep.owner}*] \cong C{:}p_2.q[\texttt{rep.owner}*]$ where $C{:}p_1 \leftrightarrow C{:}p_2$.
- Using (*Context transfer and transitivity 1*), we then combine $C{:}p_1.q_1[\texttt{rep.owner}*] \cong C{:}p_2.q_1[\texttt{rep.owner}*]$ and $C{:}p_2.q_1[\texttt{rep.owner}*] \rightarrow C{:}p_2.q_2[\texttt{rep.owner}*]$ to derive the edge $C{:}p_1.q_1[\texttt{rep.owner}*] \rightarrow C{:}p_2.q_2[\texttt{rep.owner}*]$.
- (*Context transfer and transitivity 1*) may introduce cyclic access paths. If $C{:}p \rightarrow C{:}p.q.q$, we add an edge $C{:}p \rightarrow C{:}p.(q)$. In the ownership types aliasing graph, such a connection can exist only when all variables in q are annotated with `owner`.

In the final aliasing graph, all possible edges can be categorised as shown below; we include only aliasing and equivalence edges here as flow edges are not required for contract verification:

[11] Distinguishing between these three cases is sufficient for contract verification. Verifying the contract for rooted path $C{:}p.q[\texttt{rep.owner}*]$ leads to a $[\![\texttt{rep}]\!]$ contract verification, regardless of the annotations of p, while verifying the contract for rooted path $C{:}p.q[\texttt{norep.owner}*]$ requires a $[\![\texttt{norep}]\!]$ contract verification, also independent of the annotations of p. Therefore, it is sufficient here to categorise rooted paths by their last annotations and ignore earlier annotations.

[12] In this case, both rooted paths must share the same prefix p to ensure that an object cannot be assigned to a different ownership context.

[13] No restrictions on prefixes p_1 and p_2 are required here as the assignment is taking place in the global `norep` context.

- $C\!:\!p.v_1[\texttt{rep}].(q_1[\texttt{owner}*]) \leftrightarrow C\!:\!p.v_2[\texttt{rep}].(q_2[\texttt{owner}*])$
- $C\!:\!p_1.v_1[\texttt{norep}].(q_1[\texttt{owner}*]) \leftrightarrow C\!:\!p_2.v_2[\texttt{norep}].(q_2[\texttt{owner}*])$
- $C\!:\!(p[\texttt{owner}*]) \leftrightarrow C\!:\!(q[\texttt{owner}*])$
- $C\!:\!p_1.v_1[\texttt{rep}].(q_1[\texttt{owner}*]) \leftrightarrow C\!:\!p_2.v_2[\texttt{rep}].(q_2[\texttt{owner}*])$ can exist only if the aliasing graph also contains $C\!:\!p_1.v_1.(q_1) \cong C\!:\!p_2.v_1.(q_1)$ and $C\!:\!p_2.v_1.(q_1) \rightarrow C\!:\!p_2.v_2.(q_2)$ (and thus $C\!:\!p_2.v_1.(q_1) \leftrightarrow C\!:\!p_2.v_2.(q_2)$). This is a consequence of the application of (*Context transfer and transitivity 1*) described above, which combines these two edges to derive $C\!:\!p_1.v_1.(q_1) \leftrightarrow C\!:\!p_2.v_2.(q_2)$. The existence of the equivalence edge is essential for contract verification, as we explain below.
- There may also be additional equivalence edges, but we omit them here as they are not relevant to contract verification below.

The important result shown here is that there can be no connections in the aliasing graph between rooted paths $C\!:\!p_1.v_1[\texttt{rep}].(q_1[\texttt{owner}*])$ and $C\!:\!p_2.v_2[\texttt{norep}].(q_2[\texttt{owner}*])$.

Having constructed the augmented aliasing graph, we now demonstrate that all possible source-code object accesses in a Clarke-style ownership type program can be verified by our static analysis. We consider only object accesses conforming to our ANF-style syntax: field reads $\texttt{this}.f$ and $h.f$, field updates $\texttt{this}.f = x$ and $h.f = x$, and method calls $\texttt{this}.m()$ and $h.m()$. Programs with a wider range of object accesses can easily be transformed to adhere to such ANF-style syntax.

We also note that there is no need to differentiate between object reads and object writes because in Clarke-style ownership types each variable's read and write contracts are the same.

We merely need to show that that all arising $[\![\texttt{rep}]\!]$ contract verifications hold. We know that $\mathit{verify}([\![\texttt{norep}]\!], C\!:\!p_{\text{accessor}}, C\!:\!p_{\text{accessed}}, C\!:\!p_{\text{this}})$ always holds. The contract verification $\mathit{verify}([\![\texttt{owner}]\!], C\!:\!p_{\text{accessor}}, C\!:\!p_{\text{accessed}}, C\!:\!p_{\text{this}})$ causes indirect contract verification of the original contract's declaring object. This will eventually lead to the verification of a $[\![\texttt{rep}]\!]$ contract (which we need to show holds) or a $[\![\texttt{norep}]\!]$ contract (which we know always holds).

We note here that $\mathit{verify}([\![\texttt{rep}]\!], C\!:\!p.v[\texttt{rep}].(q[\texttt{owner}*]), C\!:\!p_{\text{accessed}}, C\!:\!p)$ always holds; given the definition of the encapsulation groups `repGroup` and `ownGroup` presented above, $C\!:\!p.v[\texttt{rep}].(q[\texttt{owner}*])$ must be in `repGroup` of $C\!:\!p$.

Similarly, $\mathit{verify}([\![\texttt{rep}]\!], C\!:\!p, C\!:\!p_{\text{accessed}}, C\!:\!p)$ holds trivially, as it satisfies the "`accessor == this`" condition of the $[\![\texttt{rep}]\!]$ contract.

Proposition 2. *All arising $[\![\texttt{rep}]\!]$ contract verifications hold.*

Proof. We look at all possible ANF object accesses, showing that each required $[\![\texttt{rep}]\!]$ contract verification holds. As above, we consider source-code accesses to variable v and distinguish the two separate cases $v = \texttt{this}$ and $v = f$:

Access to v, with $v = \texttt{this}$ in class D: This requires contract verification for all rooted paths $C\!:\!p$ where $\tau(C\!:\!p) \bowtie D$:

- For $C\!:\!p.v[\texttt{rep}].(q[\texttt{owner}*])$ we must check $\mathit{verify}([\![\texttt{rep}]\!], C\!:\!p.v[\texttt{rep}].(q[\texttt{owner}*]),$ $C\!:\!p.v[\texttt{rep}], C\!:\!p)$ which holds as explained above[14].

[14]This contract verification is an indirect contract verification, required for the verification of the $[\![\texttt{owner}]\!]$ contracts of q.

- For $C\!:\!p.v$[norep].(q[owner$*$]) we do not require verification of ⟦rep⟧ contracts.
- For $C\!:\!(p_1$[owner$*$]) we require the verification of the contracts of all rooted paths $E\!:\!p_2$ such that $\tau(E\!:\!p_2) \bowtie C$[15]. $E\!:\!p_2$ can have one of three possible forms:
  - For a rooted path $E\!:\!p_2.v_2$[rep].(q_2[owner$*$]) our analysis requires *verify*(⟦rep⟧, $E\!:\!p_2.v_2$[rep].(q_2[owner$*$]).(p_1[owner$*$]), $E\!:\!p_2.v_2$[rep], $E\!:\!p_2$) which holds as explained above.
  - For $E\!:\!p_2.v_2$[norep].(q_2[owner$*$]) we do not require any ⟦rep⟧ contract verifications.
  - For $E\!:\!p_2$[owner$*$], verification proceeds recursively as for the original rooted path $C\!:\!(p_1$[owner$*$]) until either a ⟦norep⟧ or ⟦rep⟧ contract is verified; the two previous cases show that these verifications must hold.

Access to v, with $v = f$: This requires verification of the contracts of rooted paths ending in f and all aliased rooted paths:

- For a rooted path $C\!:\!p.v$[rep].(q[owner$*$]) with final term f, we require *verify*(⟦rep⟧, $C\!:\!p.v$[rep].(q[owner$*$]), $C\!:\!p.v$[rep], $C\!:\!p$) which holds as explained above.

  In addition, we have to verify the contracts of $C\!:\!p.v_2$[rep].(q_2[owner$*$]), where we have $C\!:\!p.v.(q) \leftrightarrow C\!:\!p.v_2.(q_2)$: *verify*(⟦rep⟧, $C\!:\!p.v$[rep].(q[owner$*$]), $C\!:\!p.v_2$[rep], $C\!:\!p$) also holds as explained above.

  We do not have to verify contracts for $C\!:\!p_2.v_2$[rep].(q_2[owner$*$]) where $C\!:\!p.v.(q) \leftrightarrow C\!:\!p_2.v_2.(q_2)$; we know that there must exist a rooted path $C\!:\!p.v_2$[rep].(q_2[owner$*$]) such that $C\!:\!p.v.(q) \leftrightarrow C\!:\!p.v_2.(q_2)$ and $C\!:\!p.v_2.(q_2) \cong C\!:\!p_2.v_2.(q_2)$; otherwise our analysis could not have deduced $C\!:\!p.v.(q) \leftrightarrow C\!:\!p_2.v_2.(q_2)$. This implies that the contract verification can be skipped due to the equivalence with $C\!:\!p.v_2.(q_2)$ whose contract will already be successfully verified by the previous paragraph.
- For a rooted path $C\!:\!p.v$[norep].(q[owner$*$]) whose last term is f, we require verification of ⟦norep⟧ contracts only. We know that $C\!:\!p.v$[norep].(q[owner$*$]) can only be aliased with rooted paths $C\!:\!p_2.v_2$[norep].(q_2[owner$*$]), which also do not require ⟦rep⟧ contract verifications.
- For $C\!:\!(p$[owner$*$]) whose last term is f, we need to find and verify the contracts of all rooted paths $E\!:\!p_2$ which satisfy $\tau(E\!:\!p_2) \bowtie C$. As above, $E\!:\!p_2$ may have one of three forms:
  - Rooted path $E\!:\!p_2.v_2$[rep].(q_2[owner$*$]) requires ⟦rep⟧ verification *verify*(⟦rep⟧, $E\!:\!p_2.v_2$[rep].(q_2[owner$*$]).(p[owner$*$]), $E\!:\!p_2.v_2$[rep], $E\!:\!p_2$) which holds as explained above.
  - Rooted path $E\!:\!p_2.v_2$[norep].(q_2[owner$*$]) does not require ⟦rep⟧ contract verifications.
  - Rooted path $E\!:\!p_2$[owner$*$] again represents the recursive case; it eventually requires verification of ⟦rep⟧ and ⟦norep⟧ contracts, which must hold according to the previous two cases.

[15]As above, this contract verification is an indirect verification required to verify the ⟦owner⟧ contracts of p_1.

We also need to verify the contracts of all rooted paths $C\!:\!(q[\texttt{owner}*])$ such that $C\!:\!(p[\texttt{owner}*]) \leftrightarrow C\!:\!(q[\texttt{owner}*])$. As for the contract verification of $C\!:\!(p[\texttt{owner}*])$, we need to find and verify the contracts of all rooted paths $E\!:\!p_2$ which satisfy $\tau(E\!:\!p_2) \bowtie C$, leading to very similar contract verifications:

– Rooted path $E\!:\!p_2.v_2[\texttt{rep}].(q_2[\texttt{owner}*])$ requires $[\![\texttt{rep}]\!]$ verification $\mathit{verify}([\![\texttt{rep}]\!],$ $E\!:\!p_2.v_2[\texttt{rep}].(q_2[\texttt{owner}*]).(q[\texttt{owner}*]), E\!:\!p_2.v_2[\texttt{rep}], E\!:\!p_2)$ which holds.
– As above, rooted path $E\!:\!p_2.v_2[\texttt{norep}].(q_2[\texttt{owner}*])$ does not require $[\![\texttt{rep}]\!]$ contract verifications.
– As above, rooted path $E\!:\!p_2[\texttt{owner}*]$ represents the recursive case which must hold given the previous two cases.

□

This shows that all possible source-code object accesses in a program using Clarke-style ownership types can be verified and thus eliminated using our static analysis of aliasing contracts (provided the program is well-ownership-typed).

Clarke-style ownership types are a transitive owners-as-dominators system; thus, this shows that our static analysis can verify transitive owners-as-dominators encapsulation.

It is straightforward to adapt the above proof to transitive owners-as-modifiers systems. In these systems, the read contracts of all variables are "`true`"; thus, proving the validity of read object accesses becomes trivial. The write contracts match the contracts of a transitive owners-as-dominators system; these can be verified as we proved above.

6.3.2 Verifying full encapsulation

In Section 4.11, we used aliasing contracts to show that full encapsulation (as for example provided by Hogg's islands [48] and Almeida's balloons [4](is a special case of Clarke-style ownership types. This implies that if our static analysis can verify the contracts modelling Clarke-style ownership types (as we proved above), this result must also hold for programs which use aliasing contracts to implement full encapsulation.

6.4 StatCon: practical static analysis for Java

In this section, we present StatCon, a Java prototype implementation of the static analysis described above; we implement StatCon as an extension of JaCon (presented in Chapter 5). Section 6.4.1 describes how StatCon works and how it interacts with JaCon. In Section 6.4.2 we conduct a performance evaluation which shows that StatCon can be used to analyse large real-world Java programs.

The prototype implementation of StatCon we describe here is simple and rather limited; however, it gives us a good indication of the feasibility and benefits of combining static analysis with run-time contract evaluation.

6.4.1 Description of StatCon

StatCon is designed to work closely with JaCon, our prototype for aliasing contracts in Java presented in Chapter 5. We modified the existing JaCon compiler to output information about a program during the compilation process, including information about classes, variables and their contracts, assignments and object accesses. StatCon uses this information as input.

As explained above, the analysis works in two stages: first, StatCon analyses the relationships between variables; then, it uses this information to verify the contracts for each object access in the program.

At the end of the analysis process, StatCon outputs two separate results: a list of variables whose contracts may be violated at run time and a list of source-code object accesses that may fail at run time. When StatCon analyses an object access and cannot prove its correctness, it adds the object access to the list of potentially failing object accesses and the contract which caused the failure to the list of potentially failing contracts.

JaCon in turn uses this information as input: during compilation, it removes the contracts of all variables, except those which may be violated at run time according to StatCon; these contracts need to be tracked as usual while the program executes. JaCon also omits contract evaluation before object accesses, except for those object accesses which StatCon has determined may fail. In this way, JaCon reduces the number of contracts which need to be tracked and the number of contract evaluations which need to be performed. In the best case, where StatCon can fully verify all contracts in a program, no contract tracking or evaluation would be required at all. This case is equivalent to what is achieved by static alias protection schemes such as Clarke-style ownership types, which do not require any run-time aliasing checks.

Unlike the theoretical analysis we presented earlier which only works with a simplified ANF-style syntax, StatCon can analyse programs with full Java syntax.

The analysis proposed above requires variable names to be unique within a class; additionally, all inherited versions of the same method must have the same return statement. StatCon automatically refactors the source code to meet these requirements.

For simplicity, StatCon uses a Steensgaard-style analysis rather than an Andersen-style analysis as originally proposed above; it treats assignments as bidirectional. This is simpler to implement in practice, since it means that the set of flow edges ($\rightarrow$) and the set of aliasing edges ($\leftrightarrow$) coincide.

StatCon also does not consider cycles. The analysis we presented above substitutes a summary vertex $C{:}p.f.(q.f)$ when it encounters a vertex $C{:}p.f.q.f$ in which a field name f occurs more than once. This ensures that the total number of vertices remains finite. StatCon stops the creation of vertices of the form $C{:}p.f.q.f$ (to ensure a finite number of vertices) but, for simplicity does not create a summary vertex. This means that it lacks some aliasing information and as a result can correctly verify only a small range of contracts (if situations requiring summary vertices occur in a program); for example, the contracts "`true`" and "`accessor == accessed`" are unaffected by the presence of cyclic access path[16]; they can thus be correctly verified without creating summary vertices.

Although StatCon currently does not create summary vertices, this simple version is sufficient for our performance evaluation, as we explain below. Extending the current prototype to support summary vertices with cyclic access paths would be straightforward.

In order to maintain correctness, StatCon must analyse a program's entire source code, including the source code of libraries used by the program. Consider the Java code

[16]This is the case, at least, for programs without the contracts "`accessor canread this`" and "`accessor canwrite this`". In such programs the rooted paths representing `accessor` and `accessed` can never be cyclic and therefore the evaluation of "`accessor == accessed`" is not affected by the presence of cyclic access paths.

below, where `Stack` is a library class:

```
Stack<Foo> stack = new Stack<Foo>();
Foo x = new Foo();
stack.push(x);
Foo y = stack.pop();
```

When executed, this code creates aliasing between x and y but StatCon will only discover this relationship if it analyses the `Stack` library class in addition to the program's source code. Analysing collection libraries is particularly important since they store objects and are likely to create aliasing, as in the example above.

6.4.2 Performance evaluation

To evaluate the performance of StatCon, we again use the five real-world Java program to which we applied JaCon in Chapter 5: FindBugs, JGraphT, JUnit, NekoHTML and Trove. First, we use StatCon to analyse these programs, using the default contracts "`accessor == accessed || accessor == this`" for non-static, non-public fields and "`true`" to all other variables. We record which contracts and object accesses need to be evaluated at run time according to StatCon and which can be eliminated.

We noted above that StatCon does not use summary vertices with cyclic access paths and therefore can only correctly analyse a small range of contracts. The default contracts we use here, "`accessor == accessed || accessor == this`" and "`true`" can both be analysed without considering cycles: as noted above, the contracts "`true`" and "`accessor == accessed`" (and thus "`accessor == accessed || accessor == this`") are unaffected by the presence of cyclic access paths.

Second, we feed information about required contracts and object accesses to JaCon and execute the unit tests of the programs, as we did in Chapter 5. This allows us to measure the performance improvement achieved by combining static and dynamic checking of contracts, where the static analysis can eliminate some contract tracking and evaluation from the run-time execution of the program.

Table 6.1 and Table 6.2 show the results of applying StatCon to our five test programs. Table 6.1 presents information about the input programs, including LoC, number of variables and number of object accesses. Trove is the largest program, with 239,266 LoC, 16,632 variables and 88,292 object accesses. Table 6.1 also shows the time required to statically analyse each program with StatCon; *total time* is the time taken by StatCon, while *analysis time* is the time taken for the analysis of contracts only, excluding input and output time. We can see that all programs, including large programs like Trove, can be analysed by StatCon in under 10 seconds. The actual analysis time is even lower, less than 7 seconds. Input and output take up a significant portion of the total time because StatCon must read in large files with information about variables and object accesses in a program before starting its analysis.

Table 6.2 shows information about the output of StatCon, including the number of required contracts and object accesses in each program. This demonstrates that only a small proportion of contracts and object accesses actually need to be evaluated at run time, while StatCon can prove that most will not cause contract violations.

Next, we take the output produced by StatCon and feed it into JaCon. The updated JaCon compiler, which we call `javacStat` here, removes tracking of contracts and evaluation of object accesses which are not required according to StatCon. We compare

Program	LoC	Variables	Object Accesses	Total time (in s)	Analysis time (in s)
FindBugs	192,259	25,345	113,273	9.40	6.96
JGraphT	32,899	2,570	9,917	1.02	0.52
JUnit	11,994	1,645	4,824	0.40	0.08
NekoHTML	11,482	1,122	6,817	0.36	0.03
Trove	239,266	16,632	88,292	9.46	5.41

Table 6.1: Static analysis measurements – input program size and analysis time

Program	Required contracts	Percentage	Required accesses	Percentage
FindBugs	187	0.74%	4,878	4.31%
JGraphT	24	0.93%	249	2.51%
JUnit	2	0.12%	13	0.27%
NekoHTML	9	0.80%	35	0.51%
Trove	117	0.70%	28,145	31.88%

Table 6.2: Static analysis measurements – analysis results

the performance of programs compiled with `javacStat` to the performance of programs when compiled with `javac0` (no contracts) and `javac2` (the same default contracts as `javacStat`: "`accessor == accessed || accessor == this`" for non-static, non-public fields and "`true`" for all other variables). Our performance evaluation in Chapter 5 concluded that although these default contracts are in some sense artificial, they provide a good approximation for the actual encapsulation used in the test programs. The compilers `javac0` and `javac2` do not interact with StatCon and are described in detail in Chapter 5; `javac0` gives us a base-line performance without any contract evaluation, while `javac2` shows the performance of programs with full dynamic contract checking.

Although we already measured the performance of the five test programs with `javac0` and `javac2` in Chapter 5, we remeasure their performance here. The performance evaluation in Chapter 5 was conducted more than six months before this one; in this period of time, the performance of the laptop used for the experiments decreased markedly. Thus, remeasuring was required to obtain comparable results.

Table 6.3 shows the execution time for programs compiled with `javac0`, `javac2` and `javacStat`. Here, we take two separate measurements using `javacStat`: the version called `javacStat1` below *reduces contract tracking*; it removes contracts which are not required according to StatCon, but evaluates all object accesses. The version called `javacStat2` below *reduces contract evaluation*; it tracks all contracts but only evaluates the object accesses which may fail at run time according to StatCon. The two different measurements show us which approach performs better; alternatively, the two could be combined.

The data shows a significant performance improvement for both versions of `javacStat` compared to `javac2`. JGraphT, for example, runs 17.5 times more slowly with `javac2` than with `javac0`, but only about 11.5 times more slowly with `javacStat`; this corresponds to a performance improvement of around 30%.

If StatCon works correctly, we would expect to see the above performance improvement while still measuring the same number of contract violations as with `javac2`. A

Program	`javac0` (in s)	`javac2` (in s)	Ratio	`javacStat1` (in s)	Ratio	`javacStat2` (in s)	Ratio
FindBugs	35.08	42.69	1.22	41.19	1.17	41.27	1.18
JGraphT	5.70	100.02	17.55	66.59	11.69	65.44	11.48
JUnit	15.79	22.82	1.44	21.82	1.38	20.30	1.29
NekoHTML	3.01	3.80	1.26	3.60	1.20	3.19	1.06
Trove	7.11	55.57	7.82	49.27	6.93	20.95	2.95

Table 6.3: Run-time performance measurements – execution time

Program	`javac2`	`javacStat1`	Percentage	`javacStat2`	Percentage
FindBugs	339	44	12.98%	65	19.17%
JGraphT	77,015	73,290	95.16%	10	0.01%
JUnit	789	0	0%	0	0%
NekoHTML	38,163	26,320	68.97%	19,380	50.78%
Trove	306,205	240,669	78.60%	226,053	73.82%

Table 6.4: Run-time performance measurements – contract evaluation failures

smaller number of contract violations would indicate that some contracts which fail at run time with `javac2` have been mistakenly eliminated by StatCon. However, as we mentioned above, StatCon must analyse the entire source code of a program, including all libraries, to work correctly. In this experiment we only analyse the program's direct source code for simplicity. As a result, StatCon cannot deduce all aliasing relationships in the programs and thus misses some contract violations. This is not a problem with the analysis itself, but with the incomplete input used during this experiment.

Table 6.4 shows the number of contract violations which occurred when using the two versions of `javacStat` compared to `javac2`; `javacStat1` and `javacStat2` measure fewer violations than `javac2` for all of our test programs, indicating that aliasing caused by libraries is common. For `JGraphT`, `NekoHTML` and `Trove`, we can detect the majority of contract violations, but none are discovered for `JUnit`. This shows that aliasing in `JUnit` always occurs through libraries (probably collections).

In addition to the results presented here, we also measure compilation times, memory usage and the number of contract additions, removals and evaluations performed when the programs execute. We observe that compilation with the two versions of `javacStat` is slightly slower than `javac2`; this is expected since, in addition to the work done by `javac2`, `javacStat` has to remove contract tracking and evaluation statements that are not required. Unsurprisingly, programs compiled with `javacStat1` (which reduces contract tracking) perform a lower number of contract additions and removals (thus decreasing memory usage), while programs compiled with `javacStat2` (which reduces contract evaluations) perform fewer contract evaluations (but exhibit similar memory usage to programs compiled with `javac2`).

Overall, our performance evaluation demonstrates that StatCon is capable of analysing very large programs in a short amount of time. The combination of static and dynamic contract checking provides significant performance improvements and we thus conclude that it is a promising approach to making aliasing contracts more usable in practice. However, to maintain correctness, it is important for StatCon to verify the entire source code of the program, including libraries, significantly increasing the amount of code that needs to be analysed. Given the good performance of StatCon even for large programs, this certainly appears feasible.

6.5 Summary

In this chapter, we presented a static analysis which can verify many aliasing contracts at compile time. The analysis works by first constructing an aliasing graph to represent the possible aliasing relationships which can exist at run time. May-aliasing is used, meaning that a relationship in the aliasing graph must not necessarily exist at run time. On the other hand, if no relationship between two rooted paths is shown in the aliasing graph, such a relationship can certainly not exist at run time. We proved the soundness of the analysis, demonstrating that it discovers all aliasing relationships which are possible at run time.

In a second step, our proposed static analysis uses the aliasing graph to assess the validity of every object access in a program's source code. For each such object access, the analysis verifies the contracts of all variables which may at run time point to the accessed object, given the information in the aliasing graph. If a contract cannot be verified, an error is reported.

We demonstrated that this static analysis can verify aliasing contracts used to model existing alias protection schemes; for example, programs translated to aliasing contracts from Clarke-style ownership types and full encapsulation can be fully verified in this way.

Finally, we presented StatCon, a prototype implementation of the static contract analysis for Java. We used StatCon to show that statically analysing even large programs is feasible and that combining static and dynamic contract checking provides significant performance improvements.

CHAPTER 7

Discussion

In previous chapters, we proposed, characterised and implemented aliasing contracts, a novel alias protection scheme for OO programming languages. In this chapter, we discuss and summarise the important aspects of aliasing contracts. We also identify and briefly discuss promising future research directions.

7.1 Expressiveness

Aliasing contracts are highly expressive and can support the definition of many different aliasing conditions. We demonstrated this in our case study in Chapter 3, where we presented a number of possible aliasing contracts for an iterator.

Aliasing contracts can contain arbitrary boolean conditions (including calls to boolean methods). In addition, they introduce the concept of *encapsulation groups* which allow us to group objects, giving access rights to all objects in a group rather than just single objects. The real power of encapsulation groups lies in the fact that they can contain an unbounded number of objects which can change at run time. We can thus give access rights to a group of objects without knowing exactly how many objects will be in the group during program execution. In addition, encapsulation groups can contain other encapsulation groups, thus enabling the definition of deep or transitive aliasing conditions, where one object encapsulates directly and transitively referenced objects.

Contract parameters which are analogous to Clarke et al.'s context parameters further increase expressiveness. They support contract polymorphism, where objects of the same class exhibit different aliasing behaviour. This is, for example, useful for collection classes, where the desired encapsulation for the data stored in the collection may change from one collection instance to another.

In Chapter 4, we showed that aliasing contracts can be used to express various different aliasing policies including full encapsulation and three variants of owners-as-dominators and owners-as-modifiers (strict, peer and transitive). They can also model uniqueness, linearity and module encapsulation if we slightly adjust the semantics of these aliasing policies to fit dynamic checking of object accesses (rather than static checking of references). For example, aliasing contracts cannot enforce the existence of only a single unique reference to an object, but can restrict object accesses to a single unique `accessor` object; the reason for this is that aliasing contracts do not deal with references at all, only object accesses. The ability of aliasing contracts to model so many different aliasing policies demonstrates their high level of expressiveness, particularly compared to existing alias protection schemes which usually support only a single aliasing policy.

Analogous to dynamic type checking, much of the flexibility of aliasing contracts stems from their dynamic approach to alias protection. Static alias protection systems tend to be inflexible because, like static type checkers, they must make conservative assumptions to prove a program's correctness. As a result, static systems may reject programs which in fact exhibit correct alias protection at run time.

With aliasing contracts, the result of a contract evaluation depends on the aliasing structure of the program when an object is accessed at run time. As the aliasing structure changes, the result of a contract evaluation may also change. Thus, an object access may be illegal at one point during program execution, but allowed later on. This also means that access rights for an object can easily be transferred from one part of a system to another, as the aliasing structure changes. Many static systems struggle to support such ownership transfer, but with aliasing contracts it occurs naturally as the program executes.

Many static alias protection schemes also lack support for multiple ownership, where an object is encapsulated, yet shared by several encapsulating objects at the same time. Multiple ownership is, for example, essential for the efficient implementation of iterators: the items in the collection need to be shared between the iterator and the collection, but protected from aliasing from the rest of the system. Multiple ownership is straightforward to implement with aliasing contracts, as the iterator case study we conducted in Chapter 3 shows.

Despite the expressiveness of aliasing contracts, we argue that they remain simple for developers to write. Specifying an aliasing contract for a variable requires a developer to think only about the aliasing requirements of this particular variable and the object it contains. In other systems developers must structure the entire system in a particular way (for example, to achieve a tree-shaped ownership graph for Clarke-style ownership types). This is far more difficult to achieve as it requires developers to think about the structure of the *entire* system, rather than just the *local* aliasing requirements. This also complicates design changes and program maintenance because they can disrupt the program's aliasing structure.

Aliasing contracts are similar in spirit to assertions, which are easy to specify and widely used. Both describe assumptions at a particular point in the code. However, aliasing contracts are significantly more expressive. The contracts which are evaluated depend on the aliasing structure of the program at run time; one object access can trigger multiple contract evaluations. In addition, the advanced features of aliasing contracts, including encapsulation groups, allow the expression of complex conditions that are difficult to describe using standard boolean expressions.

7.2 Unification of existing work

A large amount of research has already been done on alias protection. The variety of approaches and aliasing policies is huge as discussed in Chapter 2. There is currently no unifying system or theory (although Boyland et al.'s work on capabilities [21] is arguably a first step towards unification); this makes it difficult to compare systems and understand their relative advantages and disadvantages.

As mentioned above, aliasing contracts are able to model a wide range of existing alias protection schemes and their aliasing policies, including full encapsulation, strict, peer and transitive owners-as-dominators and owners-as-modifiers, and module encapsulation. This suggests that aliasing contracts provide a unifying approach to the area of alias protection.

We suggest that expressing many different alias protection schemes in a common base language – aliasing contracts – greatly simplifies direct comparisons between otherwise seemingly different systems. In Chapter 4, for example, we used aliasing contracts to show that full encapsulation is a special case of Clarke-style ownership types.

7.3 Run-time performance

Many existing alias protection schemes can be checked at compile time and thus have no impact on program execution. Aliasing contracts, on the other hand, require dynamic contract evaluation; as a result, they cause significant performance overheads.

In Chapter 5 we used JaCon, our prototype implementation of aliasing contracts in Java, to quantify this performance overhead. We concluded that the impact of aliasing contracts on performance varies widely between different programs, depending on their run-time aliasing properties; for example, programs which build large and complex data structures are likely to be significantly affected. While some of our test programs were slowed down by less than 50%, others executed up to 50 times more slowly with aliasing contracts than without.

Some test programs (such as JUnit and NekoHTML), which are not significantly affected by aliasing contracts, could be fully utilised in the presence of aliasing contracts. For other programs, such as JGraphT and FindBugs, the performance overhead would render the programs unusable in practice. However, in Chapter 5, we noted that running the unit tests of all of our test programs with aliasing contracts still remains feasible. Thus, aliasing contracts can be used as a debugging tool to help discover unexpected aliases and associated bugs (and in fact their performance is comparable to existing debugging tools such as Valgrind); after testing, they can then be removed from the release versions of programs. This approach is similar to that taken for assertions and software contracts, which are used during testing but are disabled once the program is shipped to customers.

In Chapter 6 we presented static analysis which can verify some simple aliasing contracts at compile time. Although not all aliasing contracts may be able to be checked in this way, verified contracts can be removed from the program during the compilation process, reducing the number of contract evaluations at run time and improving performance. Warnings could be displayed to alert developers to contracts which cannot be removed in this way.

We demonstrated in Chapter 6 that a combination of static and dynamic contract checking produces performance improvements. We developed StatCon, a prototype implementation of the static contract analysis for Java, and used it to show that, unsurprisingly, programs execute significantly faster when verified contracts are removed at compile time; for some programs, we measured performance improvements of up to 30% compared to using only dynamic contract evaluation.

We also showed in Chapter 6 that the contracts required to model existing static alias protection schemes, including transitive owners-as-dominators (as for example used by Clarke-style ownership types [27]), transitive owners-as-modifiers and full encapsulation, can all be verified at compile time by our static analysis. Programs using these existing static alias protection schemes can thus be translated into aliasing contracts *without* incurring any performance penalty. Dynamic contract checking is required only when more expressiveness is required than is afforded by existing schemes. This essentially removes the run-time performance disadvantage of aliasing contracts compared to existing work.

7.4 Future directions

We now look at possible future research directions that we uncovered during our work on aliasing contracts.

7.4.1 Owners-as-locks

Owners-as-locks, first implemented by Boyapati et al. [14, 16], applies owners-as-dominators to concurrent programs. It is based on the idea that if a thread owns a particular object (and every object has a single owner) it can safely read and write the object without requiring a lock, since no other thread can simultaneously own the same object. Ownership thus acts as an implicit locking mechanism.

We suggest that aliasing contracts can similarly be seen as implicit locks on the objects to which they apply. For example, the rw-contract "`accessor == accessed || accessor == this`" represents a complete lock which restricts both object reads and writes. If object o_1 holds a reference to object o_2 with this contract, it is guaranteed that no accesses from objects other than o_1 (and o_2 itself) to o_2 can occur; any such accesses would cause a contract violation. This implies that o_1 can safely access o_2 without acquiring a lock first.

Other aliasing contracts can simulate more complex locking behaviour. For example, contract "`true, accessor == accessed || accessor == this`" represents a write lock which restricts object writes but allows (potentially concurrent) object reads.

Aliasing contracts do not model exactly the same locking semantics as ownership. If o_1 holds a reference to o_2 with rw-contract "`accessor == accessed || accessor == this`", this does not guarantee that o_1 has access to o_2; another object may hold a reference to o_2 with a conflicting contract, thus preventing o_1 from accessing o_2. Ownership, on the other hand, encapsulates an object inside the owner while at the same time *guaranteeing* access rights for the owner.

Although we have not investigated this topic in detail, we give a quick overview of how aliasing contracts could be used for implicit locking. We suggest the introduction of a `lock`-block, in which an object is considered locked by aliasing contracts. The example below shows how the proposed `lock`-block works:

```
Object obj {accessor == accessed || accessor == this};  //A
lock(obj = getObject()) {                               //B
  ...
}                                                       //C
```

First, we create a variable `obj` at program point `A` to act as the locking mechanism. In the example above, we use the rw-contract "`accessor == accessed || accessor == this`" to prevent both reads and writes from other objects.

Secondly, the lock is requested using the `lock` operator at program point `B`. The `lock` operator is followed by an assignment operation which points the locking variable (`obj`) to the object to be locked (the object returned by the `getObject` method in our example). The `lock` operator ensures that the assignment is performed only when there are no conflicting aliasing contracts; that is, contracts whose evaluation would fail. To identify conflicting contracts, it checks if the object can be read and written using the locking variable. If either the read or the write through the locking variable fails, there must be a conflicting contract. In this case, `lock` waits until the conflicting contract disappears before performing the assignment.

Once the assignment statement is executed, the locking variable holds a lock to the object it contains. In the above example, any accesses from other parts of the system will cause a violation of the rw-contract "`accessor == accessed || accessor == this`" of the locking variable `obj`. If any other parts of the system try to acquire a lock for the same object, the `lock`-block will force them to wait until the previous lock has been released. This release occurs at the end of the `lock`-block at program point `C` in the above program.

The `lock` operator outlined above would allow easy locking of objects. Unlike the owners-as-dominators framework used by Boyapati et al., aliasing contracts could be used to express a wide range of locking conditions.

7.4.2 Or-contracts

Contract evaluation is conjunctive; a violation is reported if any of the contracts associated with the accessed object evaluate to `false`. This is important to ensure that encapsulation guarantees cannot be subverted.

This approach raises the question of what would happen if contract evaluation was disjunctive instead. What would be the semantic meaning and applications of such "or-contracts"?

We suggest that or-contracts (or *disjunctive contracts*) could be an alternative approach to implementing multiple ownership. Currently, aliasing contracts require owners (or at least one owner) to be aware of the existence and identity of the other owners. Otherwise, owners might specify conflicting contracts for the multiply owned object, thus blocking other owners from accessing it. We discussed multiple ownership with aliasing contracts in detail in Chapter 4.

Or-contracts could be useful if owners know about the existence of other owners but do not know their exact identity. Each owner could specify an encapsulating contract for the multiply owned object, such as "`accessor == accessed || accessor == this`". Instead of these incompatible contracts blocking other owners from accessing the object, disjunctive contract evaluation would succeed if *at least one* owner's contract evaluates to `true`.

We could then distinguish between disjunctive and standard conjunctive contracts and combine them in contract evaluation as follows: for disjunctive contracts $dis_1 ... dis_m$ and conjunctive contracts $con_1 ... con_n$ of an accessed object, contract evaluation would be

$$(dis_1 \mid\mid ... \mid\mid dis_m) \;\&\&\; con_1 \;\&\&\; ... \;\&\&\; con_n$$

At least one disjunctive contract as well as all standard contracts must be satisfied for the evaluation to succeed.

While this is an interesting idea in theory, more thought is needed about how to ensure that encapsulation cannot be circumvented by other parts of the system. Applying or-contracts to all situations would provide no encapsulation guarantees at all: in the presence of contract disjunction, accesses to encapsulated objects from other parts of the system would be valid, even if the contract which is supposed to encapsulate the object evaluates to `false`. The combination of conjunctive and disjunctive contract evaluation we suggested above addresses this problem, by allowing developers to specify which contracts are conjunctive and which are disjunctive. Making important encapsulating contracts conjunctive would then ensure that they cannot be circumvented.

7.4.3 Andersen-style and Steensgaard-style static analysis

In Chapter 6, we developed a static analysis for aliasing contracts. Although our final approach used an Andersen-style alias analysis, we originally developed a Steensgaard-style analysis.

Andersen-style alias analyses are subset-based; assignments are treated as unidirectional (that is, data flows from one variable into another, but not vice versa). Steensgaard-style analyses, on the other hand, are equality based; assignments are bidirectional and therefore create data flow in both directions between assigned variables. Andersen-style analyses are more accurate since assignments are unidirectional in practice.

We originally used a Steensgaard-style analysis for our static analysis as it appeared to be simpler to implement. Our aim for the analysis was to be able to verify programs translated to aliasing contracts from existing alias protection schemes, such as Clarke-style ownership types. A Steensgaard-style analysis is powerful enough for this purpose. Many existing schemes such as Clarke-style ownership types enforce their aliasing policy using a static type system which restricts assignments. In Clarke-style ownership types, for example, an object in one ownership context cannot be moved to another context; the ownership contexts of both sides of assignment statements must be the same. This can be modelled by a Steensgaard-style analysis which assumes that assignments are bidirectional.

During development, we changed to a subset-based Andersen-style analysis. Such an analysis allows us to eliminate many superfluous edges in the aliasing graph and greatly decreases the number of possibilities that need to be considered by the analysis. As it is more powerful than a Steensgaard-style analysis, this Andersen-style analysis can still model existing systems as described above.

This leads us to an interesting research question: what would happen if we applied an Andersen-style analysis to existing alias protection schemes such as Clarke-style ownership types, instead of the Steensgaard-style analysis they currently use? More importantly, would this be a useful thing to do? Such an approach would allow assignment between two variables as long as the aliasing properties (or the ownership context in Clarke-style ownership types) of the left-hand side is a *subset* of the aliasing properties of the right-hand side[1]. Objects could then flow from looser to more restrictive variables but not vice versa. This is analogous to typing rules in assignments, where objects can flow from a variable of a subtype to a variable of a supertype but not the other way around.

[1]Of course this would require careful definition of what it means for one set of aliasing properties (or one ownership context) to be the subset of another.

CHAPTER 8

Conclusions

In this thesis, we have presented aliasing contracts, a dynamic approach to alias protection in OO programming. Aliasing contracts can be used to detect unintended reference leaks, thus addressing the negative consequences of aliasing, including representation exposure. Specifically, we have made the following contributions to the area of alias protection, as presented in this thesis:

- We presented a detailed overview and comparison of existing work in Chapter 2 to place our work on aliasing contracts in the context of previous research.
- We developed aliasing contracts (presented in Chapter 3) which address the limitations of flexibility and expressiveness of many existing alias protection schemes. Aliasing contracts, being dynamically checked, can be used to express aliasing conditions that are not (or only partially) supported by existing work, such as multiple ownership and ownership transfer. Therefore, they can be used to implement idioms and design patterns, including iterators, that existing schemes struggle to model.
- We demonstrated the expressiveness and flexibility of aliasing contracts using a case study in Chapter 3. We showed how aliasing contracts can be used to model and enforce encapsulation in the iterator design pattern, which is frequently discussed in alias protection literature.
- In Chapter 4, we placed aliasing contracts in the context of existing work and demonstrated that they can be used to model many existing aliasing policies. We thus argued that they can be seen as a unifying approach to the area of alias protection and can serve as a base language in direct comparisons between different alias protection schemes.
- We developed and tested JaCon (described in Chapter 5), a prototype implementation of aliasing contracts in Java, demonstrating the feasibility of adding aliasing contracts to existing programming languages.
- We used JaCon to quantify the run-time performance of aliasing contracts (presented in Chapter 5), concluding that aliasing contracts can feasibly be used as a testing and debugging tool. Our performance evaluation also demonstrated the robustness of JaCon, which we used to compile and execute several hundred thousand LoC.

- We developed static analysis (presented in Chapter 6) which can verify many simple aliasing contracts at compile time. We showed that our static analysis can verify the contracts of programs translated to aliasing contracts from existing alias protection schemes, such as Clarke-style ownership types.

- We developed StatCon (described in Chapter 6.2), a prototype implementation of the static contract analysis for Java and tested it on several real-world Java programs. In this way, we showed that static analysis is feasible in practice, even for large programs, and that combining static and dynamic contract checking leads to significant performance improvements compared to using only dynamic contract checking.

Future implementation work could focus on further developing our two prototypes, JaCon and StatCon, with the aim of producing industry-standard tools. Currently, both are proof-of-concept implementations with several limitations that make them difficult to use in practice. Error messages in JaCon, for example, are rather cryptic at the moment. StatCon currently lacks some desirable features, including support for cyclic access paths, contract suspension and contract parameters. In addition, we expect that further development and a focus on performance optimisation could significantly improve run-time performance of both prototypes.

In Chapter 7, we proposed several further avenues for future research work, including using aliasing contracts for locking, investigating the semantics and possible applications of disjunctive contract evaluation, and exploring the application of Andersen-style static analysis (that is, subset-based analysis) to existing alias protection schemes.

Bibliography

[1] M. Abi-Antoun and J. Aldrich. Ownership domains in the real world. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 93–104, 2007.

[2] J. Aldrich and C. Chambers. Ownership domains: separating aliasing policy from mechanism. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2004.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'02, pages 311–330. ACM, 2002.

[4] P. Almeida. Balloon types: controlling sharing of state in data types. In M. Aksit and S. Matsuoka, editors, *ECOOP 1997 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer Berlin Heidelberg, 1997.

[5] L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1994.

[6] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language.* Addison-Wesley, 2000.

[7] H. Baker. 'Use-once' variables and linear objects: storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[8] A. Banerjee and D. Naumann. State based encapsulation for modular reasoning about behavior-preserving refactorings. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 319–365. Springer Berlin Heidelberg, 2013.

[9] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In F. Boer, M. Bonsangue, S. Graf, and W. Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006.

[10] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[11] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg, 2004.

[12] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 3rd edition, 2004.

[13] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, US, February 2004.

[14] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'02, pages 211–230. ACM, 2002.

[15] C. Boyapati, B. Liskov, and L. 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.

[16] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'01, pages 56–69. ACM, 2001.

[17] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI'03, pages 324–337. ACM, 2003.

[18] J. Boyland. Alias burying: unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, May 2001.

[19] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer Berlin Heidelberg, 2003.

[20] J. Boyland. Fractional permissions. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 270–288. Springer Berlin Heidelberg, 2013.

[21] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing. In J. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2001.

[22] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA'07, pages 441–460. ACM, 2007.

[23] N. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'10, pages 618–633. ACM, 2010.

[24] D. Clarke and S. 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*, OOPSLA'02, pages 292–310. ACM, 2002.

[25] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In J. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76. Springer Berlin Heidelberg, 2001.

[26] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: a survey. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 59–83. Springer Berlin Heidelberg, 2013.

[27] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'98, pages 48–64. ACM, 1998.

[28] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer Berlin Heidelberg, 2003.

[29] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, 1992.

[30] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI'94, pages 230–241. ACM, 1994.

[31] W. Dietl, M. Ernst, and P. Müller. Tunable static inference for generic universe types. In M. Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 333–357. Springer Berlin Heidelberg, 2011.

[32] W. Dietl and P. Müller. Universes: lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[33] W. Dietl and P. Müller. Runtime universe type inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 72–80, 2007.

[34] W. Dietl and P. Müller. Object ownership in program verification. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 289–318. Springer Berlin Heidelberg, 2013.

[35] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI'94, pages 242–256. ACM, 1994.

[36] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In D. Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 204–218. Springer Berlin Heidelberg, 2004.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[38] J. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, January 1987.

[39] D. Gordon. *Encapsulation Enforcement with Dynamic Ownership.* PhD thesis, Victoria University, Wellington, New Zealand, 2008.

[40] D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS'07)*, pages 41–52. ACM, 2007.

[41] P. Grogono and P. Chalin. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, May 1994.

[42] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *ACM Transactions on Programming Languages and Systems*, 29(6), October 2007.

[43] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In T. D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer Berlin Heidelberg, 2010.

[44] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, M. Brown, and P Wang. The Mercury language reference manual version 11.07. `http://www.mercurylang.org/information/doc-latest/reference_manual.pdf`, 2011.

[45] T. Hill, J. Noble, and J. Potter. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages and Computing*, 13(3):319–339, 2002.

[46] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE'01, pages 54–61. ACM, 2001.

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

[48] J. 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.

[49] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *ACM SIGPLAN OOPS Messenger*, 3(2):11–16, April 1992.

[50] B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM'05, pages 137–147. IEEE Computer Society, 2005.

[51] S. Kent and I. Maung. Encapsulation and aggregation. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*, pages 227–238. Prentice Hall, 1995.

[52] E. Kerfoot and S. McKeever. Maintaining invariants through object coupling mechanisms. In T. Wrigstad, editor, *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, July 2007.

[53] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice.* CRC Press Inc., 2009.

[54] M. Klebermaß. An Isabelle formalization of the universe type system. Master's thesis, Technische Universität München, Munich, Germany, April 2007.

[55] N. Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'99, pages 29–42. ACM, 1999.

[56] R. Komondoor. Precise slicing in imperative programs via term-rewriting and abstract interpretation. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 259–282. Springer Berlin Heidelberg, 2013.

[57] N. Krishnaswami and J. Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, PLDI'05, pages 96–106. ACM, 2005.

[58] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI'92, pages 235–248. ACM, 1992.

[59] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–515. Springer Berlin Heidelberg, 2004.

[60] P. Li, N. Cameron, and J. Noble. Mojojojo - more ownership for multiple owners. In *International Workshop on Foundations of Object-Oriented Languages, FOOL*, 2010.

[61] Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 202–226. Springer Berlin Heidelberg, 2007.

[62] B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1982.

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

[64] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[65] A. Milanova and J. Vitek. Static dominance inference. In J. Bishop and A. Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 211–227. Springer Berlin Heidelberg, 2011.

[66] N. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP 1996 – Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer Berlin Heidelberg, 1996.

[67] N. Mitchell. The runtime structure of object ownership. In D. Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer Berlin Heidelberg, 2006.

[68] P. Müller and A. Poetzsch-Heffter. Universes: a type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, volume 263, Hagen, Germany, 1999. Fernuniversität Hagen.

[69] P. Müller and A. Rudich. Ownership transfer in universe types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA'07, pages 461–478. ACM, 2007.

[70] A. Mycroft and J. Voigt. Notions of aliasing and ownership. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 59–83. Springer Berlin Heidelberg, 2013.

[71] S. Nägeli. Ownership in design patterns. Master's thesis, ETH Zurich, Zurich, Switzerland, March 2006.

[72] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100. ACM, 2007.

[73] J. Noble. Iterators and encapsulation. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 431–442. IEEE Computer Society, June 2000.

[74] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, July 2003.

[75] J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *Technology of Object-Oriented Languages and Systems (TOOLS 32)*, pages 176–187. IEEE Computer Society, 1999.

[76] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP 1998 – Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer Berlin Heidelberg, 1998.

[77] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala.* Artima Inc, 2nd edition, January 2011.

[78] Oracle Corporation. OpenJDK. `http://openjdk.java.net`, 2014.

[79] J. Östlund and T. Wrigstad. Multiple aggregate entry points for ownership types. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 156–180. Springer Berlin Heidelberg, 2012.

[80] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[81] R. Plasmeijer, M. van Eekelen, and J. van Groningen. Clean language report version 2.2. `http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf`, 2011.

[82] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA'06, pages 311–324. ACM, 2006.

[83] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Communications of the ACM*, 48(5):99–103, May 2005.

[84] J. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL'78, pages 39–46. ACM, 1978.

[85] A. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[86] I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599. Springer Berlin Heidelberg, 2012.

[87] J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[88] M. Sridharan, S. Chandra, J. Dolby, S. Fink, and E. Yahav. Alias analysis for object-oriented programs. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of

Lecture Notes in Computer Science, pages 196–232. Springer Berlin Heidelberg, 2013.

[89] S. Srinivasan and A. Mycroft. Kilim: isolation-typed actors for Java. In J. Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer Berlin Heidelberg, 2008.

[90] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'96, pages 32–41. ACM, 1996.

[91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 2000.

[92] TIOBE software. TIOBE programming community index for May 2014. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, May 2014.

[93] M. Utting. Reasoning about aliasing. In *Formal Aspects of Computing*, volume 3, pages 1–15, 1997.

[94] J. Vitek and B. 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.

[95] J. Vitek and B. Bokowski. Confined types in Java. *Software: Practice and Experience*, 31(6):507–532, May 2001.

[96] J. Voigt and A. Mycroft. Aliasing contracts: a dynamic approach to alias protection. Technical Report UCAM-CL-TR-836, University of Cambridge, Computer Laboratory, June 2013.

[97] J. Voigt and A. Mycroft. Dynamic alias protection with aliasing contracts. In C. Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 140–155. Springer International Publishing, 2013.

[98] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990.

[99] A. Wren. Inferring ownership. Master's thesis, Imperial College, London, UK, 2003.

[100] T. Wrigstad. *Ownership-Based Alias Management.* PhD thesis, KTH Information and Communication Technology, Stockholm, Sweden, 2006.

[101] T. Wrigstad and Clarke D. Is the world ready for ownership types? Is ownership types ready for the world? In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2011.

[102] T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for Featherweight Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA'03, pages 135–148. ACM, 2003.

[103] T. Zhao, J. Palsberg, and J. Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, January 2006.

[104] Y. Zhao and J. Boyland. A fundamental permission interpretation for ownership types. In *Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE'08, pages 65–72. IEEE Computer Society, 2008.

[105] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. Ernst. Ownership and immutability in generic Java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'10, pages 598–617. ACM, 2010.

APPENDIX A

Additional case studies of aliasing contracts

This appendix contains three additional case studies which demonstrate how aliasing contracts can be used in practice to enforce complex aliasing conditions. Here, we consider encapsulation and aliasing the binary tree data structure and in two design patterns proposed by Gamma et al.'s [37]: observer and memento.

A.1 Binary tree

A binary tree is a commonly used data structure. A UML class diagram for a standard binary tree implementation is shown in Figure A.1; the figure also shows three references which we call `A`, `B` and `C` here.

We now consider five different aliasing policies for the binary tree; for each policy, we give the required aliasing contracts for references `A`, `B` and `C`:

1. **A `TreeNode` can be read or written by itself and the `BinaryTree`.**

   We use the rw-contract "`accessor == accessed || accessor == this`" for reference `A`. We then change `TreeNode` to take a single contract parameter, `cp`. Whenever we instantiate a `TreeNode` in `BinaryTree`, we pass in an rw-contract identical to the one used for reference `A`,"`accessor == accessed || accessor == this`". `TreeNode` uses contract parameter `cp` in its specification of contracts for references `B` and `C`. Thus, all references `A`, `B` and `C` in a binary tree will at run time have the same contract with the same declaring object.

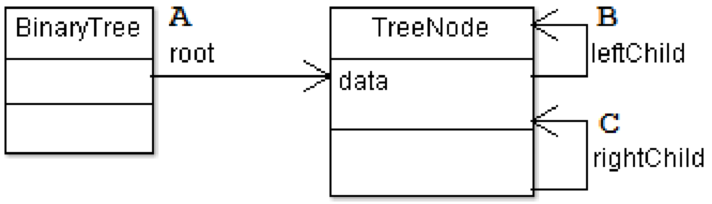


Figure A.1: UML class diagram of a standard binary tree implementation

When any `TreeNode` in the binary tree is accessed, the rw-contract "`accessor == accessed || accessor == this`" is evaluated in the context of `BinaryTree`, evaluating to `true` if the access comes from the `TreeNode` itself ("`accessed`") or from the `BinaryTree` ("`this`").

2. **A `TreeNode` can be read or written by the `BinaryTree` and all other `TreeNodes` in the tree.**

   First, we create an encapsulation group in `BinaryTree` (called `allNodes`) which contains all of the `TreeNodes` in the tree:

   ```
   //In BinaryTree
   group allNodes = {root, root.descendants};
   //In TreeNode
   group descendants = {left, left.descendants, right,
       right.descendants};
   ```

   For reference `A`, we declare the rw-contract "`accessor == this || accessor in allNodes`". We can again use contract parameters for references `B` and `C` as before. Equivalently, we can use the `canread` and `canwrite` operators in the contracts for references `B` and `C`: "`accessor canread this, accessor canwrite this`". This contract causes contract evaluation to move up the tree from a node to its parent (the contract's declaring object "`this`"); finally, this causes the evaluation of the contract on reference `A`. This contract will succeed if the access comes from the `BinaryTree` ("`this`") or any node in the tree (which is in group `allNodes`).

3. **A `TreeNode` can be read or written by itself, the `BinaryTree` and any of its ancestor nodes.**

   To enforce this aliasing policy, we use the rw-contract "`accessor == accessed || accessor == this`" for reference `A`. For references `B` and `C`, we declare the contract "`accessor == accessed || accessor canread this, accessor == accessed || accessor canwrite this`". This contract first checks if the node was accessed by itself ("`accessed`"). Otherwise, contract evaluation continues up the tree to the contract's declaring object (the tree node's parent). Each ancestor checks if the access comes from itself ("`accessor == accessed`"); thus the contract evaluates to `true` if the access comes from one of the ancestors. Finally, the contract on reference `A` will be evaluated; this contract succeeds if the access originated in the root node of the tree ("`accessed`") or the binary tree itself ("`this`").

4. **A `TreeNode` can be read or written by itself and any of its descendant nodes.**

   Implementing this is straightforward using the encapsulation group `descendants` declared above. We give reference `A` the rw-contract "`accessor == accessed || accessor in root.descendants`"; we then use the rw-contracts "`accessor == accessed || accessor in left.descendants`" and "`accessor == accessed || accessor in right.descendants`" for references `B` and `C` respectively.

5. **A `TreeNode` can be read or written by itself and any nodes that precede it in an in-order traversal of the binary tree.**

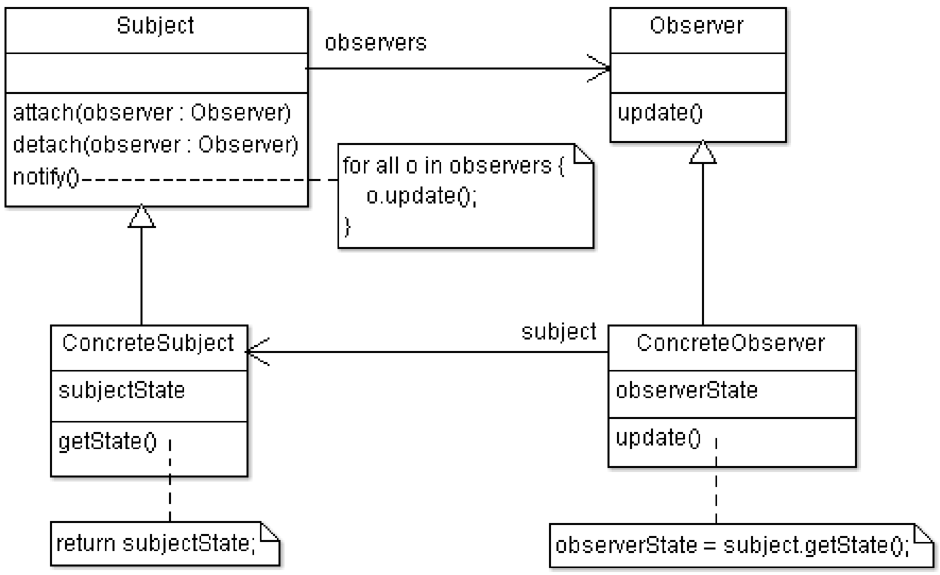


Figure A.2: UML class diagram of the observer pattern

This aliasing condition requires different contracts for references `B` and `C` in `TreeNode`. A node's left child should be accessible to itself (“`accessed`”) and its *left descendants*; that is, all descendants of its left child. A node's right child should be accessible to itself (“`accessed`”), the parent node (“`this`”), the parent node's left descendants, and its own left descendants. The `root` node in `BinaryTree` should be accessible to itself (“`accessed`”) and its left descendants.

To simplify the definition of these contracts, we define an encapsulation group called `leftDescendants` in `TreeNode` which contains all of a node's left descendants:

```
//In TreeNode
group descendants = {left, left.descendants, right,
    right.descendants};
group leftDescendants = {left, left.descendants};
```

We can then define the rw-contract “`accessor == accessed || accessor in root.leftDescendants`” for reference `A`. For reference `B`, we use the rw-contract “`accessor == accessed || accessor in left.leftDescendants`”. Finally, for reference `C`, we use rw-contract “`accessor == accessed || accessor == this || accessor in right.leftDescendants || accessor in leftDescendants`”.

A.2 Observer

Observer is a behavioural pattern proposed by Gamma et al. [37] which allows one or more objects (called the observers) to observe the state of another object (called the subject). When the subject's state changes it notifies all of its observers. In the standard version of the pattern, the observers then call the subject to get its updated state and take appropriate action. Alternatively, the subject can pass any relevant state to the observers when it notifies them. Figure A.2 shows a UML class diagram of the observer pattern.

The observer pattern contains four classes:

- `Subject` is an abstract class representing the observed subject. It can register observers (`attach`), de-register observers (`detach`) and notify all current observers (`notify`).

- `ConcreteSubject` is the concrete subject implementation which contains the subject's actual state. When the subject's state changes it calls the `notify` method to notify its observers. `ConcreteSubject` also includes a `getState` method which observers can call to get a reference to the subject's current state.

- `Observer` is an abstract class or interface representing the observer. It contains a method called `update` which is called by the subject's `notify` method.

- `ConcreteObserver` is the concrete observer implementation. It contains an implementation of `update` which gets the subject's state (by calling `getState`) and takes appropriate action.

The observer pattern leads to loose coupling between the subject and the observer and allows one subject to be observed by any number of observers.

The encapsulation weakness of the pattern arises from the `getState` method of the subject which returns a reference to the subject's state for use by the observers. This means that the subject's state needs to be shared between the subject and the observers (although arguably observers should be able to read but not write the subject's state). However, the `getState` method also leaks a reference to the state to the rest of the system; it may be called by objects other than the observers, compromising encapsulation.

Using aliasing contracts, we can ensure that only observers and the subject can access the subject's state. Other parts of the system may still call the `getState` method but will be unable to use the reference they receive. We can further enforce the constraint that observers should be able to read but not write the subject's state.

To achieve this, we annotate the `state` field in `ConcreteSubject` with the following contract:

```
{accessor == accessed || accessor == this ||
    observers.contains(accessor),
 accessor == accessed || accessor == this}
```

This contract specifies that the state can be read by itself ("`accessed`"), the subject ("`this`") and any of the observers, while it can be written by itself and the subject only. It encapsulates the state inside the subject, while allowing limited sharing with the observers.

A.3 Memento

Memento is a behavioural pattern proposed by Gamma et al. [37] which allows an object (called the originator) to store historical records (called mementos) of its state and to later restore that state. Figure A.3 shows a UML class diagram of the memento pattern.

The memento pattern contains four main classes:

- `Memento`: The memento stores the state of the originator and protects this state from accesses by anyone except the originator which created the memento.

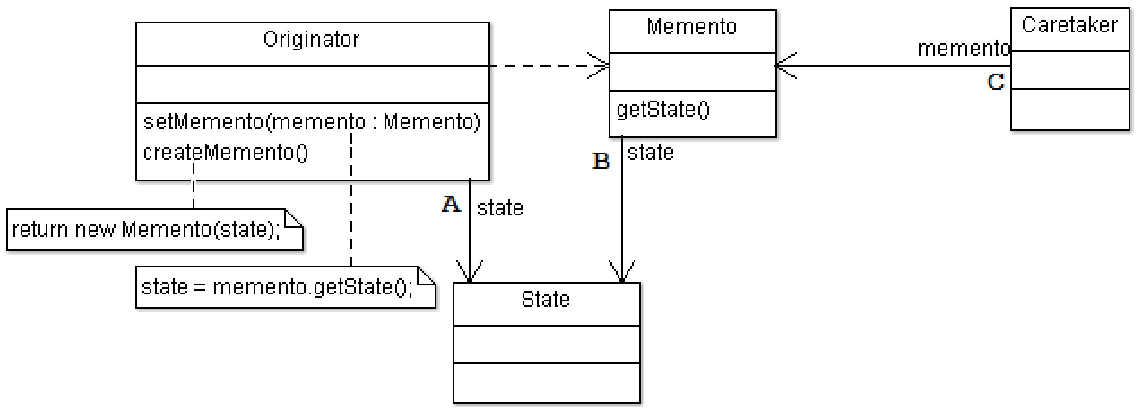

Originator
setMemento(memento : Memento)
createMemento()
return new Memento(state);
state = memento.getState();
Memento
getState()
Caretaker
memento
C
B
state
A
state
State

Figure A.3: UML class diagram of the memento pattern

- `Originator`: The originator creates mementos by taking snapshots of its state; it can later restore its state from a memento.
- `Caretaker`: The caretaker stores mementos but never looks at them. It is thus not aware of their contents (that is, the state they store).
- `State`: This class represents the originator's state. It does not appear in Gamma et al.'s original description of the memento pattern [37], but we include it here for clarity.

The advantage of using the memento pattern is that we do not need to add accessor and mutator methods to the originator to get and restore its state; this improves the originator's encapsulation. Instead, the originator itself creates mementos (passing in its state). Using memento also simplifies the originator, as it does not need to take care of its own mementos; the mementos are kept safe by the caretaker who does not allow anyone, except for the originator, to access them.

On the other hand, the memento pattern can be expensive if the originator contains a lot of state. When creating a memento, the originator must *copy* its state to ensure that the memento state and the originator state do not point to the same object; otherwise, any subsequent state changes in the originator would also change the memento's state. This is a case where unintended aliasing leads to incorrect behaviour.

Gamma et al. suggest that, to ensure that the caretaker cannot look at the mementos it holds, the `Memento` class should define a narrow interface with few methods for the caretaker, while providing a wider interface (with a method to get the state) for the originator, allowing it to access the memento's state during the restore operation. However, defining two different interfaces in this way can be difficult depending on the programming language we use.

Aliasing contracts can help address some of the difficulties with the memento pattern. We can use them to ensure that the memento's state and the originator's state are separate and do not point to the same object. In addition, we can use contracts to enforce the constraint that mementos can be accessed only by the originator which created them. To achieve this, we use the following contracts for references `A`, `B` and `C` in Figure A.3:

- Reference `A` - "`accessor == accessed || accessor == this`" - This rw-contract protects the internal state of the originator. We assume here that the state is fully encapsulated in the originator (that is, it can only be read and written by itself and the originator), but depending on the exact semantics of the originator, this contract may vary.

- Reference `B` - “`false`” - This rw-contract ensures that state cannot be read or written while it is stored in the memento.
- Reference `C` - “`accessor == originator, false`” - This contract protects the memento from being accessed by any object other than the originator which created it. It also ensures that the memento cannot be written (for example giving it a new state) after it was created. Note that this contract requires the caretaker to have knowledge of which object created which memento. However, this is a reasonable requirement, because otherwise the caretaker cannot possibly ensure that it gives the memento only to the correct originator.

With these contracts in place, the memento pattern now proceeds as follows:

1. The originator creates the memento, copying its state. The contract of reference `B` now protects this state. Therefore, if the originator has by mistake forgotten to copy its state and instead passed it to the memento by reference (so that references `A` and `B` now point to the same state object), any subsequent reads or writes to the originator state would cause a contract violation of the contract of reference `B`. This contract ensures that the state, once given to the memento, cannot be read or written at all.
2. The originator gives the memento to the caretaker. The contract of reference `C` now ensures that the memento cannot be read by anyone except the originator that created it and cannot be written at all.
3. When the originator gets the memento back from the caretaker (the contract reference `C` still applies), it can call `getState`. The originator now has two options. It can *copy* the state before storing it; the copy of the state will no longer be subject to the contract of reference `B`, which previously prevented all reads and writes of the state. Alternatively, the originator can store a reference to the state and destroy the memento, thus removing the restricting contract.

From the description above, we can see that aliasing contracts are very useful for enforcing encapsulation in the memento pattern. They ensure that the memento's and originator's states do not point to the same object. They can also enforce the constraint that each memento should only be accessible to the originator that created it. Finally, aliasing contracts make the memento and its state immutable, thus ensuring that it remains unchanged between the memento's creation and the state restoration.

APPENDIX B

Using aliasing contracts to model Clarke-style context parameters

Clarke-style ownership types include *context parameters* which enable ownership polymorphism. In Section 4.10.2.4, we showed how Clarke-style ownership types can be modelled using aliasing contracts. We simply modelled context parameters using monomorphisation. In this appendix, we show how to model them directly using contract parameters.

We explain how this is done using an example, taken directly from from Clarke et al.'s paper [27] (but shortened slightly). The code for the example with ownership types is shown below; a visual overview of the resulting ownership structure is shown in Figure B.1:

```
class Pair<m, n> {
  m X fst;
  n Y snd;
}

class Intermediate {
  rep Pair<rep, norep> pair1;
  norep Pair<rep, norep> pair2;
}

class Main {
  norep Intermediate safe;
}
```

We now translate this code to a program with aliasing contracts as shown below; this new version uses aliasing contracts to give the same encapsulation guarantees as the original program with ownership types and produces the same ownership structure as shown in Figure B.1:

```
class Pair<m, n> {                                            //A
  X fst {<m>};                                                //B
  Y snd {<n>};

  group mGroup = {fst, fst.repGroup};                         //C
  group nGroup = {snd, snd.repGroup};                         //D
}
```

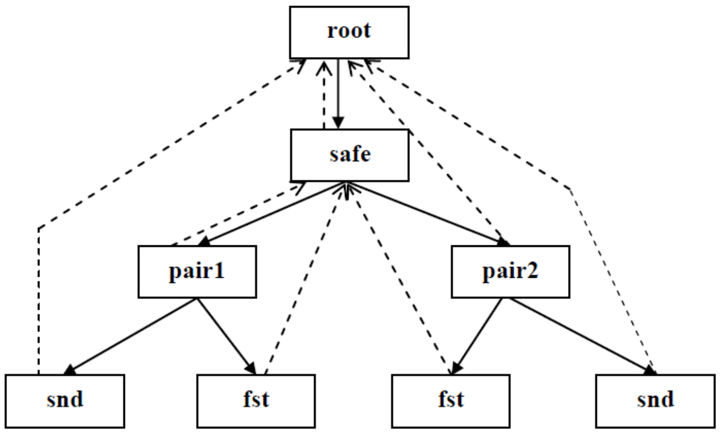


Figure B.1: The ownership structure of the `Pair` example

```
class Intermediate {
  Pair pair1 {accessor == this || accessor in repGroup};
  Pair pair2 {true};

  group repGroup = {pair1, pair1.repGroup, pair1.mGroup,     //E
      pair2.mGroup};

  Intermediate() {
    pair1 = new Pair<(accessor == this || accessor in repGroup),
      (true)>();                                             //F
    pair2 = new Pair<(accessor == this || accessor in repGroup),
      (true)>();                                             //G
  }
}

class Main {
  Intermediate safe {true};
}
```

From the example, we can see that the following steps are required to model context parameters with aliasing contracts:

- We use contract parameters to model context parameters. We replace `rep` context parameters with the rw-contract "`accessor == this || accessor in repGroup`" and `norep` context parameters with the rw-contract "`true`". This is shown at program points `F` and `G` in the above example.
- If a class `C` is declared to take context parameters, we translate it to a class which takes the same number of contract parameters. In the example above, we translate class `Pair` in the original program (which declares two context parameters `m` and `n`) to a class `Pair` with two contract parameters `m` and `n` (see program point `A` above).

  We find all fields of class `C` which are annotated with context parameters in the original program and give them a parameterised contract with the equivalent contract parameter. For example, in the original program, field `fst` in class `Pair` is annotated as `m`; we translate this to a field with rw-contract "`<m>`" at program point `B`.

Finally, we create an encapsulation group `cpGroup` for each contract parameter `cp`. The aim of this encapsulation group is to contain all owned objects in case the parameter is instantiated as `rep`. For each field of class `C` annotated with parameter `cp`, we add the field and the field's `repGroup` to `cpGroup`.

In the example above, we create groups `mGroup` and `nGroup` for contract parameters `m` and `n` respectively as shown at point `C` and `D` above; `mGroup`, for example, then contains `fst` and `fst.repGroup` (since `fst` is annotated with parameter `m`).

- If a class `D` creates an object of class `C`, giving it a `rep` context parameter, we add the group associated with the parameter in class `C` to `D`'s `repGroup`. In this way, any fields annotated with the `rep` parameter in `C` will be added to `D`'s `repGroup`, becoming directly owned by the instance of `D`.

  In the example program above, class `Intermediate` creates two `Pair` objects (stored in `pair1` and `pair2`). In both cases, it uses the two context parameters `rep` and `norep`; parameter `m` is instantiated as `rep` for both `Pair` objects. Therefore, we add `pair1.mGroup` and `pair2.mGroup` to `repGroup` in `Intermediate` at program point `E`. In this way, the `Intermediate` object will directly own the objects stored in fields annotated with parameter `m` in `pair1` and `pair2`.

- No action is required when a class uses a `norep` parameter when creating an object; this does not require modification of the ownership structure as `norep` objects are not owned by another object (only by the root of the system). In our above example, we do not add `pair1.nGroup` and `pair2.nGroup` to `repGroup` in `Intermediate`, as the `n` parameters of `pair1` and `pair2` are both instantiated as `norep`.

- If class `C` passes on a context parameter `cp` to an object `e` of another class `E` with contract parameter `cp2`, we add the `cp2Group` encapsulation group in `e` to `C`'s `cpGroup` definition. For example:

  ```
  class C<cp> {
    group cpGroup = {e.cp2Group, ...};
    ...
    E e = new E<cp>();
  }
  class E<cp2> {
    group cp2Group = {...};
  }
  ```

  This ensures that `cpGroup` in `C` contains all objects whose contracts use the parameter `cp`, both in class `C` and any other classes to which the parameter is passed.

Our definition of encapsulation groups is somewhat more complex than the simple use of context parameters in Clarke-style ownership types. The reason for this is that ownership types combine the definition of the program's ownership structure with the definition of encapsulation policies, while the aliasing contracts formulation describes the two aspects separately. This gives our system more flexibility but also adds complexity.