Dynamic Alias Protection with Aliasing Contracts

Janina Voigt and Alan Mycroft

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

Abstract. Object-oriented languages allow any object to point to any other object, limited only by type. The resultant possible aliasing makes programs hard to verify and maintain.

Much research has been done on *alias protection schemes* to restrict aliasing. However, existing schemes are either informal (design-pattern-like) or static type-like systems. The former are hard to verify, while the latter tend to be inflexible (e.g. shared ownership is problematic).

We introduce *aliasing contracts*: a novel, dynamic alias protection scheme which is highly flexible. We present JaCon, a prototype implementation of aliasing contracts for Java, and use it to quantify their runtime performance overheads. Results show that aliasing contracts perform comparably to existing debugging tools, demonstrating practical feasibility.

1 Introduction

In typical object-oriented (OO) programming languages, object variables do not contain objects directly, but references to (addresses of) other objects. Multiple variables can contain the same address and thus point to the same object at the same time; this is known as *aliasing*.

Aliasing is an important feature of OO because it allows sharing of objects between different parts of a program; this is essential for the efficient implementation of important programming idioms, including iterators over collections. However, aliasing reduces modularity and encapsulation; an aliased object can be accessed and modified through any of the references pointing to it, without the knowledge of the others; this can create bugs which are difficult to trace.

Many modern OO programming languages provide *access modifiers* such as `private`, `protected` and `public`. These modifiers limit the scope of the variable, but do not protect the *object* to which the variable points.

For example, consider the program in Listing 1. Class `Person` provides a getter method `getName` for its `myName` field, which returns a reference to the object in `myName`. Although `myName` is declared to be `private`, any client can call `getName`, obtain an alias for the object in `myName` and modify it without the knowledge of the `Person`, for example by calling `setLastName`.

C.-c. Shan (Ed.): APLAS 2013, LNCS 8301, pp. 140–155, 2013.

```
public class Person {
  private Name myName;
  public Name getName() {
    return myName;
  }
}
```

```
Person p = new Person();
Name n = p.getName();
n.setLastName("Smith");
```

Listing 1. Access modifiers protect only the variable, not the object

A number of *alias protection schemes* have been proposed to avoid situations like this and protect the *object* rather than just the variable from unwanted accesses. However, they tend to be too inflexible and restrictive in practice and have not yet been widely adopted by the programming community.

The contributions of this paper are two-fold: firstly, in Section 3 we introduce a novel, dynamic alias protection scheme called *aliasing contracts* which aims to be flexible and expressive, while at the same time remaining conceptually simple. Secondly, in Section 4 we present JaCon, a prototype implementation of aliasing contracts in Java; measurements of the runtime overhead caused by the dynamic checks of aliasing contracts demonstrate their practical feasibility.

The name *contract* comes from work on software contracts [11] which allow the specification of preconditions and postconditions for methods. Aliasing contracts allow developers to annotate a variable with assumptions about which parts of the system should be able to access the object to which the variable points; these assumptions are checked at runtime. In this way, aliasing contracts prevent the use of unintentionally leaked references, as the reference in the example above, making them particularly valuable during the testing phase of a project.

Aliasing contracts also provide a unifying model of alias protection; aliasing contracts can model existing alias protection schemes, giving us a framework for expressing and comparing many different aliasing policies.

Our dynamic approach to alias control has similar advantages and disadvantages as dynamic type checking. It is more flexible and can cover conditions which cannot be checked statically; for example, static alias protection schemes struggle with design patterns like iterators and observers, which require sharing of objects, but they can easily be implemented with aliasing contracts. Dynamic schemes also tend to be conceptually simpler; static schemes often require more artifacts to compensate for the restrictions os static checking. On the other hand, the runtime checks required by dynamic schemes cause performance overheads. In addition, problems will only be discovered when (and if) the affected code is executed.

The timing of validity checks is also different in static and dynamic checking. Like dynamic type checking, we check the validity of an object access directly before the access is made at runtime. Static alias protection schemes instead restrict assignments so that references cannot be leaked in the first place.

2 Background

The literature on aliasing and its control is huge; see [5] for a detailed description. Here, we summarise the main strands of research for completeness.

The overall idea of alias control is to construct software engineering "design patterns"—or more formal programming language structures—to discipline programmer use of aliasing; a key concept is *encapsulation* whereby usage of some objects ("rep" objects) is restricted to certain other objects (the "owners"). Early conceptual designs include Hogg's islands [9] and Almeida's balloons [1]. They implement *full encapsulation*: each object is either encapsulated within another object or shared; any object reachable through an encapsulated object is also encapsulated. To increase flexibility (though at the cost of soundness) Hogg and Almeida restrict only pointers from fields of other objects ("static aliasing") but allow pointers from local variables and method parameters ("dynamic aliasing")—the latter being more transient and easier to track.

Clarke-style ownership types [6] added significantly to the subject area by showing a type-like system could capture aliasing restrictions (later known as *owners-as-dominators*) and that these could be checked statically.

Clarke-style ownership types require each object to have a *single* owner and also do not allow ownership of an object to be transferred at runtime; this makes them too inflexible to deal with common idioms such as iterators. Follow-up work partially addressed these shortcomings, introducing multiple ownership types [10], ownership with inner classes [2], gradual ownership types [15], and Universe types (*owners-as-modifiers*) [12].

There has also been some work on dynamic ownership. Gordon et al. [7] propose a system where ownership information is checked at runtime. Like our dynamic aliasing contracts, dynamic ownership types do not directly restrict aliasing itself, but allow any references to exist; instead, they limit how these references can be used. Gorden-style dynamic ownership types differ from our work since they support only one particular aliasing policy (owners-as-dominators), while aliasing contracts support many different ones.

Another approach to alias protection is that of capabilities [4, 8] and permissions [3,19]. Capabilities and permissions associate access rights with each object reference, specifying whether the reference is allowed to, for example, read, write or check the identity of an object. This can be used to model various aliasing conditions, such as uniqueness and borrowing.

Throughout the vast literature on alias protection schemes, there is no unifying framework which can be used to embed and compare them. Boyland et al.'s [4] capabilities do this to some extent, but at a relatively low level where there is a large semantic gap to be bridged between them and high-level constructs like owners-as-dominators. Aliasing contracts provide a unifying, language-level framework which can be used to express and compare existing alias protection schemes.

3 Aliasing Contracts

This work proposes aliasing contracts which express and enforce restrictions about the circumstances under which an object can be accessed. Here, we give a basic overview of aliasing contracts; [18] presents aliasing contracts in more detail and proposes a syntax operational semantics. More detail also appears in the first author's forthcoming PhD thesis.

An aliasing contract consists of two boolean expressions, e^{r} and e^{w}, attached to a variable declaration. (Note that in this paper we use the term *variable* to refer to fields, local variables and method parameters.) An access to an object is permitted only if the contracts of *all* variables currently pointing to the object are satisfied; contracts thus essentially specify dynamically determined, conjunctive *preconditions* for object accesses. The expression e^{r} specifies preconditions for *read* accesses, while e^{w} concerns write accesses. Where the read and write contract expressions are the same, one can be omitted for convenience; we call such a contract a *rw-contract* (read-write-contract).

The distinction between read and write accesses requires a similar distinction between *pure* and *impure* methods: pure methods do not modify any state and may be called with read access permissions, while impure methods require both read and write access permissions.

For each contract, we call the nearest enclosing object the contract's *declaring* object. When the contract is evaluated, evaluation takes place in the context of its declaring object; that is `this` points to the declaring object. We can regard a contract as a `boolean` method of the class which declares it (and this is how we implement it in Section 4).

A contract may be any valid, *side-effect-free* boolean condition. Contracts have access to two special variables, in addition to `this`: `accessed` points to the object being accessed and `accessor` points to the object whose method is making the access. The value of `accessor` is determined immediately prior to contract evaluation; for a single contract, `accessor` varies between evaluations. Thus, an alternative view of a contract is a method which takes an object parameter (`accessor`) and returns a boolean value.

Listing 1 gave a program which leaks a reference from the `private` variable `myName`, thus making the object accessible and modifiable by any client. To address this problem, we instead annotate `myName` with the rw-contract "`accessor == this || accessor == accessed`" to enforce encapsulation:

```
Name myName {accessor == this || accessor == accessed};
```

This contract signifies that only the enclosing `Person` object and the `Name` object itself will be able to access the object in `myName`. The contract is evaluated in the context of the declaring object, the enclosing `Person` object; it will only evaluate to `true` if `accessor` is equal to `this` or if `accessor` and `accessed` are the same object; that is, if the access comes from the `Person` or from the `Name` itself. If a client now obtains a reference to the object in `myName` by calling `getName`, it will not be allowed to use it to read or write the object.

Alternatively, we could loosen the contract slightly to "`true, accessor == this || accessor == accessed`". This contract corresponds to an *owners-as-modifiers* approach [12]; it would allow any client to read the `Name` object (the read contract is "`true`") but would continue to prohibit write accesses.

Aliasing contracts are very flexible since their evaluation depends on the current state and aliasing structure of the program. If a client in the example above obtains a reference to a `Name` object by calling `getName` in a `Person` object, it cannot initially access it due to the contract on `myName`. Aliasing contracts do not restrict aliasing itself, but object accesses: obtaining the reference in this situation is legal but using it is not. If the `myName` field in the `Person` object is then pointed to a new `Name` object, the `Name` object referenced by the client will become accessible; the `myName` field's contract no longer applies to it.

Since aliasing contracts depend on the dynamic aliasing structure of a program at the time an access to an object is made, they cannot in general be verified statically. Instead, they need to be checked at runtime; when a contract violation is detected, an error is reported (cf. static and dynamic type checking).

Contract checks need to be performed for each object access, including field accesses, field updates and method calls. Reads and writes to local variables and parameters do not trigger contract evaluations; they represent accesses and modifications to the unaliased *stack*. Similarly, constructor calls do not change existing heap objects of themselves and thus do not require contract checks.

For each object access, we first retrieve all contracts which currently apply to the accessed object and evaluate them. Thus we track, for each object, which variables currently point to it. The conjunctive nature of contract evaluation means that if any contract evaluates to `false`, the entire evaluation fails.

Note that if there are multiple contracts to evaluate, the order in which they are evaluated is irrelevant. Expressions in contracts may not have side-effects; this means that a contract can change neither the state of the program nor its aliasing structure and therefore cannot affect other contract evaluations.

Although similar to assertions in spirit, aliasing contracts are significantly more expressive; the contracts which are evaluated depend on the aliasing structure of the program. An implementation must track references to determine exactly which contracts apply to an object. Alising contracts also have advanced features (briefly discussed in Section 6) which allow the expression of complex conditions that cannot be described using standard Java boolean expressions.

Extensions for Real-life OO Languages. Our theory of aliasing contracts is clean and simple—every object access causes a contract check—but real programming languages have more complex features that we need to address.

In particular, `static` methods do not fit well with our object-based approach. In 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 to consider. 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`" or "`false`") behave as expected.

Many modern OO languages include variables of primitive types, which store values instead of references to objects; values cannot be aliased and therefore accessessing them does not require contract checks.

Other language features, on the other hand, do not require special treatment due to the dynamic nature of aliasing contracts. Inheritance, for example, fits naturally and objects of inner classes can be treated just like other objects. 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.

4 JaCon: Practical Aliasing Contracts for Java

We have implemented a prototype, JaCon, which supports the definition of aliasing contracts in Java programs and performs contract evaluations at runtime. The prototype consists of a modified Java compiler and a runtime library (which we call the *contract library* below). The compiler injects calls to the contract library into the source code, allowing it to monitor contracts at runtime. The code for our contract library is avaliable at `www.cl.cam.ac.uk/~jv323/contractlibrary`.

We chose Java as our base programming language since it is the most popular object-oriented programming language [17] 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 and `break` and `continue` statements, complicates the tracking of contracts.

For our prototype implementation, we modified the compiler *javac* of the OpenJDK 6 [14] to inject calls to the contract library.

Contracts are parsed and converted into anonymous inner classes extending our abstract `Contract` class; one such `Contract` class is created for each syntactically distinct contract expression in a class. Each of these `Contract` classes overrides the method `checkContract`, which can be called by the contract library to evaluate the contract. The `checkContract` method takes two parameters, `accessor` and `accessed`, both of type `Object`; the contract expression becomes the method's return statement. For example, the contract "`accessor == this || accessor == accessed`" of the `myName` field of `Person` from our example in Section 3 is transformed into

```
public Contract _contractPerson42 = new Contract() {
  public boolean checkContract(Object accessor, Object accessed) {
    return accessor == Person.this || accessor == accessed;
  }
};
```

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 enclosing object; in the example above, `this` becomes `Person.this`.

The contract library tracks the `Contracts` which apply to each object and invokes their `checkContract` methods when they need to be evaluated.

Contracts are registered and de-registered when an assignment occurs; `myName = newName` points `myName` away from the object it currently points to and to the object currently also pointed to by `newName`. This change of aliasing requires modification of the set of contracts associated with these two objects; to this end, JaCon inserts two calls to the contract library, one to de-register the contract of variable `myName` before the assignment and one to register the contract after the assignment. The assignment `myName = newName` becomes

```
ContractLibrary.removeContract(myName, _contractPerson42);
myName = newName;
ContractLibrary.addContract(myName, _contractPerson42);
```

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; after this the variables are no longer available and their contracts should not persist. Exceptions are problematic because an exception causes execution to leave a block prematurely. We therefore wrap each block in a `try-finally` statement and remove local variable contracts in the `finally` block to ensure correct contract de-registration even when an exception is thrown.

Contract de-registration also takes place when an object is garbage collected; at this point, finalisation causes all of its `Contracts` to be deallocated and thus removed from the objects to which they applied. We discuss the implications of garbage collection in more detail in Section 6.

Registration and de-registration as explained above allow 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 of references for each object, which in itself is potentially useful; it means that JaCon could also be used to investigate the topology of the heap, independent of aliasing contracts.

Calls to the contract library to check contracts are inserted before any accesses and updates to fields. As explained above, accesses and updates to local variables do not require contract checks. For example, the assignment `x.f = y.g` contains a write access to `x` and a read access to `y`; it becomes

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

Methods may be declared `pure` or `impure`; if no method purity is given, JaCon automatically determines its purity. Calls to `pure` methods require read contract checks to be performed, while calls to `impure` methods need to trigger both read and write contract checks. Appropriate contract checks are inserted at the entry to the method bodies (rather than in the caller).

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 of the objects; calls to notify the contract library of context changes are inserted around each method body:

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

Our contract library implementation tracks and evaluates contracts correctly in the presence of concurrency. For example, accesses to the contract stores are synchronised and separate call stacks are maintained for each thread.

Optimisations. A naive implementation of aliasing contracts, as described above, performs many unnecessary contract checks; we have implemented optimisations to avoid such checks and improve the performance of JaCon.

A common contract is "`true`", meaning that there are no restrictions on object access. Since this contract obviously always evaluates to `true`, there is no need to store or evaluate it.

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 variables. Such contracts, for example, include the contracts "`true`" and "`false`". They need to be evaluated only *once*.
- Contracts whose result changes for different `accessor` objects but which are not affected by changes to fields. This includes the contract "`accessor == accessed`". These contracts need to be evaluated only *once for each distinct* `accessor` object. The contract "`accessor == this`" also falls into this category, as the value of `this` never changes for a given object.
- Contracts which depend on values of fields or call methods, for example "`accessor == this.f`" or "`accessor == getFoo()`". These contracts need to be evaluated every time an access is made, since the value of fields and return value of methods may have changed since the previous evaluation.

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

5 Performance Evaluation

One of the main problems with aliasing contracts is the runtime performance overhead they cause; each assignment causes contracts to be added and removed, every object access triggers a contract check.

```
class SimpleExample {
  public void run()
    Bar b = new Bar();
    Foo[] foos = new Foo[NUM_OBJ];
    for(int i = 0; i < NUM_OBJ; i++){
      Foo f = new Foo();
      foos[i] = f;
      f.bar = b;
      b.num = i; // †[here]
    }
  }
}
```

```
class Foo {
  public Bar bar {<contract>};
}

class Bar {
  // Primitive types do not
  // have contracts.
  public int num;
}
```

Listing 2. A simple program measuring object access time at †[here]

Using JaCon, we try to quantify this performance overhead. First, we investigate how the number of contracts for an object impacts the time taken to perform a single object access with contract checks. We also estimate the performance of real-world software using four open-source programs.

Performance measurements were made on a Windows laptop with 8GB of RAM and a 2.5GHz Intel Core i5 processor. All values stated below are averages of at least five separate measurements.

Performance for a Single Object Access. Whenever an object access is made, all contracts associated with the object must be checked. The time required for the object access thus increases with the number of contracts, assuming that all contracts need to be evaluated. If we assume that each contract is a simple boolean condition which can be evaluated in constant time, contract checking time increases linearly with the number of contracts of the object.

To measure object-access time, we construct a simple test program, shown in Listing 2. The program executes a loop, adding one reference (and hence contract) to an object `b` of type `Bar` per iteration. The assignment `b.num = i` performs a write access to the `Bar` object in `b`, causing all contracts associated with it to be checked. By measuring the time taken for this access on every iteration, we can measure how the object access time varies with the number of contracts.

We also vary the expressions of the contracts (marked as `<contract>` in Listing 2) to see how different contract expressions influence object access time.

Table 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, highlighting the huge difference in performance they cause.

The contracts `alwaysTrue()` and `alwaysFalse()` call a method which always returns `true` or `false` respectively. JaCon cannot optimise evaluation, since the contracts involve a method call; they needs to be re-evaluated for each object access. For the contract `alwaysTrue()`, contract checking time thus increases

Table 1. Time in milliseconds per object access for varying number n of contracts associated with the accessed object and varying contract expressions

n	`alwaysTrue()` (always succeeds)	`alwaysFalse()` (always fails)	`accessor==this||accessor==accessed` (always succeeds in this example)
0	0	0	0
5,000	0.84	0.0099	0.011
15,000	3.93	0.0038	0.0032
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

with the number of contracts, adding around two milliseconds for every 10,000 contracts. However, for the contract `alwaysFalse()` the very first contract evaluates to `false`, making it unnecessary to check the remaining contracts. Thus, the time required for each object access is very low and does not change as the number of contracts for the object increases.

The contract `accessor == this || accessor == accessed` always evaluates to `true` in the example given (but not in the general case). The contract depends on the value of `accessor` but is not affected by changes to fields; thus, it needs to be evaluated only once per `accessor`. Since `accessor` is always the same (the `SimpleExample` object running the loop), each contract needs to be evaluated exactly once; this means that for every iteration, only one contract is evaluated—the newly added contract. Time taken to access the object therefore matches the `alwaysFalse()` case and does not change as the number of contracts for the object increases.

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 is not optimised. Nevertheless, contract evaluation continues to appear feasible in this case, as long as the number of contracts remains low; we believe it is unlikely to have more than 10,000 references pointing to the same object at once, even in a large program.

We further suggest that 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, including "`true`", "`false`", "`accessor == this`", "`accessor == accessed`" and "`accessor instanceof Foo`" can be optimised by JaCon as outlined above and only need to be evaluated either once or once for each distinct `accessor`. This significantly cuts the number of required contract evaluations, making evaluation efficient even when many contracts are associated with a single object.

Case Studies. To study the performance of real-world software using aliasing contracts, we selected four open-source programs written in Java: JGraphT (`http://jgrapht.org`), JUnit (`http://junit.org/`), NekoHTML (`http://nekohtml.sourceforge.net`) and Trove (`http://trove.starlight-systems.`

Table 2. Version, size measurements and number of test cases of the test programs

Program	Version	Date	Source Files	Classes	Lines of Code	Test Cases
JGraphT	0.8.3	19/01/2012	188	270	34,266	152
JUnit	4.10	29/09/2011	168	281	13,276	524
NekoHTML	1.9.18	27/02/2013	32	60	13,262	222
Trove	3.0.3	15/02/2013	697	1,603	240,555	548

Table 3. Compilation Performance

Program	Compilation time (javac0)	Bytecode size (javac0)	Compilation time (javac1)	Bytecode size (javac1)
JGraphT	4.3 s	667 kB	7.3 s	1,182 kB
JUnit	2.3 s	524 kB	4.2 s	886 kB
NekoHTML	17.5 s	264 kB	29.7 s	444 kB
Trove	18.8 s	5,153 kB	31.2 s	8,797 kB

Table 4. Runtime Performance

Program	Time (javac0)	Time (javac1)	Ratio (javac1)	Time (ref tracking only)	Ratio (ref tracking only)
JGraphT	4.7 s	99.2 s	20.8	53.4 s	11.2
JUnit	13.1 s	21.0 s	1.6	16.4 s	1.4
NekoHTML	1.9 s	3.5 s	1.8	2.6 s	1.4
Trove	5.6 s	62.5 s	11.3	21.2 s	3.7

`com`). Table 2 shows version, size measurements and number of test cases for these programs.

We selected four programs from different domains. JGraphT is a graph library which implements various graph data structures and associated graph algorithms. It involves complex data structures likely to lead to interesting aliasing properties and runs algorithms with high asymptotic complexities; this means that its performance with aliasing contracts is likely to be particularly bad.

The Trove library provides high performance collections for Java. Again, the data structures it builds are likely to lead to interesting aliasing properties.

NekoHTML is an HTML scanner and tag balancer. As parsing involves a lot of comparatively slow input and output, we expect the performance of NekoHTML to be less strongly influenced by the presence of aliasing contracts.

JUnit is a well-known program to support unit testing for Java. It does not involve large data structures and complex algorithms and we therefore expect JUnit's performance to degrade only slightly when using aliasing contracts.

All four programs have been updated in the last two years. They include extensive unit test suites, making them suitable for performance evaluation.

Our test programs include thousands of lines of code, making it impossible for us to manually annotate them with contracts. Instead, we use default contracts for all variables. To get realistic results, we selected the contracts which we believe would be most common in practice: "`true`" for local variables and

method parameters, as well as `public` fields, and "`true, accessor == this || accessor == accessed`" 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 be readable but not be modifiable from the outside. This corresponds to an owners-as-modifiers [12] approach.

These default contracts caused relatively few contract violations, indicating that they give a good approximation of the encapsulation used in the programs (and therefore of any manually added aliasing contracts). In Trove, we recorded the lowest rate of contract violation, at 9.3 percent (783,351 of 8,453,603 contract evaluations). NekoHTML had a highest rate of contract violation at 22.2 percent (17,612 of 79,295), while in JUnit and JGraphT the violation rates were 13.3 and 18.0 percent respectively.

We compiled each of the programs twice, first using the standard compiler (called `javac0` below) and then using our the modified compiler (`javac1` below). For both compilations, we noted the time taken by the compiler as well as the size of the generated Java byte code in bytes.

Table 3 shows the results of these measurements; they show that compiling a program with `javac1` takes between 1.6 and 2.0 times longer than using the standard compiler; this closely corresponds to the amount of byte-code generated, which is around 1.6 to 1.8 times larger using `javac1`.

We also measured the time taken to execute the test suites when compiled with both of the compilers. In addition, we measured performance when contracts were only tracked by the contract library but not evaluated; this is equivalent to tracking of references for each object, for example for investigating the topology of the heap. Table 4 shows the measurement results.

The runtime measurements show that JGraphT runs 21 times more slowly with aliasing contracts than usual. While this is a significant decrease in performance, it is caused by a small number of test cases; 44 of 51 test suites run less than 10 times more slowly with contracts; the remaining 7 execute between 14 and 91 times more slowly. The situation is similar in Trove: only 5 of 26 test suites are slowed down by more than a factor of 10 in the presence of contracts.

For example, the worst-performing test suite in JGraphT is called `FibonacciHeapTest`, running 91 times more slowly with than without contracts. It builds up a fibonacci heap, performing 20,000 insertions followed by 10,000 removals. Building such a large and complex data structure requires numerous object accesses and assignments (leading to a lot of contract checks, additions and removals), explaining the observed performance overhead.

Trove runs around 11 times more slowly with aliasing contracts than normally. NekoHTML, as expected, is less affected by the presence of aliasing contracts due to the amount of input and output involved in its test cases; it runs 1.8 times more slowly with aliasing contracts. JUnit's performance is also only slightly affected by contracts, slowing down by a factor of 1.6.

Merely tracking contracts but not evaluating them roughly halved the performance overhead; this effect was particularly marked in Trove, which ran almost 4 times faster when only tracking references. This shows that the contract

library spends roughly half of the time evaluating contracts and the other half tracking them. These results also show the feasibility of using JaCon as a tool for tracking references to objects independently of aliasing contracts.

Java's garbage collection statistics indicate the impact of aliasing contracts on memory usage. We observed a significant increase in heap size for JGraphT and Trove, the two programs whose performance was most strongly affected by aliasing contracts. For JGraphT, the maximum heap size recorded in the presence of aliasing contracts was 722 kB, compared to 53 kB without contracts. This result is consistent with both the large increase in execution time for JGraphT, as well as the number of contract additions performed by JGraphT, more than 200 million. Similarly, Trove triggered around 12 million contract additions and showed an increase in maximum heap size of around 200 kB. For the remaining programs, maximum heap size increased by less than 35 kB, reflecting significantly lower number of contract additions they perform (around 210,000 for NekoHTML and 430,000 for JUnit).

6 Discussion

Aliasing contracts are a novel approach to alias protection. They gain much of their flexibility by using dynamic rather than static checking; this allows the definition of complex encapsulation policies. Below, we highlight and discuss some important considerations about aliasing contracts.

Runtime Performance. The case studies we conducted using four open-source Java programs show that aliasing contracts can cause significant performance overheads. However, we observed a wide range of behaviour; some programs, including JUnit and NekoHTML, were barely affected by aliasing contracts and would remain fully usable in the presence of contracts; others, including JGraphT, experienced severe decrease in performance, rendering the programs difficult to use in practice.

We argue that the effects of aliasing contracts vary depending on certain program attributes. The main performance issue occurs when many variables refer to the same object and re-assignment of variables is frequent. These conditions occur, for example, in programs building complex data structures, involving many assignments to build the data structure and many object accesses to visit it. We can see this effect in the performance of JGraphT's unit tests and in the example program in Listing 2.

Although unacceptable in release versions of a program, the performance overhead we measured is not a significant issue during the testing phase of a project. Executing the unit tests of our test programs in the presence of aliasing contracts is feasible, given the performance we recorded. This demonstrates that it is indeed possible to use aliasing contracts as a testing and debugging tool. Our results show that the performance of aliasing contracts is comparable to existing debugging tools such as Valgrind [13]; for Valgrind, some programs run up to 58 times more slowly using Valgrind, although the slow-down factor is below 20 for most programs.

Our case studies have also demonstrated JaCon's robustness and ability to handle real-world software; we used it to generate more than 400,000 lines of code and execute multiple large test cases. This demonstrates its robustness and ability to handle real-world software.

Contract registration and de-registration (via hash tables) and contract evaluation are 'sensible' implementations which have only been optimised in terms of whether repeated evaluation is necessary. JaCon was built as proof-of-concept; we suspect more attention to low-level implementation would expose further performance improvement of 2–5 times in contract tracking and evaluation.

Garbage Collection. In Java, objects persist until there are no more references to them and they are eventually garbage collected (at an unspecified time). This inherent uncertainty about how long objects exist significantly influences the semantics of aliasing contracts and indeed Java finalisation itself.

An object's fields (and the references they store) persist in memory until finalisation. But 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 garbage collected?

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

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 and the references they store remain in memory until garbage collection, and so do their contracts. This means that the contract of an object waiting to be garbage collected can cause contract violation; garbage collection can remove contract violations but never introduce them.

Advanced Features of Aliasing Contracts. We have presented only the basic aspects of aliasing contracts; more detail is available in [18]. Two powerful features which were not discussed here are *encapsulation groups* and *contract parameters*. Encapsulation groups allow objects to be grouped, making it possible to refer to a whole group in a contract rather than just single objects. Encapsulation groups can also be specified recursively, enabling deep and transitive contract specifications. The power of encapsulation groups lies in the fact that they can contain an unlimited of objects, which may vary at runtime. This, for example, makes it possible to group all nodes in a linked list, without knowing exactly how many nodes will be in the list at runtime:

```
class LinkedList {
  Node head;
  group allNodes = {head,
    head.nextNodes};
}
```

```
class Node {
  Node next;
  group nextNodes = {next,
        next.nextNodes};
}
```

The encapsulation group `nextNodes` in `Node` contains all subsequent nodes in the list: `next` and all nodes following it (`next.nextNodes`). The group `allNodes` in `LinkedList` contains all nodes in the list, `head` and all nodes following it (`head.nextNodes`). At runtime, JaCon evaluates these groups by resolving all paths in the group definitions to objects.

Groups can be referenced in contracts using the `in` operator; for example, the contract "`accessor in list.allNodes`" checks if `accessor` is in the set of objects described by `list.allNodes`, that is if it is a node of `list`. Using this contract, we can, for example, express the condition that all nodes in a linked list should have mutual access to each other. This example demonstrates how encapsulation groups can be used to specify transitive aliasing conditions.

Contract parameters make it possible for different instances of the same class to exhibit different aliasing behaviour. A class can take contract parameters which must be instantiated when an object of the class is created; the parameters can be used as contracts in the class, changing aliasing behaviour of objects of the class depending on the contract instantiations provided.

With encapsulation groups and contract parameters aliasing contracts can directly express the aliasing policies enforced by existing static alias protection schemes, including Clarke-style ownership types [6] and universe types [12]. They can also express many aliasing conditions not expressible with existing schemes.

7 Conclusions and Further Work

We have presented a novel, dynamic approach to alias protection called aliasing contracts. They are a general and flexible scheme, able to model a wide variety of different aliasing policies, including those already enforced by static schemes.

We have developed a prototype implementation for aliasing contracts in Java, JaCon; by running JaCon on open-source programs, we have demonstrated the practical feasibility of aliasing contracts, for example during the testing phase of a project. Our tests have shown that JaCon can handle significant programs and that its performance is comparable to existing debugging tools like Valgrind.

We are currently developing a static checker to check many common aliasing contracts at compile-time. This would allow us to eliminate some contracts during the compilation process, leaving only the more complex ones to be checked at runtime; combining the static verifier with JaCon will significantly improve performance. Combining static and dynamic checking of contracts is analogous to gradual typing [16], which combines static and dynamic type checking to gain the advantages of both.

We are also working on allowing temporary suspension of contracts; this can be used to model borrowing [4], where access to an object is granted *temporarily*, for example for the duration of a method call.

Acknowledgements. The authors thank the Rutherford Foundation of the Royal Society of New Zealand for the scholarship which supported this work, and the anonymous referees for their helpful suggestions.

References

1. Almeida, P.: Balloon types: Controlling sharing of state in data types. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 32–59. Springer, Heidelberg (1997)
2. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, pp. 213–223. ACM (2003)
3. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
4. Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: A generalisation of uniqueness and read-only. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 2–7. Springer, Heidelberg (2001)
5. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 15–58. Springer, Heidelberg (2013)
6. Clarke, D., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. ACM SIGPLAN Notices 33, 48–64 (1998)
7. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In: DLS: Dynamic Languages Symposium, pp. 41–52. ACM (2007)
8. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010)
9. Hogg, J.: Islands: aliasing protection in object-oriented languages. In: OOPSLA, pp. 271–285. ACM (1991)
10. Li, P., Cameron, N., Noble, J.: Mojojojo - more ownership for multiple owners. In: FOOL (2010)
11. Meyer, B.: Writing correct software with Eiffel. Dr. Dobb's Journal 14(12), 48–60 (1989)
12. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. In: Poetzsch-Heffter, A., Meyer, J. (eds.) Programming Languages and Fundamentals of Programming (1999)
13. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI, pp. 89–100 (2007)
14. Oracle Corporation: OpenJDK (2013), `http://openjdk.java.net`
15. Sergey, I., Clarke, D.: Gradual ownership types. In: Seidl, H. (ed.) Programming Languages and Systems. LNCS, vol. 7211, pp. 579–599. Springer, Heidelberg (2012)
16. Siek, J., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
17. TIOBE software: TIOBE programming community index for (May 2013), `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
18. Voigt, J., Mycroft, A.: Aliasing contracts: a dynamic approach to alias protection. Technical Report UCAM-CL-TR-836, University of Cambridge, Computer Laboratory (June 2013)
19. Westbrook, E., Zhao, J., Budimlić, Z., Sarkar, V.: Practical permissions for race-free parallelism. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 614–639. Springer, Heidelberg (2012)