Journal of Computer Sciences

Original Research Paper

Abstract Interpretation of Java Bytecode for Immutability Analysis

Oliver Haase

Hochschule Konstanz University of Applied Sciences, Konstanz, Germany

Article history
Received: 28-07-2014
Revised: 18-08-2016
Accepted: 20-08-2016

Email: haase@htwg-konstanz.de

Abstract: Even though immutability is a desirable property, especially in a multi-threaded environment, implementing immutable Java classes is surprisingly hard because of a lack of language support. We present a static analysis tool using abstract bytecode interpretation that checks Java classes for compliance with a set of rules that together constitute state-based immutability. Being realized as a Find Bugs plug in, the tool can easily be integrated into most IDEs and hence the software development process. Our evaluation on a large, real world codebase shows that the average run-time effort for a single class is in the range of a few milliseconds, with only a very few statistical spikes.

Keywords: Immutability, Abstract Interpretation, Bytecode

Introduction

Due to the use of multi-core technology in modern desktop computers, laptops, tablets and even smart phones, concurrent programming has become the standard programming paradigm for virtually all software development. At the same time, Java, one of today's predominant programming languages, supports concurrent program execution only rudimentarily, especially when it comes to the design of threadsafe classes. This becomes particularly evident in the context of immutable classes; instances of immutable classes can be shared freely and without synchronization between threads without the risk of data races and inconsistent states (Bloch, 2008; Peierls *et al.*, 2005). In Java, however, it is an almost surprisingly difficult task to code immutability correctly. The properties that must be implemented are not widely known amongst programmers and neither the language itself nor the commonly used development environments provide sufficient support to help programmers with this task.

In this study we present jic (java immutability checker), a comprehensive analysis tool that checks Java classes for immutability. As opposed to other tools that detect only straight-forward immutability breaches, jic performs a thorough analysis by means of abstract interpretation. For its analysis, jic checks all constructors and all methods of a class for violations of any of the rules for immutable classes. Abstractly interpreting a method basically means to execute it for all possible input values at once. Whenever a conditional statement is executed and the condition cannot be completely evaluated because it depends on some input-specific value, all possible execution paths must be followed. Evidently, this can lead to the problem of path explosion for complex methods. To mitigate this problem, we cache the execution results of nested methods calls, which reduces the total of nested method evaluations by about 25%. There remains, however, still the possibility of a class to be too complex to be completely analyzed by abstract interpretation. In our experience with a big, real life project (Apache Tomcat 7.0), only one of the approximately 2 400 classes fell into this category. All other classes could be analyzed, on average even in a very short time as we will see in section 6. We therefore believe that jic can be a valuable tool for the development of thread-safe and in particular immutable, Java classes.

Rather than on the Java source code level, we perform the abstract interpretation on the bytecode level, because of the relative simplicity of the underlying virtual machine model. As a side effect, the analyzer is usable for other languages that are executed within the Java Virtual Machine, such as e.g., Groovy, Scala and Clojure.

The rest of this paper is organized as follows: In section 2, we give an overview of relevant, related work before we give our pratical definition of immutability in section 3. Section 4 is concerned with the dependency of a class's immutability on other classes, followed by a description of our implementation in section 5. In section 6 we evaluate our approach and finally, section 7 concludes this article.

Related Work

Even though immutability seems such an obvious property, there exist several definitions for immutability in Java. For instance, give a definition in (Peierls *et al*., 2005) that includes proper construction as a pre-requisite to immutability; J. Bloch's definition in (Bloch, 2008), on the other hand, requires immutable classes not to be extensible, but does not mention proper construction. Haack *et al*. (2007), the authors differentiate between state based and *observable immutability*; while the former is more suitable for automated checking, the latter describes the intended effect of immutability, namely for immutable instances to be freely shareable without the need for synchronization in a multi-threaded environment.

The aim of escape analysis is to detect which objects remain confined to or escape a certain scope. Usually, the scope is the method stack, or the current thread. In the former case, if an object stays confined to the method stack, i.e., is only known to local variables rather than other objects, it can be allocated on the method stack instead of on the global heap. Stack allocated objects are automatically garbage collected when the method frame is popped from the stack, reducing the task of the heap garbage collector and reducing the application's memory footprint. In addition, stack confined objects are also thread confined and thus access to them need not be synchronized. The same goes for objects that do escape the method stack but remain confined to the current thread. Escape analysis can also be used for proper construction checking, with the scope not being the method stack or the current thread, but the object's constructor. For an overview of the area of escape analysis, see, e.g., (Ruf, 2000; Choi *et al*., 1999; Bogda and Holzle, 1999; Aldrich *et al*., 1999; Choi *et al*., 2003).

Immutability

From a semantical point of view, an immutable *object* is one whose behavior and visible state are constant no matter when or in which (potentially concurrent) order its methods are invoked and its visible state is read. With such a definition of *observable immutability* (Haack *et al*., 2007), the invisible state of an immutable object may change over time as long as the visible behavior remains the same. Even though this might seem unusual, it allows for techniques like lazy initialization and memoization that are taken advantage of, e.g., by the immutable Java String class.

Observable immutability, however, is not trivial to implement correctly and hard to check for a given class. Therefore, idioms and best practices for the correct implementation of immutable classes typically build upon a *state-based* definition of immutability.

Because our objective is to provide a rule set for an automatic immutability checking tool, we do the same and give a state-based definition of immutability, knowing that our definition will exclude classes that are observably but not state-based immutable (To mitigate this problem, our tool employes a white list that contains the known (observable) immutable platform classes, such as String and the wrapper classes for primitive types). In a second step, we will deduce a set of immutability properties that can automatically be checked. We start with a definition of proper construction that will be used within our immutability definition:

Definition 1 (Proper Construction). An object is properly constructed if it becomes accessible only after complete construction. A class is properly constructed if its instances are properly constructed.

Evidently, proper construction always is a desirable property. For immutable classes, it becomes indispensable, because otherwise a change of the object state can be observed during the construction process. We now give our high-level, state-based definition of immutability:

Definition 2 (Immutability). An object is immutable if (a) it is properly constructed and (b) its state cannot be modified after construction. A class is immutable if its instances are immutable.

For our analyzer, we aim at a set of rules that is equivalent to the above definition and that code can be automatically checked against. Because Java code defines classes, these rules will check for immutability on the class rather than on the instance level. We start with a rule that covers part (a) of definition 2:

Rule 1. A class is properly constructed, if in each of its constructors, the this reference is not published before completion of the construction process.

This rule might seem too strict at first glance, as publication of the this reference after proper initialization of all fields seems harmless. However, because compilers can reorder instructions as long as single-threaded equivalence is preserved, even publication of the this reference in the last line of a constructor is a violation of proper construction.

Defining a set of rules that guarantee part (b) of definition 2 is a little bit more challenging. We start our considerations with the concept of the state of a class instance.

Clearly, all primitive fields as well as all reference fields of an object o1 belong to o_1's state. If, however, the target of a reference field, i.e., the referred object o_2, also is part of o_1's state depends on o_2's ownership. In a language that uses the same kind of references for all different kinds of relationships, including aggregations, compositions and associations, it is impossible to know the boundaries of an object's state from its class definition. Semantically, the set of referred objects that belong to an object's state can range from none (*shallow immutability*) to the object closure, i.e., all objects that are directly or indirectly referred by the object (*deep immutability*). The definition of semantically more adequate, finer grained levels of state boundaries is the goal of *ownership systems*, see, e.g., (Aldrich and Chambers, 2004; Dietl and Muller, 2005; Vitek and Bokowski, 2001). Ownership system require to extend standard Java by additional mechanisms. In the absence of a language intrinsic ownership system, we take a conservative approach and assume the object's closure as the object's state boundaries.

We are now prepared to define the rules for state unmodifiability. We divide the problem space step by step by starting with the following simple and obvious rule:

Rule 2. All fields, i.e., instance variables, must be final.

For fields with a primitive type, rule 2 ist not only necessary but also sufficient, because a primitive final field can be set only once during construction. For these fields, we are done. What remains to be further considered is reference fields, because rule 2 only guarantees the references to be unmodifiable, but not the referred objects. To prune the potential complex graph of directly and indirectly referred objects, we only consider mutable objects, recursively applying our immutability rules- and operationally our immutability analyzer-to the types of referred objects. Similar to primitive fields, for reference fields with immutable targets, rule 2 ist both necessary and sufficient.

The first rule that is concerned with mutable targets of reference fields ensures that they cannot be manipulated directly from the outside:

Rule 3. Reference fields to mutable data must be private.

Rule 3 concerns one aspect of encapsulation; full encapsulation requires two more rules that ensure that the outside cannot gain direct access to the respective field. The first one affects constructors:

Rule 4. References to mutable data that enter constructors may not be directly assigned to the fields of the object under construction, but must be deep copied first.

If a reference to mutable data comes from the outside, then the outside might, at any point in time, modify the referred data. Rule 4 prevents situations like these from happening. The complementary rule ensures that encapsulated mutable data does not leak to the outside:

Rule 5. Reference fields to mutable data must not be published directly; instead, deep copies must be created and published.

Now that the outside is prevented from manipulating the mutable target of a reference field, what remains to be taken care of is that the class itself does not make any modifications to its own instances.

Rule 6. There must be no mutators, i.e., no state changing methods.

In (Block 2008), as an additional rule immutable classes are required not to be extensible. The rationale of that rule is that a subclass of an immutable class can easily break the immutability rules and itself be mutable. Due to the substitution principle, the type of a reference variable, field, or parameter can therefore be immutable, while the runtime type of the referred object can be mutable, if immutable classes can be extended.

Nevertheless, we *do* allow immutable classes to be extensible, taking into account that the subclasses of\an immutable class can themselves be mutable and that the immutability of the static type of a variable is not enough to assume the immutability of its value at runtime. To become more concrete, this consideration materializes in rules 3, 4 and 5. In all of these rules, whether a field or parameter, respectively, is considered immutable or not, is determined as follows:

- If the runtime type of a reference field is known-because the referred object has been created previously in a constructor-then the field is immutable if its runtime type is immutable
- If the runtime type of a reference field is unknown, then the field is considered immutable only if its static type is both immutable and final

Dependencies on Other Classes

Even though immutability is a local property that a class does or does not have, no Java class is entirely independent of other classes; classes have superclasses, may have inner and outer class and use other classes by instantiating them and calling their methods. In this section, we categorize the types of dependencies a class has on other classes with respect to its immutability property and how these dependencies are dealt with.

Superclasses

As a subclass inherits the components of its superclass, there is a natural and very tight dependency of a subclass on its superclass. In term of the immutability property, this dependency is two-fold:

- First, a subclass can only be immutable if its superclass is immutable, too. We therefore also analyze the immutability of the superclass when analyzing the subclass

- Secondly, when analyzing a method for immutability, this method may call other methods. Often, the (correct) behavior of the called method is essential for the calling method's compliance with the immutability rules. When the called method belongs to the calling method's superclass, we do not consider the called method an alien method, but an *internal* method and hence analyze into the called method

Evidently, if the superclass changes and its immutability property with it, the immutability of the subclass gets broken without any changes to the subclass. Also, the implementation of a method of the superclass can change in a way that does not violate the superclass's immutability but breaks the subclasses immutability. This potential effect is just another variant of the so-called *fragile base class problem* (Mikhajlov and Sekerinski, 1998) that describes typical problems arising from the tight coupling between super and subclasses.

Inner and Outer Classes

Similar to super and subclasses, inner and outer classes have a very tight relationship with each other. Whoever has control over the implementation of a class also controls the implementation of its inner and outer classes if there are any. We therefore do not consider methods belonging to an inner or outer class *alien* methods, but treat them as *internal* methods and consequently step into them for analysis if they are called from within an analyzed method. As with superclasses, a change of an inner or outer class might break a class's immutability. However, in contrast to superclasses, inner and outer classes need not be immutable themselves to allow for a class's immutability.

All Other Classes

A class's immutability must not depend on the particular implementation of any other class, except its superclass, its inner and its outer classes. Methods of other classes are considered alien methods; if an alien method is called from a method being analyzed then it must not be analyzed, but its worst possible behavior must be assumed. In the context of immutability checking, the worst possible behavior means that the alien method:

- Publishes (makes external) all input parameters including the this reference if its amongst them
- Returns a reference to an external object if its result type is a reference value

Only if this worst case behavior does not lead to an immutability violation-be it immediately or during the further evaluation process-is the call to the alien method legal in terms of immutability. However, even though a class's immutability must not rely on another class's implementation, it may rely on its *contract*. If a class contract specifies the class to be immutable-we use an @Immutable annotation for that matter, as will be discussed in section 5.3-then we assume it to comply with the contract. This consideration is relevant to distinguish between mutable and immutable reference fields of a class, see rules 3, 4 and 5.

Implementation

The jic immutability checker is implemented as a generic abstract bytecode interpreter with hook points for tests that can introspect the state of the virtual machine, together with specific tests that check for immutability breaches. In this section, we first describe the abstract interpreter and then the immutability checks.

Abstract Interpreter

The data structures and the inner workings of the abstract interpreter are similar to a regular Java virtual machine, in the sense that for each method to be analyzed, it computes step by step the effect of each instruction on the current state, i.e., on (a) the values of all local variables, (b) the operand stack and (c) the object heap. The pseudocode in algorithm 1 shows how the abstract interpreter transfers the initial state of a method into its final state. As can be seen, the local variables are initialized with the arguments into the method; due to abstract interpretation, these will be symbolic rather than concrete values:

Algorithm 1 Abstract interpretation of a Java method's bytecode.

```
initialize localVars with args
opStack←θ
heap←θ
instr←first instruction
while instr 6 ≠ null do
  localVars←localVars.transfer(instr)
  opStack←opStack.transfer(instr)
  heap←heap.transfer(instr)
end while
if instr.isBranchInstruction() then
  instr←instr.target()
else
  instr←instr.next()
end if
```

As can also be seen in algorithm 1, each of the three state components (*localVars*, *OpStack*, *heap*) has a *transfer* function that computes the output state component depending on the input state component and the current instruction. For *localVars*, e.g., the *transfer* function is the identity function for all but store instructions. For a store instruction, the indicated local variable is updated with the given value. Likewise, the *opStack transfer* function updates the operand stack whenever an instruction consumes stack entries, or pushes new entries onto the stack. The *heap* finally, is modified when a new object or array is created or when an object field (or reference valued array component) is set to a new value, that is a new link is drawn between two objects on the *heap*. For all other instructions, the *transfer* function is the identity function.

Though the basic working of the abstract interpreter is very similar to a regular Java virtual machine, there are also several key differences, as described in the following:

Single Threaded

All immutability rules (i.e., if an object is properly constructed, if the input parameters to a constructor are deep-copied, if the mutable reference fields of an object are published, or if a method changes the state of its object) are independent of whether the corresponding byte code is executed single or multi threaded. Thus, to keep the interpreter simple, it has only one method stack that is associated with the sole thread of execution.

Backward References

In addition to regular object references we also use explicit backward references from the target to the originating objects. These references are a useful performance measure because many immutability checks need to test whether a given object is, e.g., transitively reachable from the this object, see section 5.2.

External Objects

As reference parameters to a constructor or method, we use two special objects, a *mutable external object and an immutable external object*. In the terminology of abstract interpretation, these two special objects are the symbols for unknown reference type values. They represent unknown class instances and arrays alike. When an internal object is published during the course of symbolic evaluation, it is replaced by one of the two external objects, depending on its immutability property.

Symbolic Primitive Values

For each primitive type, there is exactly one symbolic value, i.e., someInt, someDouble, etc. These values represent unknown method parameters of the respective type.

The main reason is as follows: When evaluating a constructor or method, be it directly or indirectly through nested method invocation, we are not interested in primitive result values but only in the object graph that is built during evaluation, because this is where immutability breaches can happen. However, beside not knowing the exact result value of a method with primitive return type, this design decision has two more consequences: (1) For condition branch instructions with numeric conditions, the condition cannot be evaluated and thus all branches must be executed. Evidently, if the code contains many levels of nested conditional branch instructions, this can lead to path explosion. On the other hand, abstract interpretation basically means to evaluate a method for *all* possible input values and thus evaluating both possible paths of a conditional branch instruction seems the natural thing to do. (2) When accessing an array component with an unknown index, each component of the array must be assumed to be the desired one. Thus, the code following the array access must be evaluated for each possible component of the array. For arrays containing many components, this can also lead to path explosion.

Interpretation of Internal Vs. Alien Methods

When a method or constructor *f* is interpreted and within *f* another method or constructor *g* is invoked, then the interpreter steps into g only if it is an internal method-see section 4 for a definition of 'internal'-but not if g is an alien method. To skip an alien method, the interpreter pops the right amount of parameters from the operand stack and pushes a correctly typed symbolic result value onto the stack. For a primitive return type of *g*, the result is one of the primitive symbolic values described above. For a method *g* that returns a reference, the referred object must be considered external to method *f*, because g may have published it prior to returning it to *f*. If the static return type of *g* is an immutable and final reference type, then the runtime type of the result object must equal its (immutable) static type and hence a reference to the immutable external object is pushed onto the operand stack. Otherwise, if the static return type is non-final or mutable, then the runtime type of the result object cannot be known to be immutable and thus a reference to the mutable external object is pushed onto the operand stack. Also, as already mentioned in section 4, all reference parameters into *g* are made external, because g might publish them and public objects are the most harmful in terms of immutability.

For internal methods, on the other hand, the interpreter pushes a new frame onto the method stack and recursively continues interpretation with the called method, the same way a regular JVM would do it.

The challenge, however, is to determine whether a method is an internal or an alien method. The challenge lies in the fact that for a method call *o*.*g*(), the target object *o*'s runtime type is not known to the interpreter if *o* is a reference parameter that has been passed externally into the calling method *f*. In that case, the evaluator proceeds as follows: If the static type of *o* is an alien type, then *o*.*g*() must be an alien method call, independent of *o*'s runtime type; if *o*'s static type is an intenal class, the interpreter performs the regular late binding resolution, starting at *o*'s static type. If the resolved method implementation is final-either because the target class is final, or because *g* is final, then *o.g*() must be an internal method call and hence the evaluator can step into it.

Forking

Because of the symbolic values for input parameters, there are several situations when the interpreter has more than one option to continue execution:

- A conditional statement whose condition involves a symbolic value and thus cannot be (completely) evaluated
- Access to an array component at an index that involves a symbolic value
- A try block with multiple catch clauses

In all of the above cases, the interpretation process forks and follows all possible options. In some cases, this means to execute different paths, in other cases it means to assume different array components and then continue for each one of them. To avoid the problem of path explosion, the interpreter combines the results to the largest possible extent at the end of method calls, i.e., before returning the results to the calling method.

Immutability Checks

The abstract interpreter contains several hook points where the state of the VM can be checked for violations of the immutability rules from definition 2. These hook points concern all instructions that either modify the object graph, or publish-or may publish-an object outside its intended scope. More specifically, these are:

Hook Point (i)-Object Field Assignment

Assigning a reference value, *v*, to a field, *f*, of an object, *o*, that is *o.f* = *v*, establishes a reference from *o* to *v*, if f has a reference type. This newly created reference can be checked for compliance with the immutability rules.

Hook Point (ii)-Array Component Assignment

Similarly, assigning a reference value, *v*, to a component of an array, *a*, that is *a*[*i*] = *v*, establishes a reference from *a* to *v*, if a has a reference component type. However, even if *v* is a primitive value, the assignment can constitute an immutability breach, for example if the array is a field of a supposedly immutable object.

Hook Point (iii)-Static Field Assignment

Assigning a reference value, *v*, to a static field publishes the value and hence can be an immutability breach depending on how *v* is connected in the object graph.

Hook Point (iv)-Virtual Method Invocation

Within a virtual method, any of the above can happen to any of the reference parameters; whether this can lead to an immutability breach depends on the connection status of the individual parameters.

Hook Point (v)-Method Returning

If a method returns a reference value, then the referred object escapes the scope of this method. Whether this is harmful or harmless depends on the connection status of the referred object.

Checking rules 2 and 3 from section 3 is as trivial as inspecting the final and the access right modifiers of all or the mutable reference fields, respectively. A breach of any of the rules 1, 4, 5 and 6 technically translates to an invalid object graph in the sense that one object, o_1, is-or might be-reachable from another object, o_2, when it should not be. In the following, we describe to what invalid object interconnection each of the above mentioned rules corresponds:

Proper Construction Check (Rule 1)

An object is improperly constructed, if the this reference, either directly or indirectly, escapes the scope of its constructor. An indirect escape occurs, if an object that (transitively) refers the this object, escapes the constructor scope. On the Java source code level, this happens, e.g., when an inner instance of the this object, which always has a hidden reference to its outer object, escapes. The proper construction check is hooked into hook points (i), (ii), (iii) and (iv).

For hook point (i), the check is applied to each reference parameter of the virtual method; for hook points (ii), (iii) and (iv), the check is applied to the value v to be assigned. For hook points (ii) and (iii), however, harm is only done if v is assigned to an object or array, respectively, that is external to the constructor.

Constructor Parameters are Copied Check (Rule 4)

References to mutable objects that are passed into a constructor must not be directly assigned to any field or subfield of the this object. This check is only hooked into hook points (ii) and (iii); the check specifically tests if (a) the value v to be assigned or any object transitively referred by *v*, is external to the constructor and (b) the object or array that is assigned to is a field or a subfield of the this object.

Fields not Published Check (Rule 5)

In the most general case, a reference field (to mutable data) of an object, o_1, is published if an object, o_2, that refers to a field or a subfield of o_1, is published. Such an object o_2 can be published as a parameter to a virtual method, by assignment to an external object, an external array, or a static field; or by returning it as a method result. Consequently, the fields not published check is hooked into all hook points.

No Mutators Check (Rule 6)

Mutating an object, *o*, whose fields are all final boils down to assigning a value to an object field or an array component that is reachable from *o*. Consequently, this check is hooked into hook points (i) and (ii).

Technical Realization and Usage

To allow for an easy and seamless integration of the jic analyzer into the software development process, we have implemented it as a FindBugs plugin. As FindBugs is available as a plug in for the most widely used Java IDEs, including Eclipse, NetBeans and IntelliJ, jic can easily be integrated into most academic as well as commercial development processes.

FindBugs uses Apache BCEL for the bytecode AST representation and so does jic for the sake of easy integration into FindBugs. However, changing the bytecode representation to a different AST format, if it turned out to be necessary or convenient, would be straightforward and can, e.g., be achieved by a pre-phase that converts one AST representation into another one.

When running FindBugs within a Java IDE, the developer can select Java classes on the class, the package, or the project level. Configured appropriately, jic will check those classes that are annotated as immutable for their compliance with our immutability definition in section 3. Jic accepts any @Immutable annotation as a trigger for its check; this includes, e.g., the predefined @Immutable annotations in the packages net.jcip.annotations and javax.annotation.concurrent, but also any other pre- or user defined @Immutable annotation. Rule 1 of definition 2, the rule for proper construction, however, is checked not only for supposedly immutable, but for all classes. This is because proper construction is not only a fundamental pre-requisite for immutability, but also an essential property for any mutable classes. Class instances are never supposed to be visible before they are completely initialized and therefore in a valid state with all their consistency invariants established. If an instance escapes during its own construction process, hard to detect bugs can occur even in a single-threaded environment.

Evaluation

We have run a field test of the jic tool on the sources of the Apache Tomcat project, version 7.0, which comprises 2 412 classes that contain 2 753 constructors and 19 391 methods (without constructors). Because the supposedly immutable Tomcat classes are not annotated and thus cannot be differentiated from the intendedly mutable classes, we checked all classes for proper construction only. However, as the abstract interpreter accounts for 89% of the jic implementation, whereas the individual checks only sum up to the remaining 11%, the field test is a good indication for the overall stability of the jic tool.

Functional Evaluation

The analysis of the Tomcat classes has yielded a total of 663 bugs that is comprised of the following categories:

- 1 out of the 2 412 classes was reported as too complex to be analyzed
- In 193 cases, the this reference was passed into an alien method
- In 21 cases, the this reference escaped through indirect passing into an alien method. This happens mostly when an inner instance of the object being created is passed into an alien method, because inner instances have hidden references to their embedding outer instance
- About 448 bugs where subsequent bugs caused by an another bug in another class. In the vast majority of the cases, the subsequent bugs occur in classes that extend improperly constructed superclasses, which renders the subclasses themselves improperly constructed. In around half of the cases, the improperly constructed superclasses are Java platform classes or subclasses thereof. In particular, 108 subsequent bugs occur in specific Exception and Error classes, because the Throwable class uses a native method in its constructors and therefore must be considered improperly constructed. If the developer of a subclass, however, trusts the superclass to be properly constructed nevertheless (because, e.g., the native method does not let the this reference escape), they can switch off the subsequent bug with a FindBugs specific @SuppressWarning annotation

Runtime Performance

We have measured the time a jic analysis takes per class; to get the numbers of a broad array of different classes, we have run the measurement for the entire Apache Tomcat 7.0 package. To average out the effects of spikes due to, e.g., background tasks such as garbage collection, we've collect the results from 10 different measurement runs, leading to the measurement of around 24 000 class analyses. The measurements were made on a machine with a 2 GHz Intel Core i7 processor, 8 GB of RAM and 500 GB of secondary storage. Figure 1 depicts a plot of the result; please note the logarithmic scale of both axes that was necessary because the vast majority of analyses ranges within a few milliseconds, with a few spikes up to a maximum of 4217 ms. Out of the around 24 000 results, 216 values were greater than 100 ms, out of which 31 were greater than 1000 ms.

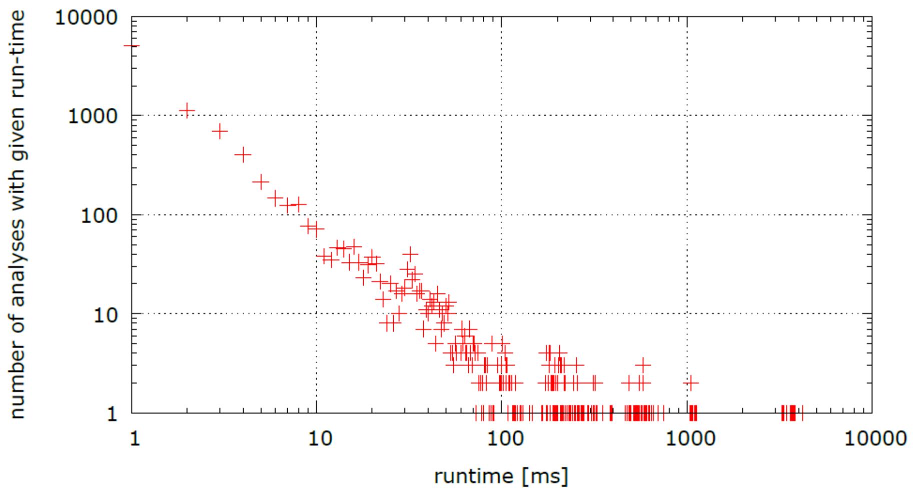

Fig. 1. Measurement of the jic analysis runtime per class, both axes in logarithmic scale

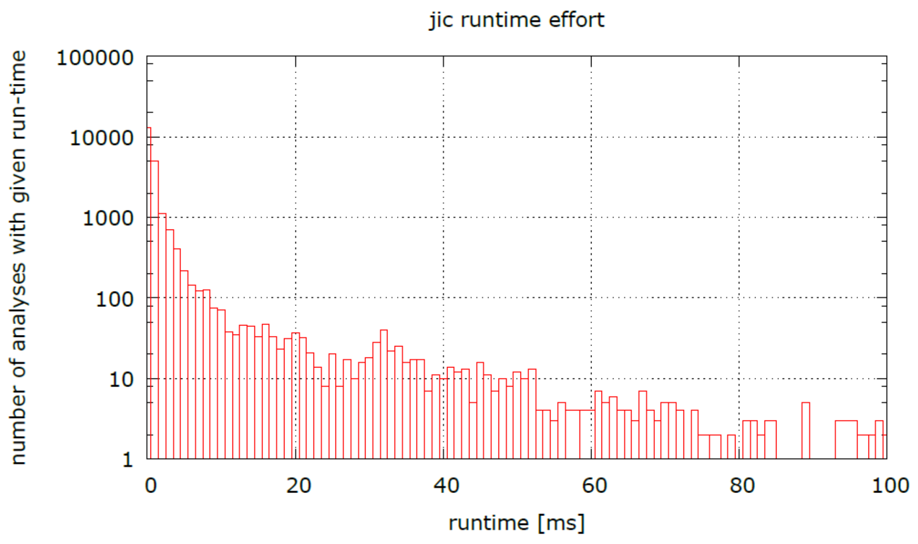

Fig. 2.Measurement of the jic analysis runtime per class, only the y-axis in logarithmic scale, the x-axis zooming into the range of 0 to 100 ms

The histogram in Fig. 2 zooms into the range of 0 to 100 ms on the x-axis, this time with a logarithmic scale only for the y-achsis. This graph gives a better visual impression on how low the measured runtimes for more than 99% of the analysis runs were.

Conclusion and Future Work

Even though immutability is a desirable and seemingly simple property, it is surprisingly difficult to achieve in Java, because of the language's lack of support of it. What is more, there exists not only one but multiple definitions of immutability that differ in the degree of strictness of their rules. Our definition aims at being as little restrictive as possible, while maintaining automatic checkability and, of course, the desired effect of immutable objects being freely shareable without any need for synchronization.

Based on our immutability definition we have presented the java immutability checker, jic, an analysis tool that checks java classes for compliance with a set of rules that resemble immutability. Jic is implemented as an abstract bytecode interpreter, i.e., it performs its analysis on the Java bytecode level. The rationale behind this decision is the relative simplicity of the Java bytecode language and the JVM machine model, as compared to Java source code with its huge and ever growing array of language features.

As a field study, we have evaluated jic on the Apache Tomcat 7.0 project with its around 2 400 classes that comprise around 20 000 methods. The runtime effort a single class analysis is in the range of a few milliseconds for more than 99% of classes, only one class was too complex to be completely analyzable.

We have implemented jic a a FindBugs plugin; as FindBugs in turn is available as a plugin for the most widely used Java IDEs, including Eclipse, Net-Beans and IntelliJ, jic can easily be integrated into most academic as well as commercial development processes. The code, as well as the latest readily packaged FindBugs plugin are publicly hosted at https://github.com/seerhein-lab/jic and https://github.com/seerhein-lab/jic/ releases, respectively and can be used under the Apache License, Version 2.0.

In a future project, we intend to evaluate jic's usefulness for other Java bytecode based languages, such as Groovy, Scala and Clojure. Even though some of these languages-most notably Scala-are designed to avoid the classical concurrency problems by the explicit support of immutably types, some of the sources for immutability breaches do remain. For instance, in Scala a reference to the this reference can escape during object construction just as well as in Java. It will be interesting to study which other immutability rules are relevant for non-Java bytecode based languages, or if there are new rules to be considered as well.

Acknowledgement

This work has been done at the Hochschule Konstanz University of Applied Sciences. The author would like to thank everyone who has contributed to the progress of the research.

Funding Information

This work was funded by the Hochschule Konstanz University of Applied Sciences.

Ethics

This article is original and contains unpublished material. The author approved the manuscript and confirms that no ethical issues arise.

References

Aldrich, J. and C. Chambers, 2004. Ownership domains: Separating aliasing policy from mechanism. Proceedings of the European Conference on Object-Oriented Programming, Jun. 14-18, Oslo, Norway, pp: 1-25. DOI: 10.1007/978-3-540-24851-4_1

Aldrich, J., C. Chambers, E.G. Sirer and S. Eggers, 1999. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In: Static Analysis, Cortesi, A. and G. Filé (Eds.), Springer, Venice, Italy, pp: 19-38.

Bogda, J. and U. Holzle, 1999. Removing unnecessary synchronization in java. Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, Nov. 01-05, Denver, CO, USA, pp: 35-46. DOI: 10.1145/320384.320388

Bloch, J., 2008. Effective Java. 2nd Edn., Addison-Wesley Professional, ISBN-10: 0132778041, pp: 368.

Choi, J.D., M. Gupta, M. Serrano, V.C. Sreedhar and S. Midkiff, 1999. Escape analysis for java. Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, Nov. 01-05, Denver, CO, USA, pp: 1-19. DOI: 10.1145/320384.320386

Choi, J.D., M. Gupta, M.J. Serrano, V.C. Sreedhar and S.P Midkiff, 2003. Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. Programm. Lang. Syst., 25: 876-910. DOI: 10.1145/945885.945892

Dietl, W. and P. Muller, 2005. Universes: Lightweight ownership for JML. J. Object Technol., 4: 5-32.

Haack, C., E. Poll, J. Schafer and A. Schubert. 2007. Immutable objects for a java-like language. Proceedings of the 16th European Symposium on Programming, Mar. 24-Apr. 1, Springer, pp: 347-362. DOI: 10.1007/978-3-540-71316-6_24

Mikhajlov, L. and E. Sekerinski, 1998. A study of the fragile base class problem. Proceedings of the 12th European Conference on Object-Oriented Programming, Jul. 20-24, Belgium, pp: 355-383. DOI: 10.1007/BFb0054099

Peierls, T., B. Goetz, J. Bloch, J. Bowbeer and D. Lea *et al*., 2005. Java Concurrency in Practice. 1st Edn., Pearson Education India, ISBN-10: 8131713393, pp: 424.

Ruf, E., 2000. Effective synchronization removal for java. Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, Jun. 18-21, Vancouver, BC, Canada, pp: 208-218. DOI: 10.1145/349299.349327

Vitek, J. and B. Bokowski, 2001. Confined types in java. Software: Practice Experience, 31: 507-532. DOI: 10.1002/spe.369