Mining for Safety using Interactive Trace Analysis

Stephan Brandauer
Uppsala University
stephan.brandauer@it.uu.se

Tobias Wrigstad
Uppsala University
tobias.wrigstad@it.uu.se

This paper presents the results of a trace-based study of object and reference properties on a subset of the DaCapo benchmark suite with the intent to uncover facts about programs that can be leveraged by type systems, compilers and run-times. In particular, we focus on aliasing, and immutability, based on their recent application in the literature.

To facilitate analyses like this one, we previously created Spencer (`http://spencer-t.racing`, [8]), a web based tool and API that hosts dynamic trace data and enables researchers to query and analyse the data. In this paper we only use data that are openly accessible via Spencer's API – all code written for this paper (not counting Spencer itself) amounts to 13 lines of bash and 260 lines of python to plot the results.

We find that while Java allows aliasing and mutation by default, objects are often unique, unique on the heap, immutable, or stack-bound – 97.7% of objects fulfill at least one of these properties. Furthermore, uniqueness and immutability, or their absence, are class-properties, not object-properties: *e.g.,*, it is surprisingly rare for classes to produce both immutable and mutable instances.

Although we use a different, more fine-grained, methodology, our findings confirm prior results.

1 Introduction

In this paper, we study the object graphs that make up object-oriented programs to uncover common properties about the object structures and how object aliasing and mutation are used. Our main motivation for this work is the wealth of work on controlling and managing object aliasing, *e.g.,*various proposals for ownership [10, 13] and uniqueness [6, 7], and various forms of immutability [18, 22]. With this study, we wish to understand how "programs in the wild" (where such properties are not enforced) compare to such proposals (which impose restrictions), and to motivate their existence. We seek to answer questions such as "how many objects are aliased?", or "are immutable objects used for longer periods of times than non-immutable objects?" There is a substantial amount of programming languages work on abstractions like uniqueness, immutability, and combinations thereof [13, 10, 2, 1, 15].

This is not the first paper to ask these or similar questions, or to study the shape or structure of the heap (see *e.g.,*[16, 17, 22, 20, 9, 14, 18]) and in some respects, this paper reproduces results studied by other authors using different, *and arguably more fine-grained*, techniques. Whereas prior work approaches these questions using static analysis or heap snapshotting, we employ a tracing approach in which we are able to study the life-cycle of all individual objects in a running program. In particular, our traces include all reads and writes to local variables, which are ignored by previous dynamic analyses. This avoids the conservative over-approximation built into static analysis, and avoids false positives due to incomplete data when snapshotting. Thus, to the best of our knowledge, we are the first to attempt to answer such broad research questions with such high-resolution data.

The data sets (the traces in this paper, combined, weigh in at 680GB) are hosted by SPENCER (`http://spencer-t.racing`, [8]), a web based tool that lets users query dynamic program traces using their web browser. All results in this paper are produced using Spencer's data.

Submitted to:
QAPL'17

	Name	Objects	Log	
1.	`luindex`	81,158	5.8GB	37.906.637
2.	`pmd`	131,462	2.7GB	18.107.255
3.	`fop`	521,789	10GB	63.957.143
4.	`batik`	526,945	21GB	134.864.425
5.	`xalan`	1,133,391	48GB	302.083.030
6.	`lusearch`	1,212,743	61GB	380.164.919
7.	`sunflow`	2,419,900	91GB	569.648.600
8.	`h2`	6,655,852	207GB	1.468.550.379
9.	`avrora`	932,085	236GB	1.514.972.478
	Total:	13,615,325	≈680GB	4,490,254,866

Table 1: Currently loaded benchmarks, a similar list can be found in the tool: `http://spencer-t.racing/datasets`.

The main contributions of this paper are the results from analyses of program traces to find evidence of immutability, and uniqueness from 9 programs from the DaCapo benchmark suite and a comparison with and discussion of results from prior work. We use a more fine-grained approach than previous studies by considering all program events, and stack variables.

2 Methodology

We employ a trace-based method to study the behaviour of objects. Our corpus of programs is the DaCapo benchmark suite release 9.12 [3], limiting ourselves to 9 programs[1] because of the volume of data involved. This suite contains a wide range of workloads, including simulations of micro controllers (`avrora`), a static analysis tool (`pmd`), an in-memory database (`h2`), and others. Table 1 lists the benchmarks we are studying. Tracing allows us to track not only how aliases to each object are created, but where they are stored, and how they are used at a level which previous dynamic analyses do not do. For the 9 programs in our study, we recorded $\approx 4.5 \cdot 10^9$ events such as object creation, field read, field write, etc. To facilitate studies like this one, we created Spencer, a web service that hosts large program traces, and provides a user interface and API to query these datasets. All data used in this paper (and more) can be accessed openly, and we will provide appropriate links in the paper.

Spencer uses a JVMTI agent loaded into the a stock JVM that listens to events in the running program. Because of limitations in these events (*e.g.,*JVMTI does not allow events on stack variables), we also instrument the loaded Java byte code on the fly to emit additional events[2]. This is done on the fly with *no need to manipulate program sources*, and automatically includes any loaded libraries in the analysis. At the time the analysis was made, the tooling infrastructure consisted of more than 10.000 LOC of a mix of C++, Java, Scala, and Javascript. These files are loaded into a relational database (PostgresSQL) and hosted by a web server. The web server provides a user interface for users to interactively explore data sets (Figure 1), and also a JSON API (`http://spencer-t.racing/doc/api`). For both web based UI and API, the concept of a *query* is central: a query is a selection operation on the set of objects in a trace – it analyses the whole trace, and returns, as its result a set of objects. Queries can be combined by using query

[1] `avrora`, `batik`, `fop`, `h2`, `luindex`, `lusearch`, `pmd`, `sunflow`, and `xalan`.

[2] Our events: object creation; method entry and exit; field loads and stores; variable loads and stores.

combinators, and the server will cache results that it computed in the database to improve performance.

2.1 Queries

Queries being selections, the queries that a user submits are literally translated to SQL. To run a query, it is enough to access the URL http://spencer-t.racing/query/⟨datasetname⟩/⟨query⟩, the page also contains links to the corresponding API call that will just return the selected objects in JSON format at the bottom.

This paper is not mainly about Spencer; but we will give a brief explanation of the queries[3] that we will will use here:

Query	*Explanation*
ImmutableObj()	This query selects all objects that were modified only in their constructor, but never after.
UniqueObj()	This query selects all objects that had at most one reference from fields or (stack-) variables at one time, but never more.
StackBoundObj()	This query selects all objects that are never reachable from any field.
HeapUniqueObj()	This query selects all objects that had at most one reference from fields at one time, but any number of references from variables.
HeapDeeply(q)	If q selects objects from a trace, then HeapDeeply(q) selects all the objects that are selected by q from which, following only fields, only objects in q are reachable.
Or($q_1 \dots q_N$)	This query selects all objects that are selected by at least one of the inner queries.

3 The Properties of Interest in our Study

We now describe the properties of programs, individual objects and references that are the subjects of our study. For each property, we explain it briefly; outline why it is important and how it is used in the programming language literature; and state the definitions of what we have studied in our traces in relation to the property. A discussion about each property is found in conjunction with the results.

3.1 Property 1: Uniqueness

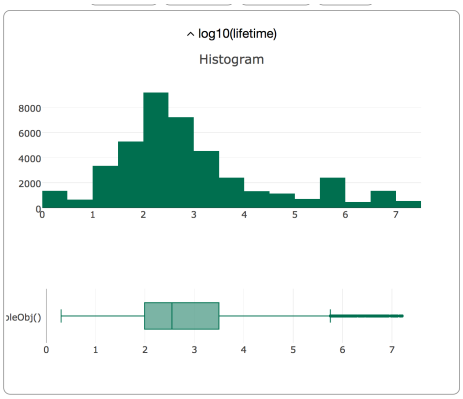
Unique references are references that have no aliases. Unique references simplify reasoning about software, both by programmers and tools. Verifying properties of an object is much easier in the absence of aliasing. When unique references go out of scope, the object they reference can be free'd. Many optimisations are unlocked in compilers due to alias-freedom.

Studying uniqueness through heap snapshots as done by Potanin et al. [20] is simple: count the in-degrees of incoming reference for all objects. However, snapshots have the problem that they are not aware of behaviour in between snapshots – when "no one is looking". We study two definitions of uniqueness:

Heap-Uniqueness A reference is heap-unique if there is, at any one time, at most one reference to it from a live object on the heap, with no restrictions on the number of field references.

[3] These queries are links in the PDF file. Clicking them will bring you to the user interface.

(a) The percentage of objects that are selected by a query, and a sample of them.



(b) Visualisation of the life time of selected objects (the difference between the event index of the last access to the object and the first).

Figure 1: The Spencer user interface as seen from a web browser.

Uniqueness A reference is unique if there is at most one reference to it from either variables or fields. Even though this definition seems to be very constraining, we shall see that a considerable amount of objects and fields fulfill it.

The properties are captured by the Spencer queries HeapUniqueObj()) and UniqueObj(), respectively.

3.2 Property 2: Immutability

Immutable objects are objects that will not change. A classic example of immutable objects in the Java world are strings and boxed primitives such as `Integer` and `Boolean`. Immutability is a powerful property and gives similar reasoning power as uniqueness: a value will not change under foot. The recent years have seen several designs of type systems for Java-like object-oriented programming languages with the aim of simplifying concurrent programming that use some form of immutable object to share data without risking data-races, *e.g.,*[5, 11, 4, 12, 19]. Java does not support immutable objects except for the ability to declare a field as final (a final field has to be assigned to in the object's constructor, and it can never be assigned to outside of the object's contstructor). This is a limited support, *e.g.,* cannot express construction of cyclic immutable structures, or initialisation that is distributed over several parts of a program.

We explore the presence of immutability in three forms:

Shallow-Immutable An object is *shallow immutable* if its fields are never written to – except in its constructor. In Java, this could be handled by final fields, unless the initialisation of the object is complicated or requires some form of delay.

Deeply-Immutable An object is *deeply immutable* if it is immutable and all its fields are deeply immutable. Most immutability constructs that appear in the literature rely on deep immutability.

The properties are captured by the Spencer queries ImmutableObj()) and Deeplu(ImmutableObj()), respectively.

4 Approximating Pseudo Static Properties from Trace Data

Our analysis works on dynamic program traces. We follow an object through the program trace and analyse whether or not the property of interest holds for that object – f.ex., whether the object is immutable. Each analysis partitions the set of known objects into those that satisfy immutability and those that do not. To approximate invariants that could be enforced statically – in particular captured by type system-like annotations as in most works referred to above – we search for patterns among variables, fields and classes:

Field Analysis: For each field F of each class C, we collect all objects that any C-instance ever referred to from the field, yielding the set $O_{C:F}$.

Class Analysis: For each class C, we collect all of its instances, yielding the set O_C.

These sets approximate static properties – "*pseudo static properties*" – in our analysis. For example, if a field F of the class C only stores references to immutable objects (in other words, all objects in $O_{C:F}$ satisfy the immutability property), we hypothesise that there is an invariant in the program that guarantees that all references that could ever be stored in the variable are immutable, too (Section 5.1.1, Section 5.2.1)

We cover field (using the $O_{C:F}$ sets in Section 5.1.1 and Section 5.2.1), and classes (using the O_C sets in Section 5.1.2 and Section 5.2.2). This reasoning is, of course, unsound (it may produce false positives[4]) and should be taken in context with results of sound static analyses (in Section 7) – that will, on the other hand, produce false negatives.

5 Results

We now discuss the outcome of applying our analyses to traces from our program corpus. Since the programs come out of the DaCapo benchmark suite, each program comes with pre-set instructions for how to run it on representative data. Many studies of *e.g.,* performance have been carried out on these programs with the exact same input.

N.B: Results in this section that are not annotated with the name of a specific benchmark are to be read as being a result that summarises the results of all benchmarks.

5.1 Uniqueness and Heap-Uniqueness

Our studies find that 24% of all objects satisfy **Uniqueness** and 45.5% of all objects satisfy **Heap-Uniqueness**. This suggests that aliasing is more commonly happening on the stack and that many objects are either flat or tree-shaped (in fact, the query). The results are visible in Figure 2. Especially interesting is the fact that 97.7% of all objects are "safe" where to be safe means that they are either unique, heap-unique, stack-bound, immutable, or deeply immutable.

In a study from 2004, Potanin et al. [20] find that 13.6% of objects had more than one field pointing to them. We get a similar result: by running the query Or(StackBoundObj() HeapUniqueObj()) – it returns 89.5% on average[5], leaving $\approx$ 10.5% of objects with more than one field reference. Their methodology is different from ours in that they use snapshots of heaps, but not stacks. We are able to show that uniqueness is much lower if stack references are also analysed. Our measuring methodology also works with a continuous view of the heap whereas Potanin's might overlook aliasing that happens in between snapshots.

[4]For example, statically capturing a property can be complicated by value-based overloading. For example, if there can be an assignment from some not always immutable set $O_{C':F'}$ to the variables in $O_{C:M:F}$ when some guard holds true, which guarantees the particular object's immutability.

[5]See `http://spencer-t.racing/json/percentage/pmd/Or(StackBoundObj()%20HeapUniqueObj())`, modify

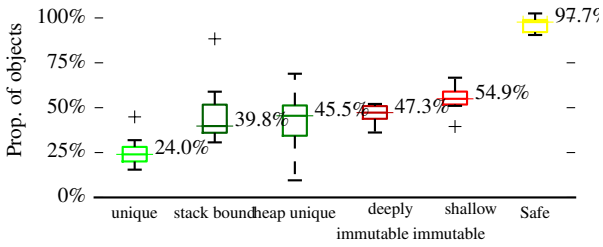


Figure 2: Proportion of **Unique/Stackbound/Heap-Unique/DeepImmutable/Immutable/Safe** objects. A single object can be in several categories, objects that fulfill any definition are classified as "**Safe**". Labels denote the median percentages for all benchmarks.

5.1.1 Unique, Stack-bound, and Heap-Unique Fields

The percentage of heap-unique objects, as shown in Figure 2 varies wildly across different programs. This might imply that heap-uniqueness as a language abstraction is doomed to work only in some niches. However, when we look at the explain pseudo static objects that fields contain, and count those fields that always (and those that never) contain heap-unique objects, we see clear patterns emerge. Figure 3a-e show histograms. Each field goes in the bin according to the percentage of objects it referred to that were selected by the respective query. The distributions are clearly bi-modal – there is lots of fields that rarely contain objects selected, and there is lots of fields that often contain the objects selected by the query.

As programmers, knowing that `x.f` is immutable *most of the time* is of course not very helpful. Invariants are. Our next question is therefore to ask whether there might be static invariants guaranteeing that a property holds *always* or *never* for a given field. This question is what Figure 3f answers: it shows, in green the proportion of fields that contained *only* objects with the given property, and in red the proportion of fields that *never* contained objects with the given property.

A striking property of Figure 3f is that so few fields contain mixes of, say, heap-unique and unique objects. This suggests that invariants about sharing (or its absence) are as common in the wild as invariants about mutability.

5.1.2 Unique, Stack-bound, and Heap-Unique Classes

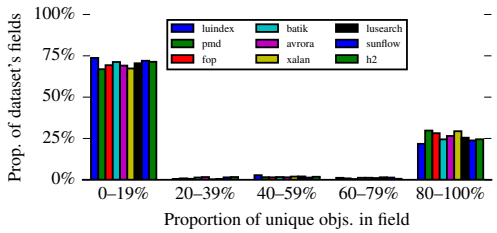
Our results in Figure 4 show[6] that the properties under consideration are predominantly pseudo-static: $\approx 61\%$ of all classes either always or never produce instances that are unique (12% always, and 49% never), $\approx 63\%$ of fields are always or never heap-unique (27% always and 36% never). We don't mention stack-bound objects in these field statistics, as – per definition – no fields ever refer to stack-bound objects.

This results are encouraging for creators of unique type systems as it suggests that in most cases, *a single declaration-site annotation* will be enough to capture uniqueness, rather than annotations on types at *use-site*. Since most programs have far fewer declarations than uses.

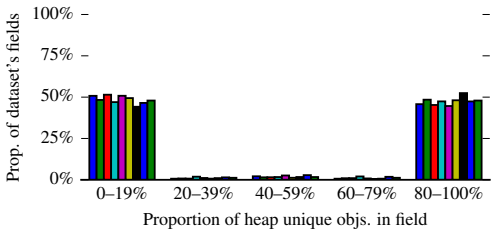
---

the URL for other datasets.

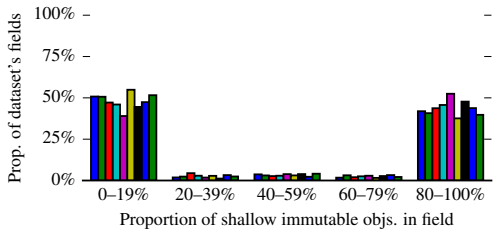
[6]Schema for raw data: `http://spencer-t.racing/json/classpercentage/pmd/Deeply(ImmutableObj())`, and similar for other data sets and queries.



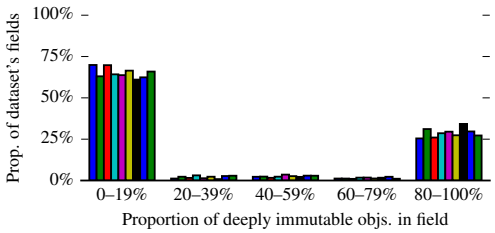
(a) Unique.



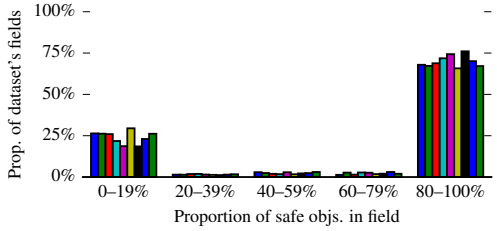
(b) Heap unique.



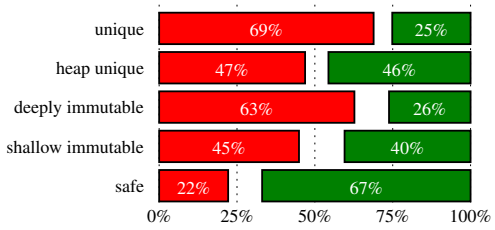
(c) Shallow immutable.



(d) Deeply immutable.

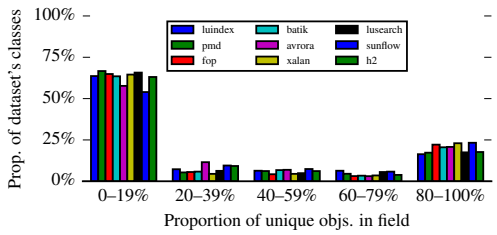


(e) Safe.

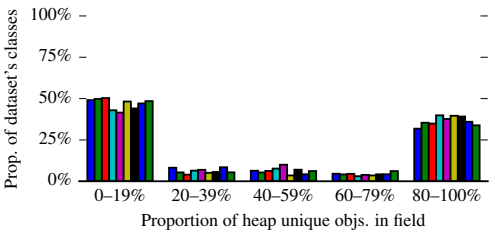


(f) Proportion of fields that contained only objects with the given property (green), not a single one with the given property (red), or other (proportion reported as average of all datasets).

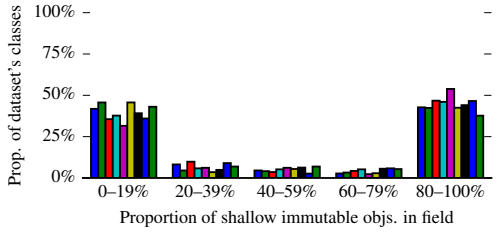
Figure 3: Subfigures a-e: Per syntactic field, out of all the objects it referred to, the percentage of objects with the given property. Fields with a high proportion of objects with the property go to the right-most bin, fields with a low proportion go to the left-most bin. Since benchmarks have different sizes, they also access different numbers of fields in total. To account for this incidental detail, we normalise all bars such that the bar height shows the proportion of all fields *in one benchmark* – in other words, the bars for one color will always add up to 100%.



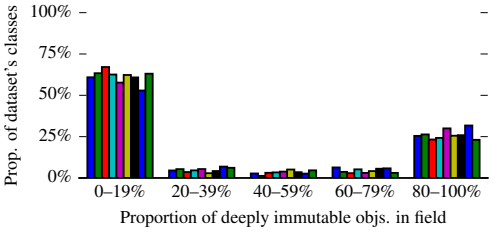
(a) Unique.



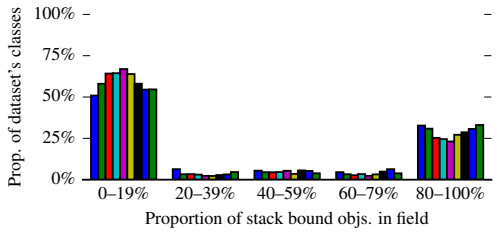
(b) Heap unique.



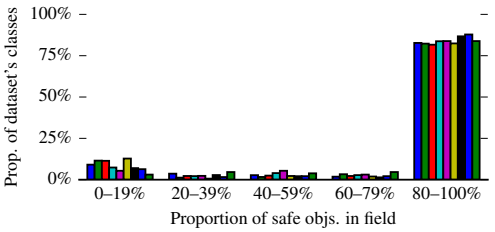
(c) Shallow immutable.



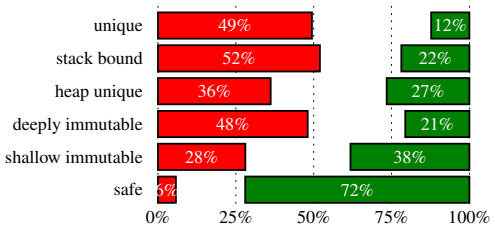
(d) Deeply immutable.



(e) Stack-bound.



(f) Safe.



(g) This shows, for each property, the average across data sets of the proportion of classes that *only* (green, right)/*never* (red, left) produced instances that had the property. We infer, unsoundly, that there exists an invariant that guarantees the property holds statically. Classes with less than 10 instances are ignored.

Figure 4: Subfigures a-f: Per class, out of all the instances this class had, proportion of objects that had the given property. Classes with a high proportion of instances having the property go to the right-most bin; classes with a low proportion of instances having the property go to the left-most bin.

5.2 Immutability

Our analysis in Fig. 2 shows that 54.9% of all objects are shallow-immutable, and 47.9% are deeply immutable. Especially the latter number is encouraging, as deep immutability is a very powerful property when reasoning about code.

5.2.1 Immutable Fields

Our analysis finds that it is quite common for *fields* to refer to deeply immutable, and immutable objects. In Figure 3, we see that 26% and 40% of fields refer to only deeply or shallow immutable objects respectively. The (unsound) conclusion, again, is that up to 26%/40% of fields might be annotated with such qualifiers in a language that provides them. On the contrary, 63%/45% of fields *never* contain deeply/shallow immutable objects.

5.2.2 Immutable Classes

As shown in Figure 4, shallow and deep immutability (or their absence), just like uniqueness, is often a property of the class, not something that varies with individual objects. A third (38%) of all classes produce only immutable instances and half of classes never produce stack-bound objects.

5.3 Summary

In summary, we find that *aliasing is more common on the heap* (Fig. 2): uniqueness is more rare than heap-uniqueness. Figure 3f shows that it is very common for fields to have pseudo-static invariants – there seems to be a lot of knowledge in programmers' minds that is not captured by Java semantics that does not allow distinguishing between the green, the red, or the white part of the bars.

Previous dynamic studies have focused their attention on the objects in benchmarks – essentially providing information like what we do in Fig. 2. What is especially interesting is that the variation in some metrics is quite high (especially heap uniqueness and stack-boundedness), yet where we innovate – by looking at pseudo static results – we find that this variance *does not* seem to carry over to fields and classes: Fig. 3a-f and Fig. 4a-e clearly show that all programs under consideration show very similar behaviour. This is good news for researchers who want to build, or implement language abstractions concerning the properties under consideration.

We believe that program correctness, program understanding, and program performance all benefit from clear expression of mutability and aliasing invariants. The reason is that, when reasoning about a program conservatively, language semantics that allow aliasing and mutability are *hindering*. In nine programs in our study, the freedom given by default by Java's static semantics is rarely needed in practise. Consequently, we argue that instead of enabling, it is hindering, by complicating both reasoning and aspects of the deep run-time. This disconnect between freedoms given and freedoms used is overviewed below:

Semantics	*vs.*	**Programs**
Deep immutability is not a language concern	*vs.*	Half of all objects, a quarter of all fields, and a fifth of all classes only have deeply immutable instances (Fig. 2, Fig. 3f, Fig. 4g)
A field's contents can always be aliased	*vs.*	Almost half of fields are always heap-unique, almost half are never heap-unique Fig. 3f.
Objects may live forever	*vs.*	39.8% of objects (median), and 22% of classes are stack-bound (Fig. 2, Fig. 4g).

5.3.1 Uniqueness

Heap uniqueness in fields is much more common than deep immutability, and so is it in classes. Heap uniqueness is an interesting property – it says that there is one "main alias" and several stack aliases that all have limited life time. It can help the garbage collector, or help optimise memory layouts (heap unique fields do not need to be represented as pointer).

For heap-uniqueness, half of all fields could be annotated unique. Thus, we find no strong evidence that Java's design decision in this respect is well-founded. Making fields unique rather than shared by default would increase static safety, could potentially help the garbage collector (the referent of a unique variable can be de-allocated when the variable goes out of scope) and improve performance (the referent of a unique variable could be on the stack). For designers of future object-oriented programming languages, this is encouraging.

5.3.2 Immutability

With respect to annotations on fields, shallow immutability and deep immutability are roughly as common as **Heap-Unique** and **Unique**, respectively – as shown in Figure 3. Simple immutability semantics – where an object is fixed right after creation – seems to be less useful in practise, *e.g.,* because of delayed initialisation. There have been proposals for Java [21] but, to our knowledge, none have yet been included in a mainstream language[7]. From our results, these patterns that these proposals capture are common.

6 Threats to Validity

Dynamic Analysis The single biggest threat to validity of the results obtained is the fact that we use program traces as input, not static analysis. Dynamic analysis can soundly prove some properties, but not others – For instance, a static analysis can easily prove that a class is *immutable*, but not easily prove that a class is *mutable* (there might be mutating code that is unreachable). Dynamic analysis can easily prove that a class is mutable, but not that it's immutable. As Spencer already stores the classes that where used during an execution, it might be possible to add static analysis facilities in the future – then, an analysis could produce both upper and lower bounds for its results.

Our traces are obtained from single runs of each program, meaning we are not able to detect variations in the same program that stem from non-determinism. Over-interpreting our results may perceive static invariants where there are, in fact, none. This is why our results, should be taken as *research hypotheses*, rather than fact.

Non-Instrumented Classes We are unable to instrument all classes because they are loaded early in the bootstrap process of the JVM, meaning these classes are not covered in our analysis. We mitigate this risk by recording most classes that are being loaded and transforming them as soon as it is safe to do so (at the start of the application). This means that – to the best of our knowledge – all application code and all data structures of the standard library is instrumented.

Native Methods Native methods are rare, but still may cause distorted results. Most notably, we do not get events from methods in the `java.util.Arrays` utility class, such as `copyOf`. That method creates a new array and initialises it to contain the same values or references as an existing array. Our analysis will miss the assignments to the array's cells and therefore classify the newly created copied instance as immutable (assuming it is never changed again), instead of stationary, as it

[7] Scala case classes with lazy fields might qualify

would have been classified with a fully instrumented `copyOf` implementation. Since `copyOf` has semantics similar to a constructor – it creates and initialises an array – we do not correct for this case. However, it may be the case that we miss other important side effects or sharing caused by native implementations. We plan to identify the most common native methods systematically and call equivalent mock implementations in Java (that are instrumented) instead.

Benchmarks are Not Representative Our findings are extracted from 9 programs in the well-known and well-studied DaCapo benchmark suite [3]. While the domains of the programs are very different, it cannot be excluded that our benchmarks are not representative of a larger class of Java programs. We have observed very stable behaviour, and it is interesting to grow the set of datasets, especially to ones implemented in different languages. We'd expect to see more

Bugs in Tracing or Analysis Because of the large amount of code involved in obtaining these traces and analysing them, it is likely that there is some bug somewhere. Due to the nature of tracing large applications, verification is very hard except for very small programs. To mitigate this risk – or at least make sure bugs are found, rather than stay hidden forever – our data sets are hosted publicly. We went a long way to make the data convenient to analyse, in part because we want others to be able to fact check our claims.

7 Related Work

A number of studies on aliasing in object-oriented or imperative programming exist in the literature. In contrast to this work, these studies either employ static analysis, or use a snapshot-based approach to collect data at run-time and excludes stack variables. Even though the stack is bound to be relatively small at any given instant, most objects are referenced from the stack at some point. The difference between **Unique** and **Heap-Unique** hints at the impact of considering/ignoring stack references. Static analysis is by nature conservative, meaning the results from static analysis is likely to include many false negatives, *e.g.,* because an analysis is unable to reason about branching in the running program. On the other hand, static analysis-based approaches are a naturally good fit for determining whether certain static information could be propagated through the code.

Snapshot-based approaches are similar in spirit to this work, but rely on sampling instead of tracing, leading to "lower resolution". Following an object through its entire life-cycle through sampling suffers from false positives (*e.g.,* snapshots before and after a violation of a property P for object o will falsely report $P(o)$ holds) and, according to our methodology, false negatives (a snapshot cannot discover that one alias is effectively buried [6]). An obvious up-side of sampling over our approach is the reduced complexity and improved speed of gathering data.

Hackett and Aiken use static analysis on C programs [14]. They find, like we do for Java, that most fields hold unaliased objects, but are unable to reason about the proportion of unaliased object at run-time for particular program runs. Similar to our findings, but for structs (which importantly lack a **this** pointer), how an object is aliased is a declaration-site property (per struct) not a use-site property (per "object"). Following their static analysis approach is less feasible in Java programs because of the additional problems that must be solved, such as dealing with dynamic dispatch and dynamic code generation.

Unkel and Lam [22] use static analysis on Java benchmarks and open source programs to detect the number of stationary fields. Nelson et. al later study the same property using dynamic analysis [18]. The find the number of stationary fields to be in the range of 55–82% in a variety of programs (the static analysis giving the lower bound and the dynamic analysis giving the upper bound). We measure a stronger property of stationary *objects*, which are objects with only stationary fields. Our findings are in line with

the aforementioned results: 73.4% (median) of all objects are stationary, and 72.6% (median) of all classes only produce stationary objects. This suggests that the stationary fields measured by Nelson et. al tend to cluster, rather than being scattered across all classes.

Chis et al. analyse heap snapshots, focusing on memory bloat in Java programs and identifies common problems that are specific to Java programs [9]. Mitchell et al. [16], summarises heap snapshots in ways that programmers may comprehend with a different goal than ours – to identify memory bloat.

Potanin et al. [20] analyse heap snapshots of Java programs and report among other things on uniqueness (on the heap) and ownership. Their analysis of uniqueness only considers pointers on the heap and finds that 87% of fields are unaliased. Our findings are similar but importantly reached through a different methodology.

8 Conclusion and Future Work

We have presented a trace-based analysis of 9 programs from the DaCapo benchmark suite studying uniqueness, stack-boundedness, and immutability. The ultimate goal of this work is capturing the de-facto properties of real-world programs, with a minimal effort on the behalf of the programmer. Properties such as uniqueness, immutability and their cousin, encapsulation, unlock compile-time and run-time optimisations and can avoid catering to pathological cases that rarely show up in practise, but cannot generally be ruled out.

The results we obtain suggest that a significant amount of static invariants relating to aliasing and immutability exists in unaltered Java programs.

In future work, we want to move the Spencer service forward – by adding more data, implementing more analyses, and doing more analyses like this one. We have preliminary evidence that tracking move semantics might be very interesting – by that, we mean tracking objects that may well be aliased, but where only the *newest* alias is used. A common way to implement unique references in practice (C++, Rust use similar constructs) is to consume them after reading them by setting the read variable or field to `null`, but in unmodified Java programs, explicitly nullifying fields is of course rare. Another research direction is to add static analysis to Spencer. We currently already store all classes that are loaded (even classes that were dynamically generated) in the data base. Mixing static and dynamic analysis techniques would potentially give ranges upper and lower bounds on reported results.

References

[1] Paulo Sérgio Almeida (1997): *Balloon Types: Controlling Sharing of State in Data Types*, pp. 32–59. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/BFb0053373.

[2] Paulo Sérgio Almeida (1997): *Balloon Types: Controlling Sharing of State in Data Types*. In Mehmet Akşit & Satoshi Matsuoka, editors: *ECOOP'97 — Object-Oriented Programming*, *Lecture Notes in Computer Science* 1241, Springer Berlin Heidelberg, pp. 32–59.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage & B. Wiedermann (2006): *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, ACM Press, New York, NY, USA, pp. 169–190, doi:http://doi.acm.org/10.1145/1167473.1167488.

[4] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek & Tobias Wrigstad (2009): *Thorn: robust, concurrent, extensible scripting on the JVM*. In: *ACM SIGPLAN Notices*, 44, ACM, pp. 117–136.

[5] Chandrasekhar Boyapati & Martin Rinard (2001): *A parameterized type system for race-free Java programs*. In: *ACM SIGPLAN Notices*, 36, ACM, pp. 56–69.

[6] John Boyland (2001): *Alias burying: Unique variables without destructive reads*. *Softw., Pract. Exper.* 31(6), pp. 533–553.

[7] John Boyland, James Noble & William Retert (2001): *Capabilities for sharing*. In: *ECOOP 2001—Object-Oriented Programming*, Springer, pp. 2–27.

[8] Stephan Brandauer & Tobias Wrigstad (2017): *Spencer: Interactive Heap Analysis for the Masses*. *under submission*.

[9] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons & John Murphy (2011): *Patterns of Memory Inefficiency*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6813 LNCS, pp. 383–407, doi:10.1007/978-3-642-22655-7_18.

[10] Dave Clarke & Tobias Wrigstad (2003): *External Uniqueness Is Unique Enough*. In Luca Cardelli, editor: *ECOOP 2003 – Object-Oriented Programming*, *Lecture Notes in Computer Science* 2743, Springer Berlin Heidelberg, pp. 176–200.

[11] Dave Clarke, Tobias Wrigstad, Johan Östlund & Einar Johnsen (2008): *Minimal ownership for active objects*. *Programming Languages and Systems*, pp. 139–154.

[12] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing & Andy McNeil (2015): *Deny Capabilities for Safe, Fast Actors*. In: *AGERE15*. Available at `http://www.doc.ic.ac.uk/~scd/fast-cheap-AGERE.pdf`.

[13] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield & Joe Duffy (2012): *Uniqueness and Reference Immutability for Safe Parallelism*. *SIGPLAN Not.* 47(10), pp. 21–40.

[14] Brian Hackett & Alex Aiken (2006): *How is Aliasing Used in Systems Software?* In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, pp. 69–80, doi:10.1145/1181775.1181785.

[15] Philipp Haller & Martin Odersky (2010): *Capabilities for Uniqueness and Borrowing*. *24th European Conference on Object-Oriented Programming (ECOOP 2010)* (June), pp. 354–378, doi:10.1007/978-3-642-14107-2_17.

[16] Nick Mitchell (2006): *The Runtime Structure of Object Ownership*. *ECOOP 2006–Object-Oriented Programming*, pp. 74–98, doi:10.1007/11785477_5.

[17] Nick Mitchell, Edith Schonberg & Gary Sevitsky (2009): *Making Sense of Large Heaps*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5653 LNCS, pp. 77–97, doi:10.1007/978-3-642-03013-0_5.

[18] S Nelson, D J Pearce & J Noble (2013): *Profiling Field Initialisation in Java*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7687 LNCS, pp. 292–307.

[19] Johan Östlund (2016): *Language Constructs for Safe Parallel Programming on Multi-cores*. Ph.D. thesis, Department of Information Technology, Uppsala University.

[20] Alex Potanin, James Noble & Robert Biddle (2004): *Checking Ownership and Confinement*. *Concurrency Computation Practice and Experience* 16(7), pp. 671–687, doi:10.1002/cpe.799.

[21] Alexander J Summers & Peter Mueller (2011): *Freedom before commitment: a lightweight type system for object initialisation*. In: *ACM SIGPLAN Notices*, 46, ACM, pp. 1013–1032.

[22] Christopher Unkel & Monica S. Lam (2008): *Automatic Inference of Stationary Fields: A Generalization of Java's Final Fields*. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, ACM, New York, NY, USA, pp. 183–195, doi:10.1145/1328438.1328463.