Identifying Equivalent Objects to Reduce Memory Consumption

Alejandro Infante, Juan Pablo Sandoval, Alexandre Bergel

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

ABSTRACT

Executing an application may trigger the creation of a large amount of objects. For many applications, a large portion of these objects are unnecessary and their creation could simply be avoided.

We describe a lightweight profiling technique to identity "equivalent" objects. Such equivalent objects are simply redundant and may be shared or reused to reduce the memory footprint. We propose *object-centric execution blueprint*, a visual representation to help practitioners identify cases where objects may be reused instead of being redundant.

1. INTRODUCTION

Garbage collection alleviates the programming activity by delegating the burden of memory deallocation to the virtual machine. The advances of memory models and garbage collectors have significantly reduced the cost of managing memory. Benefits of garbage collection are tremendous: software programs are easier to write and are likely to have less memory-related problems than when written in plain C or C++. However, an excessive use of garbage collection may have a significant impact on the application performance. Creating, initializing and destroying an object consume execution time and memory space.

Current object-oriented programming languages "make it too easy" to create objects. Consider the following code example, inspired by one of our case studies:

```
"Version 1 of Builder"

Builder>>createNode
        ^ GraphicalElement new color: self defaultColor.

Builder>>defaultColor
        "Gray color"
        ^ Color r: 0.5 g: 0.5 b: 0.5
```

Pharo is an object-oriented language and environment for the classic Smalltalk-80 programming language [1].

[1]http://pharobyexample.org

The class `Builder` creates a new colored graphical element when receiving the message `createNode`. The method `defaultColor` creates an instance of the class `Color`. In Pharo, the class `Color` is defined as immutable: `Color` does not provide mutators for its instance variables and any attempt to modify it raises an error. Once instantiated, the value of a color cannot be modified.

The version of the class `Builder` given above is clearly suboptimal since a new color object is associated to each element and all these color objects are equals. In Pharo, a color object weighs 36 bytes.

We found that each ∼18,000 color object creation initiates a garbage collection, thus incurring the inconvenience of garbage collecting the memory (*e.g.,* pause in the program execution, lack of reactiveness in case of CPU intensive activity).

A possible improvement of the `Builder` class may be (difference is indicated in **bold**):

```
"Version 2 of Builder"

Builder>>createNode
        ^ GraphicalElement new color: self defaultColor.

Builder>>defaultColor
        "Gray color"
        defaultColor ifNil: [ defaultColor := Color r: 0.5 g:
      0.5 b: 0.5 ].
        ^ defaultColor
```

In this Version 2, `Builder` is augmented with a new instance variable called `defaultColor`. This variable acts as a memorization cache to keep a unique reference of the default color.

Identifying the necessary changes to move from Version 1 to Version 2 of `Builder` does not present any significant challenge on this contrived example. However, typical object-oriented programs create and destroy a large number of objects. Identifying the objects that are unnecessarily created or destroyed too early presents some challenges[**?**]. Identifying places of redundant object creations is not trivial in many cases. It often requires a deep knowledge of the program intent and implementation. Unfortunately, traditional memory profiling tools do not give any indication about whether objects are redundant or not. As discussed in the related work section, traditional memory analyzers are limited to providing metrics about the heap consumption.

This paper is about a profiling technique to help software engineers identify situations for which reusing or sharing an object is beneficial.

This paper presents a lightweight profiling technique that identifies equivalent objects, intended to be shared to reduce the memory footprint. Our profiler identifies for a given program execution objects that are both non-mutable and are structurally equals. Equality is verified by comparing object snapshots, a kind of hash value that does not rely on the object identity.

Our profiler is accompanied with *Object-Centric Execution Blueprint*, a visual representation of the memory consumption to help practitioners identify sets of equivalent immutable objects that may safely be replaced by one representative shared object. We have successfully used the blueprint to detect and remove a number of redundant objects in a Pharo real world application.

The paper is structured as follows. Section 2 presents our memory profiling in a nutshell. Section 3 describes a case study we have carried out on the Roassal application. Section 4 presents the visual support given to the practitioner to identify critical situations. Section 5 briefly describes the implementation of our profiler. Section 6 gives an overview of the related work. Section 7 concludes and presents future work.

2. IDENTIFYING EQUIVALENT OBJECTS

We propose to optimize applications by identifying groups of equivalent objects. Once identified, a group of equivalent objects may be merged into a unique sharable and reusable object. A definition of object equivalence is provided (Section 2.1) and how such equivalence is measured in an application in the Pharo programming language (Section 2.2).

2.1 Object equivalence

Two objects o_1 and o_2 are said to be *equivalent* if all objects pointing to o_1 may instead point to o_2 without affecting the program semantics and execution. We say that o_1 and o_2 are equivalent if:

(a) **o_1 and o_2 are instances of the same class** – This requirement implies that two objects being from different classes are not interchangeable. This requirement is not strictly necessary, meaning that two objects may be inter-changeable even if they have different classes as long as their interface and contract is similar. However, this requirement significantly simplifies our profiling technique.

(b) **both o_1 and o_2 have identical state** – This requirement implies that each pair of corresponding field values in both objects are either a pair of identical values or a pair of references to objects which are themselves equivalents. For instance, if o_1 := `Point x:5 y:4` and o_2 := `Point x:5 y:4` then o_1 and o_2 have identical state, because their field values in both objects are identical.

(c) **both o_1 and o_2 do not mutate once their construction has completed** – *i.e.,* after the control flow has left the `initialize` method. This implies that side effects are permitted up to the point the object is initialized. If an object changes its state after its creation, such object cannot be equivalent to any other object. In practice, an object is initialized within a factory method located on the metaclass. Examples of such factory methods are `new` and `new:`. Sending the message `new` returns an object supposedly initialized (the method `new` invokes `initialize`). We designate a factory method as a class method returning an instance of the class.

(d) **neither o_1 nor o_2 receive the `identityHash` and `==` message** – It forbids any attempt to access the identity of an object. Receiving a message `identityHash` or `==` makes the object receiver not equivalent to any other object. In Pharo, the identity hash value is a value that reflects an internal number in the virtual machine. The reference equality compares two memory locations.

(e) **Neither the creation of o_1 nor o_2 perform any side effect on the executing context** – It implies that during the creation of an object, side effects are allowed only on the object under creation. Any side effect on another object carried out before exiting a factory method makes the object not equivalent to any other object. Our motivation behind this requirement is that if creating an object performs a side effect, then this creation cannot be avoided else the application behavior is not preserving, even if the object is redundant.

The proposed definition of object equivalence is conservative, meaning that (i) if two objects are equivalent, then one of them is redundant and (ii) two redundant objects are not necessarily equivalent.

This definition is similar to the definition of "mergeability" given by Marinov and O'Callahan [2]. Section 6 detail difference and motivate the need for another definition.

2.2 Profiling

We have built a profiler that identifies groups of equivalent objects. During an execution, our profiler stores in a global table recorded information for each object created in the profiled application. The profiler knows for each object its bit of "history" to determine after the program execution whether that object is equivalent to other objects.

More specifically, our profiler records for each object (i) the number of times it has mutated after having left a factory method and (ii) whether it has received the message `identifyHash`.

In Pharo, everything is an object, and everything happens by sending messages. Nevertheless, certain messages are byte coded by the compiler and no lookup is performed. This is the case of the `==` method. Because of this, detecting when an object receives the message `==` is difficult and, in fact, it is unsolved issue.

After a profiling, it compares all objects and categorizes them in:

- *groups of equivalent objects* – All objects in these groups have the same final state and they did not mutated during the execution after their creation, it means that they had the same state during the execution.
- *groups of near-to-be equivalent objects* – All objects in these groups partially meet our object equivalent definition. We consider that two object are near-to-be equivalent if they do not meet some requirements, for instance, without meeting requirement (c) and (d).

3. CASE STUDY

We have carried out an analysis of the Roassal application and identified a number of situations in which objects have been unnecessarily created.

Roassal. We have analyzed Roassal, an agile visualization engine[2]. Roassal allows one to build sophisticated visualizations, pluggable for any arbitrary domain model. Many objects are involved in a typical Roassal visualization. Each visual element comes with a web of interconnected objects to offer support for interaction and representation.

Excessive use of memory is a barrier from making Roassal scalable: visualizations get slower and less responsive. In addition, by being realized within the virtual machine, the garbage collection overhead does not explicitly appear in a profiling report.

Equivalent objects. Roassal comes with a large amount of tests. The test coverage of Roassal is about 80%, giving us confidence that a fair portion of Roassal features are exercised by unit tests. We have profiled the execution of Roassal unit tests and extracted the following information.

Running Roassal unit tests produces 112,513 objects, instances of Roassal classes. Our profiler has identified that 10.97% of these objects are redundant with the remaining 89.03% of the objects. These 10.97% represents the portion of objects that are unnecessary, and thus the possible gain of the reduction of the object construction.

The largest group of equivalent of objects we have identified is made of instances of the class `RONullShape`. Running the tests of Roassal instantiates this class 13,343 times for which 11,777 objects are equivalent between them. This result means that 11,776 objects are simply unnecessary.

The second largest group is made of all instances of the class `RODraggable`. The 3,027 instances of this class are all equivalent, indicating the need of a singleton pattern.

Improvement of Roassal. We went through some of the group of equivalent objects mentioned in the previous section and refactored Roassal accordingly. We have reduced the amount of created objects by 5.1%. The total amount of objects went from 112,513 to 106,806. This 5.1% of reduction represents a gain of 45Kb approximately, leading to a reduction of 1.4% of the memory consumption.

We have refactored Roassal by implementing singleton patterns on various classes. The class `RODraggable` has been refactored as follows:

```
RODraggable class>>elementToBeAdded
   instance ifNil: [ instance := self new ].
   ^ instance
```

The singleton pattern is implemented in **bold**.

4. VISUAL SUPPORT

Object-centric execution blueprint is a visual aid to identify groups of equivalent (and therefore redundant) objects. We use a polymetric view [1] for that purpose, since we relate different metrics for each structural visual element.

Our blueprint is made up of colored boxes and inner boxes and links (Figure 1). Such visual representation of the program execution is obtained after the completion of the execution.

Nesting outer boxes represents classes. Inner boxes represent groups of objects that are either equivalent or near-to-be equivalent (*i.e.,* without meeting requirement (c) and (d), about the mutation).

[2]`http://objectprofile.com/#/pages/products/roassal/overview.html`

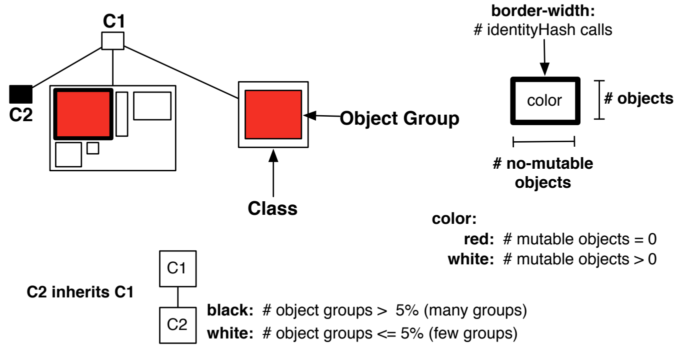
C1
C2
Object Group
Class
border-width:
# identityHash calls
color
# objects
# no-mutable objects
color:
red: # mutable objects = 0
white: # mutable objects > 0
C2 inherits C1
C1
C2
black: # object groups > 5% (many groups)
white: # object groups <= 5% (few groups)

Figure 1: Object-Centric Execution Blueprint

Each object group is characterized with two metrics and two properties:

- height is the amount of (mutable and immutable) objects that belong to the group, using a logarithmic scale;
- width is the amount of immutable objects belonging to the group, using a logarithmic scale;
- presence of a bold border indicates `identityHash` has been invoked on any object of the object group;
- color can be red or white, red if all the objects in the object group do not mutate and white if at least one of them mutate.

An object group painted in red visually indicates that all the objects belonging to this group are equivalent. A white group indicates that the objects are near-to-be equivalent. Such groups may require some further action from the software engineer to make these objects equivalent.

It frequently happens that instances of a class are heterogeneous, which may result in many different groups. Such situations are discarded from the visualization by using a threshold number of groups. In our experiment, we consider a threshold of 5, meaning that groups of a class are shown if at most 5 groups for 100 instances. The purpose of this arbitrary heuristic is to reduce the amount of data that would be difficult to improve.

Figure 1 shows that `C1` has three subclasses. Each of them tells a different story and we need to deal with them in different ways.

From left to right, class `C2` is black filled, meaning that instances of `C2` cannot be grouped into equivalent objects.

The class in the middle has 5 object groups inside, which means that in this experiment the instances of this class can only have 5 possible states. Furthermore, one of these groups is colored red, indicating no mutation occurs for that group. Objects belonging to that group would have been equivalent if they had a non bolded border: the message `identityHash` is sent to the objects of this group.

The last class of the figure shows a single red object group, so all the instances of this class do not mutate and have the same state across all the execution, also nobody called `identityHash` on them. This is an excellent opportunity to use the singleton pattern and reuse a single object to fulfill the job of all the previously used objects.

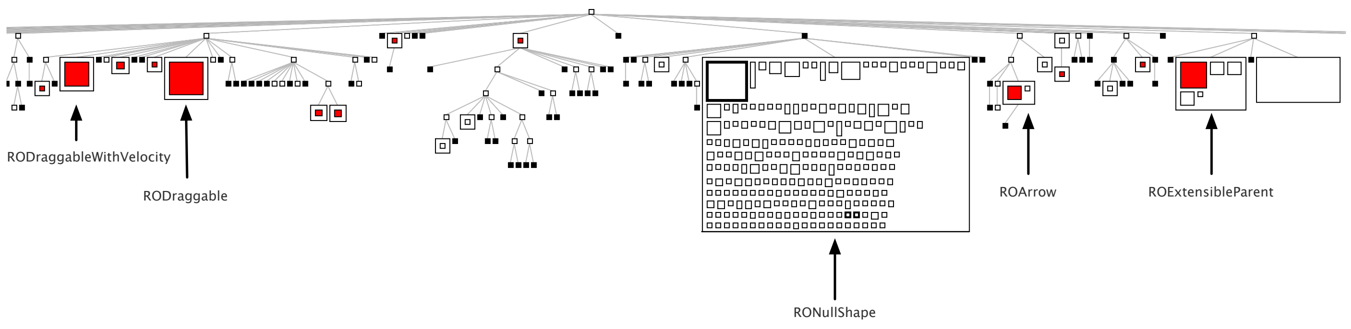

Figure 2: Object-Centric Execution Blueprint before optimization

Also the visualization provides some interactions to provide the user the opportunity go deep into the experiment. Just moving the mouse over a class displays as a tooltip its class name, the number of instances and the memory consumed by its instances. Moving over an object group displays the number of objects, the amount of mutable and non-mutable objects. Furthermore the tooltip displays whether any object in the group caused a side effect on its creation, a requirement settled before reusing the objects.

Clicking the shapes also allows the user to inspect some objects or browse some classes.

Figure 2 shows an excerpt of the visualization for the unit test execution of Roassal.

5. IMPLEMENTATION

We briefly describe the two key ingredients to implement our approach. Our Memory profiler is available under the MIT License[3].

5.1 Gadget profiling

Gadget Profiler[4] is a framework for method instrumentation. It allows the programmer to inject code before and after every method of an automatically set of selected classes. Also it provides some essential information about the execution to be used by the injected code, like the receiver and the arguments of the message.

Memory Profiler is built using Gadget Profiler. Also when a method is called we check if the method is the `identityHash`, performs a mutation or causes an external side effect. Finally, it groups the objects as we described before.

5.2 Snapshotting objects

Keeping track of the side effects may be done in a number of fashions (*e.g.,* keeping track of the write bytecodes, modifying the abstract syntax tree [3]). We employ here a technique based on object snapshotting. We define an *object snapshot* as an integer that represents the complete state of an object. This integer is computed using a *bitXor* operation between the identity hash of attributes and the identity hash of object class.

```
Object>>snapshotAsInteger
   | index value |
   index := self class instSize.
   value := self class gadgetIdentityHash.
   [index > 0]
      whileTrue:
         [ value := (value bitShift: 1) bitXor: (self instVarAt:
      index) gadgetIdentityHash.
            index := index − 1].
   ^ value
```

[3]`http://smalltalkhub.com/#!/~ainfante/MemoryProfiler`

[4]`http://smalltalkhub.com/#!/~ainfante/GadgetProfiler`

An object snapshot is useful to compare objects states. Comparing objects states we can detect: (i) objects that have the equivalent state, and (ii) if an object has a different state after a method execution. Both features are essential to detect equivalent objects.

6. RELATED WORK

Marinov *et al.* presented Object Equality Profiling (OEP), a profiling technique to discover opportunities for replacing a set of equivalent object instances with a single representative object [2].

Their tool performs a dynamic analysis that records all the objects created during a particular program run. The tool partitions the objects into equivalence classes, and uses collected detail timing information to determine when elements of an equivalence class could have been safely collapsed into a single object. They use an instrumentation byte code technique to record fine-grained information. They insert instrumentation at the following program points: allocation sited for objects and arrays, field writes, array element writes, field reads, array element reads among others. Adding a considerable overhead.

Our object snapshot technique takes snapshots before and after a method execution, and saves the last state of the objects (the last snapshot) causing a significantly lower execution overhead. And having a trade-off between overhead and accuracy. We also propose *object-centric execution blueprint* as a visual aid to detect, understand, and delete redundant object.

7. CONCLUSION & FUTURE WORK

Currently, the large majority of code profilers and debuggers use inadequate abstractions in their analysis. We believe this is a critical situation and hope the tool and ideas presented in this paper will contribute to addressing it.

Thanks to the analysis above, we eliminated more than 5.1% of the identified unnecessary objects refactoring code using singleton pattern, but working on the analysis and the abstraction we expect to categorize possible source code refactoring to eliminate the totality of redundant objects.

Acknowledgements

Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT and AGCI, Chile. CONICYT-PCHA/ Doctorado Nacional/2013-63130199. This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile, and FONDECYT project 1120094.

8. REFERENCES

[1] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[2] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 313–325, New York, NY, USA, 2003. ACM.

[3] Jorge Ressia. *Object-Centric Reflection*. Phd thesis, University of Bern, October 2012.