A Tool for Ownership and Confinement Analysis of the Java Object Graph

Alex Potanin
Victoria University of Wellington
PO Box 600
Wellington, New Zealand
alex@mcs.vuw.ac.nz

ABSTRACT

This poster presents a tool for the analysis of Java heap snapshots. The tool supports a flexible query language to measure various aspects of the object graph related to the studies of uniqueness, ownership, encapsulation and confinement. One of the applications of our tool was the verification of the power law dependency in the distribution of incoming and outgoing references to objects.

Keywords

Java, object graph, ownership, confinement, power law

1. INTRODUCTION

In this poster, we present a software tool written in Java called Fox. We use it to measure several aspects of the ownership tree and the object graph. In particular, we examine the average depth of each node in the ownership tree, object confinement, and distribution of incoming and outgoing references. One of the highlights of our measurements is the verification of the power law in the distribution of the number of incoming or outgoing references. We use the results obtained using our tool to compare with the relevant work in the field and to detect interesting points that only come up through empirical studies.

2. THE TOOL

The Fox is written in Java and it works with heap dumps produced using the Heap Profiler (HPROF) [10] library that comes with Sun's JDK 1.2 or higher. After loading the heap dump using the Heap Analysis Tool (HAT) [9] library, it constructs an ownership tree following a well-known dominator algorithm [2]. The user is presented with a window in which they can edit and execute the queries that they would like to execute on a given dump or a collection of dumps.

3. MEASUREMENTS

Our measurements were performed upon a corpus of major Java programs, more than half of which came from the Purdue Benchmark Suite [5]. Our results can be broken down into approximately four categories: uniqueness, ownership [4], confinement [5], and incoming and outgoing references distribution. The 52 heap dumps from a corpus of around 30 programs range in sizes from 3,500 to 350,000 objects.

An important point about the Fox is that it empirically evaluates encapsulation based on a heap snapshot of a running Java program, as opposed to the static source code or dynamic behavior traces. We chose this method because we aim to analyze the encapsulation latent within a corpus of programs, rather than benchmark a particular algorithm or alias protection scheme. Compared to static confinement checkers like Kacheck/J [5] our approach is optimistic: an object may be identified as unique, confined, or otherwise encapsulated, depending on the state of the object graph at the instant a heap dump is taken. Thus, our results overestimate the encapsulation data: an object which is encapsulated at one instant (or in one program run) may be aliased the next instant, or in the next run. On the other hand, more conservative approaches such as Kacheck/J cannot identify objects which are effectively encapsulated in practice unless they can prove the objects will be encapsulated across all potential program executions, most of which will never occur.

4. UNIQUENESS

Uniqueness is the most basic type of aliasing control: a unique object is encapsulated within its sole referring object [3]. We analyzed the object graphs to determine the number of non-unique objects, in particular, the percentage of objects with more than one incoming reference. We found that on average only 13% of all objects had more than one object pointing at them, or in other words had an alias. This means that the number of aliases in the systems we looked at was not too high. One of the interesting points that we will graphically demonstrate in the poster is the rise in the percentage of aliased objects as we move through the dumps taken at the later stages of the program run.

5. OWNERSHIP

Object ownership is in essence a generalization of uniqueness that underlies many more sophisticated alias management schemes. An object a owns another object b if all the paths from the root r to the object b go through a. In this case b is called the *owner* of a. The implication of ownership is that no object outside the owner b is allowed to have a reference to a. We posit a global root r through which all the objects in the root set of the current heap snapshot can be accessed. Ownership allows us to structure an object graph into an implicit ownership tree.

Our primary metric of object ownership is the average depth of an object in the ownership tree, that is, the average number of levels of encapsulation around any object. In most programs in the corpus, this was around 5 or 6; however some large programs had substantially larger values (e.g. 817.25 for BlueJ under heavy load). Given these figures, we hypothesized that large data structures such as linked lists could have a significant effect when calculating the average depth of the node in the ownership tree: the average depth of a list node would be half the length of the list. To address this issue, we decided to fold up the reference paths by counting chains of objects of the same class as having the length of 1. This gave us a less biased account, with the average depth after folding being around 5.47 as opposed to 42.77 across all programs, with the large outlying depths being greatly reduced (e.g. a simple linked list test program has average depth of 142.37 and the average depth after folding of 4.21). The resulting ownership metric does demonstrate, however, that there is a significant amount of object-based encapsulation in Java programs.

We also calculated the average ownership tree depth per class; this gave us a finer-grained picture of object ownership but the results may not be presented in the poster for space reasons. As an example of the average depth from root per class consider `java.lang.Hashtable`. Since most of its instances should own `java.util.Hashtable$Entry[]` array instance, we would expect the average depth of the latter to be one greater than the former. The following sample of the Fox output for **BlueJ** confirms this hypothesis, and also implies the actual hashtable entries are contained within the entry array.

```
Class java.util.Hashtable has 279 instances and
on average each instance node is 3.96 deep.
Class java.util.Hashtable$Entry[]; has 297 instances and
on average each instance node is 4.90 deep.
Class java.util.Hashtable$Entry has 3990 instances and
on average each instance node is 5.37 deep.
```

6. CONFINEMENT

Object confinement is an instantaneous approach to class confinement. If all the referrers to an object are in the same package, then we call such an object *strongly confined*. If all the referrers to an object are in the same top-level package, we call it *weakly confined*. Otherwise, we call the object *not confined*.

Our measurements have shown that around 46% of the objects were not confined, 21% were weakly confined, and 33% were strongly confined. At first glance, these numbers are roughly what static analysis by Grothoff, Palsberg, and Vitek [5] leads us to expect, even though we are looking from a different perspective.

7. POWER LAWS

We examined five large dumps of our corpus, in a similar manner to Clark and Green with respect to the list structures in Lisp [6]. For each of these large dumps we produced a graph of the log of the frequency rank versus the log of the number of occurrences of the objects having a corresponding rank. All five graphs for incoming reference and outgoing reference distributions demonstrated a linear or close to linear dependence, thus verifying the presence of Zipf's or Power Law. The latter was first observed in the distribution of words in any given novel: the nth most common word occurs about $\frac{1}{n}$ times as frequently as the most common word [8].

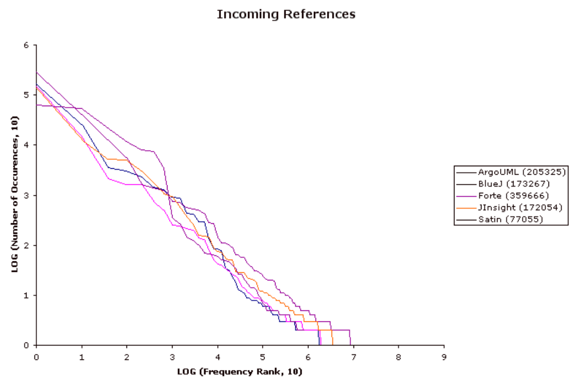

Figure 1: Distribution of the Number of Incoming References versus the Number of Objects on LOG vs. LOG scale

8. CONCLUSION

In the poster, we would like to graphically illustrate our results on the tool development, query language design, ownership and confinement metrics, and the power laws. We hope that our work will encourage people's interest in the study of aliasing and empirical measurements of the object inter-relationships, and we would like to hear from people interested in the relevant fields who may wish to use our work in their research for the advancement of object-oriented software engineering.

9. REFERENCES

[1] Raimondas Lencevicius, *Advanced Debugging Methods*, Kluwer Academic Publishers, August 2000.
[2] Thomas Lengauger and Robert Endre Tarjan, A Fast Algorithm for Finding Dominators in a Flowgraph, *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, July 1979, Pages 121-141.
[3] John Boyland, James Noble, and William Retert, Capabilities for Sharing, *Proceedings of ECOOP'01*, Springer-Verlag, 2001.
[4] David Clarke, John Potter, and James Noble, Ownership Types for Flexible Alias Protection, *Proceedings of OOPSLA'98*, ACM Press, 1998.
[5] Christian Grothoff, Jens Palsberg, and Jan Vitek, Encapsulating Objects with Confined Types, *Proceedings of OOPSLA'01*, ACM Press, 2001.
[6] Douglas W. Clark and C. Cordell Green, An Empirical Study of List Structure in Lisp, *Communications of ACM*, February 1977, Vol. 20, No. 2.
[7] Alex Potanin and James Noble, Checking Ownership and Confinement, *ECOOP'02 Workshop on Formal Techniques for Java-like Programs*, 2002.
[8] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Reading, Mass., 1949.
[9] Java Heap Analysis Tool by Bill Foote, from *http://java.sun.com/people/billf/heap/index.html*.
[10] Java Virtual Machine Profiler Interface, from *http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html*.