Flyweight ASTs: A Study in Applied Lazyness

Bachelor's Thesis

Fabian Bösiger
Supervised by Dr. Malte Schwerhoff

Programming Methodology Group
Department of Computer Science
ETH Zürich

April 17, 2021

Abstract

Acknowledgements

I would like to thank my supervisor, Malte Schwerhoff, who provided me with the opportunity to write this thesis. I am very grateful for the time and effort he expended.

I would also like to thank Peter Müller for leading the Programming Methodology Group, which always appears in an open and welcoming way.

Finally, I'd like to express my gratitude towards my family for providing me with a very pleasant home office environment.

Contents

1 Introduction

1.1 Implementation of Flyweight ASTs

Abstract syntax trees (AST) are used in compilers and similar programs to represent the structure of a program as a tree data structure. As with any other tree structure, ASTs can be traversed, searched, transformed and so forth. During such operations, subtrees within the AST are potentially checked for equality many times. Moreover, equality checks also occur in operations on collections of AST subtrees, for example in finding a specific subtree, which may add additional performance overhead.

Equality checks can't be avoided, but they can be implemented in a more performant way. Currently in Silicon, new term instances are created independently of already existing ones, which potentially leads to the coexistence of multiple structurally equal term instances. Subterm equality is checked in a structural and recursive manner.

Listing 1: Simplification of how term instances currently are implemented. Because `Plus` is defined as a case class, the compiler automatically generates code for recursive structural equality checking.

```
1 case class Plus(val p0: Term, val p1: Term) extends Term
```

Because the AST used in Silicon is immutable, the flyweight pattern [**patterns**] can be applied on AST terms. To do this, pool of term instances is maintained. Whenever a term is to be created, the components of this new term is compared with the pool of existing terms. If a term with the same components already exists, it is returned and the creation of a new instance of this term is avoided. Otherwise, a new term is created and added to the pool.

This gives the guarantee that there are no two instances of the same term in our pool, meaning every two structurally equal terms point to the same underlying object in memory. Comparing terms for structural equality then boils down to a cheap reference equality check, and recursive equality checks can be avoided.

Listing 2: Avoid instantiating multiple structurally equal terms using the flyweight pattern.

```
1  class Plus private (left: Term, right: Term) extends Term {
2      // ...
3  }
4
5  object Plus extends ((Term, Term) => Plus) {
6      // Pool object which holds our "Plus" terms.
7      var pool = new HashMap[(Term, Term), Plus]
8
9      def apply(e0: Term, e1: Term) = {
10         pool.get((e0, e1)) match {
11             // If this term already exists in pool, return it.
12             case Some(term) => term
13             // Otherwise, create a new instance.
14             case None =>
15                 val term = new Plus(e0, e1)
16                 pool.addOne((e0, e1), term)
17                 term
18         }
19     }
20
21     // ...
22 }
```

The `Plus` constructor is now private, which makes it impossible to create `Plus` instances without checking the pool first. The companion object `Plus` now contains a pool of all existing `Plus` instances. Furthermore, `Plus` is no longer a case class, which means that the default `equals` method no longer recursively checks for structural equality, but instead simply does a reference equality check. Because Scala's equality operator (==) and data structures such as `HashMap` use the `equals` method behind the scenes, this will most likely lead to performance improvements.

1.2 Automate Boilerplate Generation using Macros

Silicon's AST representation of the Viper language consists of nearly 100 different terms, all with boilerplate implementations for different operations. Our changes introduce additional boilerplate code to each term companion object, as seen in the case of the `Plus` term in listing 2.

Our ASTs shouldn't only be flyweight in the sense of the implementation pattern, but also regarding development time and effort. This is why we want to avoid such boilerplate code and instead automatically generate companion objects seen in listing 2 using Scala's macro annotations. Additional benefits of using macro annotations include improvements in code readability and maintainability. Experimenting with code changes will become a matter of editing a single macro instead of editing each term individually. Terms which may be added in the future are easier to implement.

2 Implementation

2.1 A Macro Annotation for Code Generation

2.2 Classes or Case Classes

2.3 Macro Annotations on Nodes

2.4 Clear Pools after each File

2.5 AST Reduction for Builtin Equals

2.6 Equality Defining Members

3 Evaluation

3.1 Verification Time

Usually, the flyweight pattern is

The flyweight pattern can be applied effectively if two main points hold. First, the application uses large numbers of objects and causes high storage costs. [**patterns**] This is true for ASTs as they can grow large containing a large amount of objects.

Second, groups of objects may be replaced by few shared objects once ex-

trinsic state is removed. [**patterns**] In ASTs, many nodes have no state at all, such as `True`, `False`. They are well suited as flyweight objects. For non-leave nodes, extrinsic state cannot be externalized, for example the children of a `Plus(t1, t2)` node.

3.2 Memory Consumption

3.3 Automatic Code Generation