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

March 22, 2021

Abstract

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.

Because ASTs represent the structure of programs, equality checks usually happen on a structural level. Structural equality checking of subtrees is done recursively as we have to compare not only the roots, but also their children. This may add unexpected performance costs, especially considering the depth of ASTs representing big programs.

Many operations on ASTs, including equality checks, have to be implemented for every AST node type individually. But these implementations usually follow a simple pattern and only differ in the arity and type of the node's parameters. Such repetitive code is generally known as boilerplate code and should be reduced to a minimum, for example with automatic code generation using metaprogramming techniques.

The goal of this thesis is to improve the performance of AST subtree equality checks without burdening developers with additional boilerplate implementations.

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 Viper and Silicon

Viper [3] is a verification infrastructure on top of which verification tools for different programming languages can be built. Silicon [4] is a backend for Viper, which is based on symbolic execution. To advance program verification in practice, fast verification is crucial as it provides a more streamlined experience for developers. This is the reason why one of Silicon's stated goals is performance:

> "The verifier should enable an IDE-like experience: it should be sufficiently fast such that users can continuously work on verifying programs [...]" [4]

Silicon internally builds an AST from the Viper input program. This AST is potentially checked for structural equality many times within the execution of Silicon.

Listing 1: Example of multiple subtree ("term") equality checks occuring during the execution of Silicon.

```
// args is of type Seq[Term], meaning we compare a list of
// terms within a sort function which is called many times
// during the sorting process.
relevantChunks.sortWith((ch1, ch2) => {
        // ... &&
        ch1.args == args
})
```

1.2 Scala and IntelliJ

The Viper infrastructure is written in the Scala programming language. Scala has support for metaprogramming using macros. For a nice programming experience using macros, IDE support should ideally be provided. In this case, we use the IntelliJ IDE. However, Scala macros are not supported natively by the IntelliJ IDE:

> "Since IntelliJ IDEA's coding assistance is based on static code analysis, the IDE is not aware of AST changes, and can't provide appropriate code completion and inspections for the generated code." [2]

To ensure support for macros, a macro-specific IntelliJ plugin would have to be developed such that IntelliJ properly picks up macro-generated code.

2 Approach

2.1 Implementation of Flyweight ASTs

In a first step, we evaluate the performance of the Silicon backend, which allows us to relate possible performance improvements to changes made during this project. We use existing benchmarking infrastructure and, if necessary, create new benchmarks and benchmarking tools.

We can't avoid equality checks themselves, but what we can do is implementing equality checks 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 2: 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, we can utilize the flyweight pattern [1] on AST terms. To do this, we maintain a pool of term instances. Whenever a term is to be created, we first compare the components of this new term with our pool of existing terms. If a term with the same components already exists, we return it and avoid creating a new instance of this term. Otherwise, we create a new term and add it to the pool.

This gives us 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 3: Avoid instantiating multiple structurally equal terms using the flyweight pattern.

```
class Plus private (left: Term, right: Term) extends Term {
    // ...
}

object Plus extends ((Term, Term) => Plus) {
    // Pool object which holds our "Plus" terms.
    var pool = new HashMap[(Term, Term), Plus]

    def apply(e0: Term, e1: Term) = {
        pool.get((e0, e1)) match {
            // If this term already exists in pool, return it.
            case Some(term) => term
            // Otherwise, create a new instance.
            case None =>
                val term = new Plus(e0, e1)
                pool.addOne((e0, e1), term)
                term
        }
    }

    // ...
}
```

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.

2.2 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 3.

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 3 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.

Listing 4: One possible way to use macro annotations to automatically generate code in listing 3.

```
1 @memoizing
2 case class Plus(left: Term, right: Term) extends Term
```

3 Implementation

3.1 Everything about Flyweight ASTs

3.1.1 One for All or Each on Their Own

3.1.2 Memory Management

3.1.3 Concurrency

3.2 Implementing the Macro Annotation

3.2.1 Hygiene

3.3 Support for the IntelliJ IDE

4 Evaluation

4.1 Verification Time

4.2 Memory Consumption

References

[1] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. en. 2016. ISBN: 978-0201633610.

[2] *IntelliJ API to Build Scala Macros Support*. `https://blog.jetbrains.com/scala/2015/10/14/intellij-api-to-build-scala-macros-support/`. Accessed: March 3, 2021.

[3] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A verification infrastructure for permission-based reasoning". en. In: *Dependable Software Systems Engineering*. Ed. by Alexander Pretschner, Doron Peled, and Thomas Hutzelmann. Vol. 50. Amsterdam: IOS Press

BV, 2017, pp. 104–125. ISBN: 978-1-61499-809-9. DOI: 10.3233/978-1-61499-810-5-104.

[4] Malte H. Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". en. PhD thesis. Zürich: ETH Zurich, 2016. DOI: 10.3929/ethz-a-010835519.