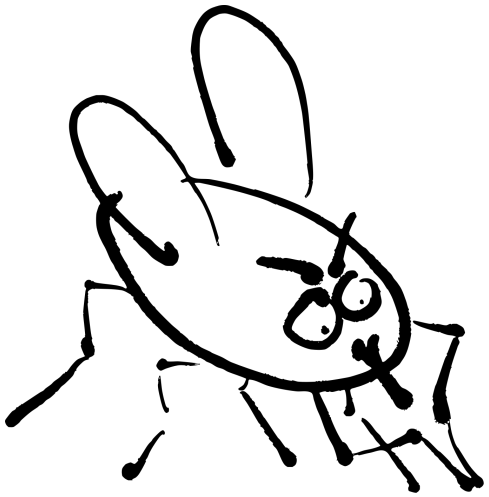
# Flyweight ASTs

*Bachelor's Project Description*



Fabian Bösiger
Supervised by Dr. Malte Schwerhoff

March 3, 2021

# 1 Introduction

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, sorted and so forth. During such operations, subtrees within the AST are potentially checked for equality many times.

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 representig big programs. Moreover, equality checks also occur in operations on collections of AST subtrees, which may add an additional performance overhead.

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

# 2 Background

## 2.1 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 cruicial 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, and as such has the properties described in section 1.

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
})
```

### 2.2 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.

## 3 Approach

### 3.1 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.

Listing 2: Simplification of how term instances currently are created when applying a term.

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

// Companion object to the "Plus" case class.
object Plus {
    // The apply method Allows us to use the notation
    // "Plus(e0, e1)" instead of "new Plus(e0, e1)".
    def apply(e0: Term, e1: Term) = {
        // ...
        new Plus(e0, e1)
    }
}
```

We can't avoid equality checks themselves, but what we can do is implementing equality checks in a more performant way. Currently in Silicon, every time a term is applied, a new instance of this term is created. Subterm equality is checked in a recursive manner.

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 applied, we first compare the new term that is to be created with our pool of existing terms. If a structurally equal term already exists, we avoid creating a new instance of this term and instead return a reference to the equal object in the pool.

This gives us the guarantee that there are no two instances of the same term in our pool, i.e. every two structurally equal terms point to the same underlying object in memory. Comparing terms for structural equality 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 {
    // ...
}
```

```
4
5  object Plus extends ((Term, Term) => Term) {
6      // Pool object which holds our "Plus" terms.
7      val pool = new HashMap[(Term, Term), Term]
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     // ...
23 }
```

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 Scalas equality operator (==) and datastructures like `HashMap` use the `equals` method behind the scenes, this will most likely lead to performance improvements.

### 3.2 Automate Boilerplate Generation using Macros

Silicons AST representation of the Viper language consists of nearly 100 different terms. Many of these terms are case classes using compiler-generated `equals` and `apply` methods. Others are classes with their own companion objects manually defined, including custom `equals` and `apply` methods. Additionally, our changes introduce more boilerplate code that has to be written manually for every term.

Listing 4: The `apply` method of the `Plus` class additionally defines AST reductions. Together with our planned changes the amount boilerplate code will increase significantly.

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


object Plus extends ((Term, Term) => Term) {
    import predef.Zero

    val pool = new HashMap[(Term, Term), Term]

    def apply(e0: Term, e1: Term) = {
        (e0, e1) match {
            case (t0, Zero) => t0
            case (Zero, t1) => t1
            case (IntLiteral(n0), IntLiteral(n1)) =>
                IntLiteral(n0 + n1)
            case _ => pool.get((e0, e1)) match {
                case Some(term) => term
                case None => {
                    val term = new Plus(e0, e1)
                    pool.addOne((e0, e1), term)
                    term
                }
            }
        }
    }

    // ...
}
```

Our ASTs shouldn't only be flyweight in the sense of the development pattern, but also regarding development time and effort. This is why we want to avoid such boilerplate code and instead automatically generate companion objects using Scalas macro annotations. Additional benefits of using macro annotations include improvements in code readability and maintainability. Terms which may be added in the future are easier to implement.

Listing 5: One possible way to use macro annotations to automatically generate code in listing 4.

```
@memoizing
case class Plus(left: Term, right: Term) extends Term {
    // The reduce method may be used by the macro to generate
    // AST reductions.
    def reduce(e0: Term, e1: Term) = (e0, e1) match {
        case (t0, Zero) => Some(t0)
        case (Zero, t1) => Some(t1)
        case (IntLiteral(n0), IntLiteral(n1)) =>
            Some(IntLiteral(n0 + n1))
        case _ => None
    }
}
```

# 4 Goals

## 4.1 Core Goals

1. **Research other potential solutions** for fast subtree equality checking, in the context of Scala and in general, that may have been tried for similar problems.

2. **Implement the solution approach** described in section 3.1 without the use of macros. This allows for a first evaluation of performance before auto-generating code using macro annotations.

3. **Evaluate performance improvements** and discuss the impact of our changes on the time needed for verification.

4. **Develop macro annotations** for automatic code generation as described in section 3.2.

5. **Ensure IntelliJ IDE support** for our macro annotations as discussed in section 2.2.

### 4.2 Extension Goals

1. **Profiling** parts of Silicon that perform many operations of terms. Possibly adapt these parts to better utilize performance improvements regarding equality checks.

2. **Evaluate performance improvements** that may be achieved in the previous step.

3. **Research ways to extend AST reductions** seen in section 3.2 as a furter step to increase performance. Usage of a DSL in combination with Scala macros to auto-generate AST simplifications. This would allow to easily add or modify AST reductions in the future.

4. **Implement better AST simplifications** found in the previous step.

5. **Evaluate performance improvements** that may be achieved in the previous step.

## 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`.