Avoiding Structural Equality Checks on Immutable Syntax Trees

Fabian Bösiger
Supervised by Dr. Malte Schwerhoff

February 24, 2021

1 Introduction

Viper [1] is a verification infrastructure on top of which verification tools for different programming languages can be built. Silicon [2] is a backend for Viper, which is based on symbolic execution.

To advance program verification in practice, fast verification is cruicial as it profides a more streamlined experience for developers. This is the reason why one of Silicons 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 [...] [2]

Because Silicon is a shared backend for many verification frontends, faster verification in Silicon is directly beneficial for the developer using such tools like Gobra for Go, Nagini for Python or Prusti for Rust.

Silicon internally builds an abstract syntax tree (AST) from the Viper input program. This AST is checked for structural equality multiple times within the execution of Silicon. Currently, Silicon recursively checks for structural equality of ASTs. For big ASTs and many equality checks, this process of becomes very inefficient.

```
1 TODO
2 class Plus(val p0: Term, val p1: Term) extends Term
3
4 object Plus {
5     def apply(e0: Term, e1: Term) = new Plus(e0, e1)
6 }
```

Listing 1: Example of a term equality check.

We can't avoid equality checks themselves, but what we can do is implementing equality checks in a more performant way. This is the core topic of this thesis.

Currently, every time a term is applied, a new instance of this term is created. This happens even if we apply the same term twice, leaving us with no other choice than checking equality structurally and recursively.

```
1 class Plus(val p0: Term, val p1: Term) extends Term
2
3 object Plus {
4     def apply(e0: Term, e1: Term) = new Plus(e0, e1)
5 }
```

Listing 2: How term instances currently are created.

We see that everytime `apply` is called, a new instance of `class Plus` is created. We want to avoid creating a new instance on every call to `apply` by using the flyweight pattern.

2 Approach

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

The general idea is to utilize the flyweight pattern to the AST which is immutable. To do this, we maintain a pool of pairwise structurally unequal terms. 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 equivalent term already exists, we avoid creating a new instance of this term and instead return a reference to the equivalent object in the pool.

This gives us the guarantee that there are no two instances of the same term, i.e. every two structurally equivalent terms point to the same underlying object in memory. Comparing terms for structural equality boils down to a cheap pointer comparison, recursive equality checks can be avoided.

```
1  class Plus(val p0: Term, val p1: Term) extends Term
2
3  object Plus {
4      def apply(e0: Term, e1: Term) = {
5          val plus = new Plus(e0, e1)
6          if (pool.contains(plus))
7              pool.get(plus)
8          else
9              plus
10     }
11 }
```

Listing 3: How a memory pool may be used to avoid creating structurally equivalent terms.

The Scala compiler automatically generates the `equals` method for case classes, where the default implementation recursively checks each field for equality. To avoid this process of recursive equality checking, we instead override the default `equals` method to do a simple reference equality check.

```
1  sealed trait Term extends Node {
2      // Override the default equals method:
```

```
3       override def equals(other: Any) =
4           this.eq(other.asInstanceOf[AnyRef])
5       // ...
6   }
```

Listing 4: We override the the default equals method to do a simple reference equality check.

3 Goals

3.1 Core Goals

1. **Research other potential solutions** that may have been tried for similar problems. The general problem of recursive equality checks may have been solved in the past.
   1 Week

2. **Implement the solution approach** described in section 1 without the use of macros. This allows for a first evaluation of performance before auto-generating code using macro annotations.
   3 Weeks

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

4. **Build macro annotations** for automatic code generation. This allows for easy implementation of terms that may be needed in the future. Macro annotations also help keeping the codebase well-arranged.
   4 Weeks

5. **Add IDE support** for our macro annotations. Because Scala macros are still in an experimental phase, we make sure that IntelliJ picks up code generated by our macro annotations.
   4 Weeks

3.2 Extension Goals

1. **Utilizing better suited data structures** to possibly achieve further performance improvments by in Silicon. If the performance of equality checks was improved in the previous steps, some data structures may become preferrable performance-wise, e.g. hashmaps become more performant over a simple iteration over a list.

2. **Evaluate further performance improvments** that may be achieved in the previous step.
3. **Research ways to extend AST simplifications** as a furter step to increase performance. Not only local simplifications, but also global simplifications are imaginable.
4. **Implement better AST simplifications** found in the previous step.
5. **Evaluate further performance improvments** that may be achieved in the previous step.
6. **Use a DSL** in combination with Scala macros to auto-generate AST simplifications. This would allow to easily add or modify AST simplifications in the future.

References

[1] Peter Müller, Malte Schwerhoff, and Alexander Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: (2016). DOI: 10.1007/978-3-662-49122-5_2.

[2] Malte Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". In: (2016).