Flyweight ASTs

Bachelor's Project Description

Fabian Bösiger
Supervised by Dr. Malte Schwerhoff

March 2, 2021

1 Introduction

Abstract syntax trees (AST) are used in compilers and similar programs to represent the structure of a program as a tree datastructure. When working with ASTs, subtrees are potentially checked for equality many times.

Structural equality checks usually happen recursively, which may add unexpected performance costs, especially considering the depth of ASTs. Moreover, equality checks also occur in operations on collections of AST subtrees, which may add additional performance overhead. The goal of this project is to find a way to increase performance of AST subtree equality checks.

2 Background

Viper [2] is a verification infrastructure on top of which verification tools for different programming languages can be built. Silicon [3] 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 [...] [3]

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. Silicons implementation recursively checks for structural equality of ASTs, which becomes more inefficient with increasing depth of the AST.

Listing 1: Example of multiple subtree ("term") equality checks occuring in Silicon's exhale supporter.

```
1 relevantChunks.sortWith((ch1, ch2) => {
2     definiteAlias.contains(ch1) || !definiteAlias.contains(ch2) &&
3         // args is of type Seq[Term]
4         ch1.args == args
5 })
```

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. This happens even if we apply the same terms multiple times, leaving us with no other choice than checking structural equality in a recursive manner using the compiler-generated `equals` method.

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

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

object Plus {
    def apply(e0: Term, e1: Term) = new Plus(e0, e1)
}
```

3 Approach

In a first step, we test 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.

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 shared 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 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.

```
// Pool object which holds our terms.
object Pool {
```

```
  import scala.collection.mutable.HashMap

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

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

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

    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
            }
        }
    }
}
```

Scalas equality operator (==) and datastructures like `HashMap` use the `equals` method behind the scenes. Because the default implementation for `equals` in case classes recursively checks each field for equality, we, instead override the `equals` method to do a simple reference equality. This will likely lead to performance improvements, which we compare to the benchmark results before our changes.

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

```
class Plus private (left: Term, right: Term) extends Term {
    // Override the equals method to do a reference equality check.
    override def equals(other: Any) =
```

```
        this.eq(other.asInstanceOf[AnyRef])
}
```

In a next step, because we can no longer use the compiler-generated `eqauls` method for case classes, we instead automatically generate boilerplate code as much as possible using Scalas macro annotations. This not only improves readability, but also makes it easier to add new terms in the future and thus improves maintainability. Because IntelliJ IDE support for macro annotations is still experimental, we make sure that the IntelliJ IDE picks up the code generated by our macros.

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

```
// Macro annotation to generate code commented out below.
@memoizing
case class Plus(left: Term, right: Term) extends Term

// Code that is automatically generated by the macro annotation:
// class Plus private (left: Term, right: Term) extends Term { ... }
// object Plus { ... }
```

4 Goals

4.1 Core Goals

1. **Research other potential solutions** that may have been tried for similar problems.
2. **Implement the solution approach** described in section 3 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. **Build macro annotations** for automatic code generation.
5. **Add IntelliJ IDE support** for our macro annotations.

4.2 Extension Goals

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

2. **Evaluate performance improvements** 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. Usage of a DSL in combination with Scala macros to auto-generate AST simplifications is imaginable. This would allow to easily add or modify AST simplifications 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] 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.

[3] 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.