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

May 7, 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 Approach

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 easily 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.

Because the AST used in Silicon is immutable, the flyweight pattern [1] 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.

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.

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

1. If an `apply` method is already defined, rename it to `_apply`. The already defined apply method can't be discarded because it potentially performs AST simplifications.
2. Define a new `apply` method which uses the flyweight pattern. If a new instance has to be created, either use `_apply` method if it exists, else simply create an instance using the `new` keyword.
3. Generate an `unapply` method.
4. Generate a `copy` method that calls `apply` instead of creating instances via `new` such that the flyweight pattern can't be bypassed.
5. Override `hashCode` to use `System.identityHashCode`.

The inner workings of our macro annotation is best shown in an example.

Listing 1: Input code given to the macro.

```
1  @flyweight
2  class Plus(val p0: Term, val p1: Term)
3      extends ArithmeticTerm with BinaryOp[Term]
4  {
5      override val op = "+"
6  }
7
8  object Plus extends ((Term, Term) => Term) {
9      import predef.Zero
10
11     def apply(e0: Term, e1: Term): Term = (e0, e1) match {
12         case (t0, Zero) => t0
13         case (Zero, t1) => t1
14         case (IntLiteral(n0), IntLiteral(n1)) => IntLiteral(n0 + n1)
15         case _ => new Plus(e0, e1)
16     }
17 }
```

Listing 2: Output code generated by our macro.

```
class Plus private[terms] (val p0: Term, val p1: Term)
    extends ArithmeticTerm with BinaryOp[Term]
{
    override lazy val hashCode = System.identityHashCode(this)

    def copy(p0: Term = p0, p1: Term = p1) = Plus(p0, p1)

    override val op = "+"
}

object Plus extends ((Term, Term) => Term) {
    import scala.collection.concurrent.TrieMap
    var pool = new TrieMap[(Term, Term), $returnType]

    def apply(e0: Term, e1: Term): Term = {
        pool.get((e0, e1)) match {
            case None =>
                val term = Plus._apply(e0, e1)
                pool.addOne((e0, e1), term)
                term
            case Some(term) =>
                term
        }
    }

    def unapply(t: Plus) =
        Some((t.p0, t.p1))

    import predef.Zero

    def _apply(e0: Term, e1: Term): Term = (e0, e1) match {
        case (t0, Zero) => t0
        case (Zero, t1) => t1
        case (IntLiteral(n0), IntLiteral(n1)) => IntLiteral(n0 + n1)
        case _ => new Plus(e0, e1)
    }
}
```

3 Evaluation

3.1 Concluding Performance Evaluation

To measure the performance difference, test cases over a wide variety of Viper frontends are considered.

Resources Used	VerCors, Prusti, Gobra, Vyper
Number of Parallel Verifiers	1
Repetitions	10

Implementation	Relative Performance Change (Negative is better)	Standard Derivation
Flyweight Pattern	2%	2.9%

Unfortunately, a performance improvement is not observable. The performance difference is well within the standard derivation.

3.2 Performance of Different Data Structures for the Memory Pool

Resources Used	VerCors
Number of Parallel Verifiers	1
Repetitions	10

Data Structure	Relative Performance Change (Negative is better)	Standard Derivation
`mutable.HashMap`	-1.3%	cell3
`mutable.WeakHashMap`	-0.2%	cell6
`concurrent.TrieMap`	-0.2%	cell9
`concurrent.ListMap`	+89.5%	cell9

As expected, the use of `ListMap` significantly worsens performance. The performance of `HashMap`, `WeakHashMap`, `TrieMap` are very similar considering the standard derivation. `TrieMap` however has the additional benefit of being concurrency-safe.

3.3 Memory Consumption

3.4 Memory Pool Hit Rate

3.5 AST Node Count on Equality Checks

3.6 Why did Flyweight Fail

3.7 Automatic Code Generation

3.8 Impact of AST Simplifications on Performance

Thanks tho the macro annotation, Silicon can easily be edited to ignore AST simplifications. To achieve this, calling `_apply`, which performs AST simplifications, is avoided. Instead, we directly create instances using `new`.

Listing 3: Use AST simplifications.

```
def apply(..$fields) = {
    // ...
    ${
        if (hasRenamedApplyMethod)
            q"${termName}._apply(..${fieldNames})"
        else
            q"new $className(..${fieldNames})"
    }
    // ...
}
```

Listing 4: Ignore AST simplifications.

```
def apply(..$fields) = {
    // ...
    ${
        q"new $className(..${fieldNames})"
    }
    // ...
}
```

Although the flyweight pattern itself didn't have a significant impact on performance, the macro annotation developed to implement the flyweight pattern can be quickly modified to perform experiments or benchmarks on the Silicon AST.

References

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