Common Subexpression Elimination in Dotty

Allan Renucci

School of Computer and Communication Sciences
Semester Project Report

Supervisor
Dmytro Petrashko
EPFL / LAMP

Supervisor
Prof. Martin Odersky
EPFL / LAMP

January 2016

Common Subexpression Elimination in Dotty

Allan Renucci
EPFL, Switzerland
allan.renucci@epfl.ch

Abstract

Common subexpression elimination is a well-known compiler optimisation that improves running time of compiled applications by avoiding the repetition of the same computation. Although it has been implemented on a low level such as bytecode, it misses multiple opportunities that are available on high level, such as optimizing lazy vals. We developed and implemented the transformation for Scala in a new mini-phase in the Dotty Compiler.

1 Introduction

The purpose of common subexpression elimination (CSE) is to reduce the running time of a program through avoiding the repetition of a computation. The transformation identifies a repeated computation by locating multiple occurrences of the same expression. Repeated computations are eliminated by storing the result of evaluating an expression and using the previously computed value instead of reevaluating the expression. However, expressions can have side effects and return different values when executed multiple times which makes it hard to perform CSE. For instance consider the following example:

```
val a = foo() + 2
val b = foo() + 3
```

it may be worth transforming the code to:

```
val tmp = foo()
val a = tmp + 2
val b = tmp + 3
```

But this transformation is not valid if `foo` is defined as follows:

```
def foo = Random.nextInt()
```

as it changes the behavior of the program.

In this report, we will define sufficient conditions under which the elimination of common subexpressions is possible and explain how CSE is implemented in the Dotty compiler.

2 Background

2.1 The Dotty Compiler

Dotty [1] is a platform to try out new language concepts and compiler technologies for Scala. Dotty compiler transformation pipeline is based on the notions of Mini-Phases [8]. These abstractions allow to modularize code transformations without sacrificing performance. Currently Dotty pipeline has more than 40 very finely grained mini-phases, compared to 25 phases in scalac [2], and this number is likely to increase.

Mini-Phases constitute transformations of trees, which can be efficiently pipelined, as they share a single tree traversal. Comparing with scalac, if you were to write a non-trivial phase in your compiler plugin, you will need a separate traversal of tree, slowing down the compilation. Unlike this, in Dotty you define a mini-phase, that typically doesn't trigger extraneous traversals.

CSE is implemented as a mini-phase in the compiler pipeline before type erasure. The transformation operates at source code level on a typed tree.

2.2 Idempotent expressions

As showed previously CSE cannot be performed for expressions which return different results if called multiple times. However, CSE can be performed on successive idempotent function calls. A function is said to be idempotent if, when called with the same arguments twice, the second call returns the same value and has no side effect which can distinguish it from the first call. For instance, consider the following example:

```
var init = false
def idem(x: Int) = {
  if (!init) {
    println("Initialise")
    init = true
  }
  x
}

val a = idem(1) // Prints "Initialise" and returns 1
val b = idem(1) // Returns 1 with no side effect
```

Here, `idem` is idempotent because the second call to `idem` (with the same argument) returns the same value and does not change the visible program state. We can replace the two calls to `idem` by a variable holding the computed value without altering the behavior of the program:

```
val tmp = idem(1)
val a = tmp
val b = tmp
```

In Scala, lazy vals are idempotent as well as most of the implicit conversions. CSE can reduce the overhead of using such features.

3 Implementation

3.1 General Idea

CSE is performed during the traversal of method's body. As we recursively traverse the tree, we collect available idempotent function calls. The result of a function call can be reused if:

1. It is **idempotent**. The function must return the same value when called with the same arguments multiple times and subsequent calls must have no side effect. Otherwise, subsequent calls cannot be eliminated without changing the program's behavior.

2. It can be extracted with no impact on the program's behavior. Consider the following example:

   ```
   val a = foo() + idem()
   val b = idem()

   // Rewritten
   val tmp = idem()
   val a = foo() + tmp
   val b = tmp
   ```

   We eliminate the second call to the idempotent function `idem` by reordering the instructions. We evaluate `idem` before `foo` and reuse the computed value. However, this transformation is only valid if the function `foo` doesn't have side effect. For instance, the first call to `idem` may have side effects, and reordering the instructions will reorder the effects which eventually, will change the behavior of the program.

Purity, idempotency and side-effect detection is an extremely hard problem and is not in the scope of this project [7]. This is why CSE in Dotty relies on the following assumptions:

1. Constants and immutable references are idempotent (i.e. `val`, `this`, `super`).

2. Lazy vals are idempotent. In Scala, when a val is declared with the `lazy` modifier the right-hand side of the value (the definition) will not be executed until the first time the value is accessed.

3. A method is idempotent if and only if it is annotated with the `@Idempotent` annotation. Idempotent methods are either `final` or can be be overridden only by idempotent ones.

4. A method call of the form `qual.fun(args)` is not idempotent unless the method is idempotent, its (optional) qualifier is idempotent and its arguments are idempotent.

3.2 Nested Functions

Scala lets you define functions inside other functions. Nested functions are frequently used in high level languages and are generated by the compiler for constructs such as pattern matching and closures so it is very important to handle them properly. Consider the following example:

```
def example = {
  def inner() = idem()
  val a = idem()
  foo() + a
}
```

At the time the function `inner` is executed, the expression `idem()` is available. Thus this code snippet can be rewritten as follows:

```
def example = {
  def inner() = a
  val a = idem()
  inner() + a
}
```

Note that such code cannot be written by users but the compiler can perfectly emit such code. Indeed, nested functions capture their environment and any free variables will be passed as argument of the function. This transformation is done in a subsequent phase of the compiler.

To obtain the set of available expressions of a function, we simply perform the intersection of available expressions between all the call to the nested function. In the example above, there is only one application of the function `inner` and the result is straightforward but consider the following example:

```
val a = idem1()
if (condition)
  idem2() + inner()
else
  idem3() + inner()
```

The set of available expression to `inner` is $\{idem1, idem2\} \cap \{idem1, idem3\} = \{idem1\}$.

4 Related Work

To our knowledge, CSE has not yet been implemented in any Java or Scala compiler at source level although it has been implemented at bytecode level in

JIT Compilers. "Just-In-Time" (JIT) compilers such as Java HotSpot [9] or the SELF compiler [5] eliminates redundant loads and stores, arithmetic operations, and constants.

CSE is implemented in many compilers for imperative languages [3]. The program to be optimised is represented as a flow graph whose nodes are basic blocks, that is sequences of 3-address instructions. An expression on the right hand side of an assignment is a common subexpression if it has been computed before and there is no assignment to any variable of the expression in between. CSE for imperative languages can only eliminate an expression, if all its subexpressions including itself only handle primitive values.

CSE has been implemented for compilers in strict functional languages [4]. It uses continuation passing style as intermediate language on which all transformations operate. Whereas 3-address code consists of a sequence of instructions, continuation passing style code makes control flow explicit by nesting. Hence an expression is evaluated before another expression, if it syntactically dominates that expression. Only in case of syntactic domination common subexpressions can be eliminated. Common subexpressions are restricted to those built from primitive operations that operate only on primitive types.

CSE has been implemented for lazy functional languages [6]. It uses an augmented version of the λ-calculus as intermediate language on which all transformations operate. The referential transparency of these languages makes the identification of common subexpressions very simple. Furthermore, more common subexpressions can be recognised because they can be of arbitrary type whereas standard common subexpression elimination only shares primitive values. However, analysing its effects and deciding under which conditions the elimination of a common subexpression is beneficial proves to be quite difficult.

5 Conclusion

In this project we have developed a version of CSE for the Dotty compiler. It eliminates redundant idempotent function calls in method bodies and inner functions. We have defined conditions under which our transformation is sound (i.e. it does not change the program behavior).

Our optimization complements the existing ones as they perform CSE at different levels. We recognise redundant idempotent function calls which makes it particularly relevant for optimizing lazy vals or implicit conversions.

Furthermore, CSE in Dotty currently relies on user annotated methods and poor heuristics to determine an expression purity. Therefore, it could greatly benefit from purity and side effect analysis as well as general idempotency detection.

References

[1] Dotty. `https://github.com/lampepfl/dotty`.

[2] Scala. `https://github.com/scala/scala`.

[3] J. Ullman A. Aho, R. Sethi. *Compilers: Principles, Techniques, and Tools*. Cambridge University Press, 1992.

[4] Andrew W Appel. *Compiling with Continuations*. Pearson, 2013.

[5] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.

[6] Olaf Chitil. Common subexpressions are uncommon in lazy functional languages. Technical report, Aachen University of Technology, 1998.

[7] Martin Odersky Lukas Rytz and Philipp Haller. Lightweight polymorphic effects. Technical report, EPFL, 2012.

[8] Dmitry Petrashko. Hands-on dotty. `https://www.youtube.com/watch?v=aftdOFuVU1o`, 2015. Scala World.

[9] Hanspeter Mössenböck Thomas Rodriguez Kenneth Russell David Cox Thomas Kotzmann, Christian Wimmer. Design of the java hotspot^{TM} client compiler for java 6. Technical report, Johannes Kepler University Linz, Austria, Sun Microsystems, Inc., Santa Clara, CA, 2008.