Compiler Optimizations Summary

Nicholas Springer

December 4, 2022

Introduction

Optimizations carried out by compilers provide the potential for program speed-ups without any extra effort for the programmer of the compiled code. One of the ways compilers achieve this is through reducing computation steps by removing redundancy and trivial computations, as well as simplifying certain procedures. I implemented three such compiler optimizations for a compiler of a Lisp-like language. The next section will explain the implementations of the three optimizations, and the final section will explore the impact of the optimizations on the performance of compiled programs.

Optimizations

Constant Propagation

One way to remove unnecessary computation is to precompute values that are trivially known. For instance, the compiler can easily tell that `(+ 1 2)` is equivalent to 3. By replacing the expression `(+ 1 2)` with the constant 3 in the compiled program, the compiler removes the addition step that the processor would otherwise have to do. If the expression is in a function that is called often, then many processor cycles may be saved.

My implementation of constant propagation works recursively. Each expression is tested to determine if it is a constant. For operations that can potentially be computed statically, constant propagation is applied to their arguments. If after constant propagation, all of the arguments to such an operation are constant, then the expression can be replaced with its result.

Similarly, for if statements, if the condition expression is constant after constant propagation, then the if expression can be replaced with the appropriate branch of the if statement. For instance, `(if true 1 2)` can be replaced with the constant 1.

Finally, if a variable is bound to a constant, then the locations that the variable is used can be replaced with the constant value, saving the program from having to store and load the variable value from memory. For example, `(let ((x 2)) x)` could be replaced with the constant 2. My implementation achieves this by tracking all constant variable bindings in the surrounding expressions to the current expression, allowing replacement of those variables with their constant value.

Function Inlining

The compiler can also create speed-ups by removing the overhead needed for function calls. If the function `f` is defined as `(define (f x) (+ x 2))`, then the expressions `(f 1)` and `(let ((x 1)) (+ x 2)` are equivalent, but the latter avoids the overhead of copying the argument to `f` to a new stack frame.

My implementation for inlining works by replacing calls to functions with their bodies, as shown in the previous paragraph. However, a recursive function, or multiple mutually recursive functions, cannot be inlined, as inlining the body of the function will result in another function call that would need to be inlined. To remedy this, my implementation checks to ensure the function has no recursive calls or calls to another function before inlining.

Inlining eliminates the overhead of function calls, but at the cost of increasing the size of the executable. Hence, the benefits of inlining a function may not outweigh the costs. To address this tradeoff, my implementation compares the product of the size of the body of a given function and the

number of call sites to that function with a threshold. If the product does not exceed the threshold, then the function is inlined.

Common Subexpression Elimination

Redundant computation can be avoided by detecting equivalent subexpressions and only evaluating them once. For instance, the expression `(+ (+ 1 2) (+ 1 2))` could be simplified to `(let ((x (+ 1 2))) (+ x x))`. This optimization is called common subexpression elimination.

My implementation of works by first creating a list of all non-constant subexpressions in the given expression, paired with the path taken to reach each subexpression. Then, each path is grouped with the other paths that correspond to an equivalent subexpression. The subexpressions with only one copy can then be discarded.

Common subexpressions often contain other common subexpressions. For example, the subexpression `(+ (+ (+ 1 2) 3) (+ (+ 1 2) 3))` contains common subexpressions `(+ (+ 1 2) 3)` and `(+ 1 2)`, but the former contains the latter. Hence, only the former should be substituted for a variable. My implementation achieves this by removing common subexpression paths that are prefixed by another common subexpression path. After removing these expressions, the paths must be regrouped by expression and filtered to ensure that each common subexpression still has at least two paths.

In addition to replacing subexpressions with a variable, the optimizer must also create a binding for the variable. This binding must be inserted at the lowest common ancestor of the given common subexpression. If it were placed lower in the syntax tree, then the binding would not be available at the location of some of the common subexpression instances. If the binding is placed higher in the syntax tree, then variables used in the common subexpression may not yet be bound. For each common subexpression, my implementation calculates the path to the lowest common ancestor of the subexpression's instances.

To apply the variable substitutions and insert the variable bindings, the optimizer must traverse the paths it has recorded and make the necessary substitutions and insertions. However, these modifications change the structure of the tree, causing paths to become outdated. To resolve this problem, my implementation sorts the modifications by path length, and performs the longest-path modifications first. Hence, when a given modification is applied, the modifications that have already been applied must not have changed the tree above the node of the current modification. This means the original paths of the modifications can be used directly.

Impact on Performance

One consideration of the discussed optimizations is that they can work in tandem to produce better optimizations than on their own. For example, in the following program, constant propagation would not do anything. However, after inlining the function `f`, constant propogation can directly evaluate `(+ (f 1 0) (+ (f 2 3) (f 4 5)))` into a constant expression, saving multiple addition operations.

inlining-allows-cprop.lisp:

```
(define (f x y) (+ x y))
(define (do-n-times n)
    (if (= n 0) true (do
        (print (+ (f 1 0) (+ (f 2 3) (f 4 5))))
        (do-n-times (- n 1)))))
(do-n-times 1000000)
```

Similarly, in the following program, there are no common subexpressions, but once constant propagation is applied, the subexpression `(= x 3)` becomes common, meaning common subexpression elimination can be applied.

cprop-allows-cse.lisp:

```
(define (do-n-times n)
    (if (= n 0) true (do
```

```
        (print (let ((x (print 1))) (pair (= x (+ 1 2)) (= x 3))))
        (do-n-times (- n 1)))))
(do-n-times 1000000)
```

To evaluate constant subexpression elimination, the following program can be used:

cse.lisp:

```
(define (f x)
    (+ (+ (+ (+ (+ (+ x x) (+ x x)) (+ x x)) (+ x x)) (+ x x)) (+ x x)))
(define (do-n-times n) (if (= n 0) true (do (f 2) (do-n-times (- n 1)))))
(do-n-times 1000000)
```

To test the efficacy of the optimizations, the above programs were benchmarked before and after optimization.

inlining-allows-cprop.lisp had an average runtime of 79.57 milliseconds with no optimization. After inlining and constant propagation were applied, the runtime dropped to an average of 71.56 milliseconds, representing a 10.1% speedup.

cprop-allows-cse.lisp had an average runtime of 7.90 milliseconds with no optimization. After inlining and constant propagation were applied, the runtime dropped to an average of 6.72 milliseconds, representing a 14.9% speedup.

cse.lisp had an average runtime of 21.73 milliseconds with no optimization. After inlining and constant propagation were applied, the runtime dropped to an average of 18.14 milliseconds, representing a 16.6% speedup.

We can see from the benchmark results that inlining, constant propagation, and common subexpression elimination are effective optimizations that can result in significant speedups. Given the low cost for compiler users to apply these optimizations, it seems that these optimizations are important to include in any compiler that wishes to generate efficient programs.