Ian Layzer
Rob Lewis
CSCI 1260
12 December 2022

Capstone: Compiler Optimizations

Introduction

A compiler is a function that takes in a program in one language and outputs a program that does the same thing in another language; it transforms. A compiler optimization is also a program transformation that trades compile-time computation for runtime efficiency (usually time or space). I implemented four compiler optimizations for this project: variable uniquification, constant propagation, function inlining, and common subexpression elimination. All four of these optimizations happen at the abstract syntax tree (AST) level, meaning they are all functions that take an AST as an input and output an 'optimized' AST, applied to the input program sequentially after the tokenization and parsing phases.

Implementation

Variable uniquification

This optimization simply makes every variable and function argument in the program unique. Consider the following program:

(define (f x) (+ x 1)) (let (x 2) (f x))

The function **f** and the **let** expression both have a variable **x**. This duplication of variables will cause problems in our other optimizations, so we want to make every variable unique, outputting the following program:

(define (f x1) (+ x1 1)) (let (x2 2) (f x2))

Constant propagation (CP)

Constant propagation is an optimization that identifies compile-time constant computation and performs that work at compile-time rather than at runtime. For example:

(+ 1 (+ 2 (+ 3 (+ 4))))

will yield the same result at every invocation of the program, meaning that inside the compiler we can interpret that expression into the constant **10**. Because the work (interpretation) is only done once at compile time rather than each time the program runs, this can lead to a speed up.

Function inlining (FI)

Function inlining is an optimization that chooses to replace all calls to a function throughout the program with the variable-bound body of the function. Given the program:

(define (f x) (+ x 1)) (f 2)

We would inline the function **f** yielding the program:

(let (x 2) (+ x 1))

Thus, we can avoid the relatively expensive function call to **f**. To decide which functions to inline, I created a heuristic that examines the size of the function and the number of invocations of that function and compares it to some threshold. Essentially, **sizeof(f) * num_calls(f) < threshold**.

Common subexpression elimination (CSE)

Common subexpression elimination is meant to eliminate redundant work in a program. Consider the example program:

(+ (do-work 1000) (+ (do-work 1000) (do-work 1000)))

In this case, **do-work** is a statically constant function that performs a complex mathematical computation that takes a long time to run. Because calling **do-work** on the same input **1000** will produce the same result every time, we can see at compile-time that our program is performing unnecessary work. To address this, we can *eliminate* the common subexpression **(do-work 1000)** by factoring it out into a variable, so that we only need to perform the computation once rather than three times. The optimized program would look like the following:

(let (var (do-work 1000)) (+ var (+ var var)))

It's not always so simple, however, because not all expressions can be safely eliminated in this way. Any expression with side effects is not safe, and in our simple Lisp-like language, only **(read-num)** and **(print)** are unsafe. However, this means that any subexpression that contains somewhere down the tree calls to these functions is also unsafe. To address this, my implementation recursively explores the AST, gathering all the safe subexpressions in the tree,

the paths from root to node of that subexpression, and keeping track of the safety of user-defined functions as it goes. Function safety detection also needed to include protection against cyclical function dependency a la (mutual) recursion to prevent infinite recursion in my safety determination function.

Next, my implementation finds subexpressions that occur more than once in the tree, as well as filters out constants like numbers, booleans, and null, as it doesn't really make sense to eliminate those. Then, for each set of common subexpressions, the implementation takes the paths to each node, finds the lowest common ancestor of the common nodes, and replaces that common ancestor with a **let** expression as shown in the example above.

Results

I used the course-provided benchmarking script to compile and run a collaboratively generated suite of benchmarking programs with various combinations of optimizations. These optimized runs were compared to the unoptimized baseline. The following table shows the percentage reduction in execution time when compiling with various (combinations of) optimizations.

Percentage Reduction in Execution Time for Various Optimizations

Percentile	*FI*	*CP*	*FI + CP*	*CSE*	*FI + CP + CSE*
0.25	0.92	2.06	-8.88	-22.92	-3.09
0.50	10.84	6.71	8.61	-7.37	4.17
0.75	27.80	18.86	35.50	3.28	25.39
0.90	44.89	42.20	56.85	10.41	35.53
0.95	51.97	52.97	68.13	17.22	74.5

We can see that both the *function inlining* and *constant propagation* see very sizable performance speedups for P50 and above (especially P75+). For *CP*, very few files saw decreased performance, while *FI* saw a bit more slowdowns in the P0-P25 range. In my opinion,

this indicates a suboptimal heuristic for deciding whether or not to inline a function. More tweaking of the inlining decision threshold could improve this optimization's low-end performance. Overall, it appears that *FI + CP* was the most widely beneficial combination of optimizations, with a P75 of 35.50% reduction in execution time. One thing that surprised me was the poor performance of *common subsequence elimination* at the low end. I think this is because I did not use a complex heuristic for when to eliminate a subsequence. My implementation does not eliminate simple int, boolean, nil, and variable expressions, but it seems that it was still making inefficient elimination decisions in some cases, leading to the low end slowdown because of the added overhead of the added variable lookups. By looking at the P95 for all three optimizations, we see that there is clearly a specific subset of programs where they really excel. This indicates that developing better heuristics for when to apply each optimization would lead to significantly better results over the body of all programs.