Arithmetic and Stack Optimizations

Lecture #11

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

Previously, we added in support for pointers to our language. This required a major overhaul of how we handled Numbers and LLVMValues.

```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/test9
int main() {
    int a; int b;
    int c; int d;
    int *e; int *f;

    a = 18; print a;
    e = &a; b = *e; print b;

    c = 12; print c;
    f = &c; d = *f; print d;

    int g; int *h; int **j; int ***k;
    g = 5; h = &g; j = &h; k = &j;
    int n; int *m; int **l;
    l = *k; m = *l; n = *m;
    print n;
}
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/test9 && clang test9.ll -o test9 && ./test9
-----RUN-----
18
18
12
12
5
```

Optimizations

Optimizations are modifications to either parsed ASTs or generated LLVM that aim to retain identical runtime behavior, but reduce binary size or increase runtime speed.

Optimizations that deal with ASTs are often recursive (one that we'll look at today will be), but often they can be applied as soon as a statement is parsed (we will look at some of these optimizations later on).

Because we have a recursive AST traversal, we are subject to DFT complexity, a.k.a. $O(V + E)$. We will see that this greatly increases compile-time when we compile large files. This compile-time duration increase is part of why both `clang` and `gcc` turn off optimization by default. We will leave it on, and you will soon see why.

The Goal

I've written a script, `test/generate_arithmetic_test.py`, that generates a random file for ECCO to compile. The file contains 500 print statements, each with 5-10 binary arithmetic operations, and therefore 6-11 integer constants.

Our goal is to minimize the output LLVM generated for this file as much as possible.

The Plan

We will implement two optimizations, and slightly update our LLVMValue interactions to minimize generated code:

1. Arithmetic expression folding
2. Stack allocation reduction
3. New LLVMValueType.CONSTANT to do last-minute or missed optimizations at the LLVM level

Arithmetic Expression Folding

If we want to compile the statement `print 1 + 2 - 3 * 4 / 5;`, there's really no good reason why any computations should be performed at runtime. This expression should have the same value today as it does in a million years, it doesn't rely on any variables or configurations of the compiler. Therefore, we can "fold" the expression down into its value (1) at compile-time, so that we don't waste any time at runtime computing what is functionally identical to an integer literal.

This works for purely constant sub-expressions as well. `print 1 * 2 + x - 4 / 5;` shouldn't compute $1 \cdot 2$ and $4 \cdot 5$ at runtime.

Expression Folding Rules

We're going to apply 3 sets of folding rules:

1. Any operation between two constants is folded
2. "Zero" rules ($x + 0 = x$, $x \cdot 0 = 0$, etc.) are folded
3. Any double-operation is folded ($x \cdot y \cdot z = (x \cdot y) \cdot z$)

Stack Allocation Reduction

There is no good reason why we should allocate a register on the stack, store a constant value into that register, load that register's contents into a new register, and then use the value. We did this to learn the ins and outs of basic LLVM pointers, but now we should fix it so that this:

```
%51 = alloca i32, align 4
%52 = alloca i32, align 4
store i32 0, i32* %52
%53 = load i32, i32* %52
store i32 3, i32* %51
%54 = load i32, i32* %51
%55 = add nsw i32 %53, %54
```

becomes this:

```
%51 = add nsw i32 0 3
```

(ignoring our previous optimization)

`LLVMValueType.CONSTANT`

I've added an `LLVMValueType.CONSTANT` type that allows us to short-circuit a bunch of our redundant stores/loads.

Consequences:

- Everywhere we printed a "%" is now wrapped in a conditional to check if we're actually dealing with a register or not
- `llvm_resize` acts solely as a truncate function for constants
- A bunch of register loading is now conditional

Success!

```
(base) charlesaverill@pop-os:~/Desktop/ecco$ time ./scripts run examples/arith_test -O0 --output unoptimized.ll && wc -l unoptimized.ll
-----RUN-----

real    0m1.009s
user    0m0.965s
sys     0m0.042s
19052 unoptimized.ll
(base) charlesaverill@pop-os:~/Desktop/ecco$ time ./scripts run examples/arith_test -O1 --output mid_optimized.ll && wc -l mid_optimized.ll
-----RUN-----

real    0m14.550s
user    0m14.507s
sys     0m0.044s
524 mid_optimized.ll
(base) charlesaverill@pop-os:~/Desktop/ecco$ time ./scripts run examples/arith_test -O2 --output full_optimized.ll && wc -l full_optimized.ll
-----RUN-----

real    0m5.156s
user    0m5.128s
sys     0m0.028s
524 full_optimized.ll
```

36X line reduction. I'm working on fixing the bug causing the 15x compile-time increase for O1, but as we can see it currently functions identically to O2 so we shouldn't be too worried yet.