Code optimization

Handout written by Maggie Johnson and revised by me.

Optimization is the processing of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster or consumes less memory. It is a really a misnomer that the name implies you are finding a "optimal" solution— in truth, optimization aims to improve, not perfect, the result.

Optimization is the field where most compiler research is done today. Although there are still some advanced problems left, much of parsing is very well understood. Optimization, on the other hand, still retains a sizable measure of mysticism. High-quality optimization is more of an art than a science. Compilers for mature languages aren't judged by how well they parse or analyze the code (you just expect it to do it right with a minimum of hassle.) They are judged by the quality of the object code they produce.

Many optimization problems are NP-complete, and for most examples here, it is probably possible to come up with a case where the algorithm fails to produce better code or perhaps even makes it worse. However, these algorithms tend to do rather well overall.

It's worth reiterating here that efficient code is mainly the result of intelligent decisions by the programmer. No compiler I know of is smart enough to replace BubbleSort with Quicksort. If a programmer starts off with a lousy algorithm, no amount of optimization can make it super-fast. In terms of big-O, a compiler can only make improvements to constant factors. But, all else being equal, you want an algorithm with low constant factors.

First let me note that you probably shouldn't try to optimize the way we will discuss today in your favorite high-level language. Consider the following two code snippets which each walk through an array and set every element to one. Which one is faster?

```
int arr[10000];

void Binky(void)
{
   int i;
   for (i=0; i < 10000; i++)
       arr[i] = 1;
}
```

```
int arr[10000];

void Winky(void)
{
   register int *p;
   for (p = arr; p < arr + 10000;)
      *p++ = 1;
}
```

You will invariably encounter people who think the second one is faster. And they are probably right if you use a compiler without optimization. However, many modern compilers can emit the same object code for both, using some clever optimization techniques (in particular, this one is called "loop-induction variable elimination). The moral of this story for a programmer is that most often you should write code that is easier to understand and let the compiler do the optimization.

Control-flow analysis

Consider all that has happened up to this point in the compiling process—lexical analysis, syntactic analysis, semantic analysis and finally intermediate-code generation. The compiler has done an enormous amount of analysis, but it still doesn't really know how the program does what it does. In control-flow analysis, the compiler figures out even more information about how the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

As an example, here is some SOOP code and the corresponding TAC that computes fibonacci numbers. (This example is adapted from Muchnick.)

```
int fib(int base)
{
   if (base <= 1) {
      return base;
   } else {
      int i;
      int f0;
      int f1;
      int f2;
      f0 = 0;
      f1 = 1;
      i = 2;
      while (i <= base) {
          f2 = f0 + f1;
          f0 = f1;
          f1 = f2;
          i = i + 1;
      }
      return f2;
   }
}
```

```
_L0:
   BeginFunc 4 ;
   _t0 = 1 ;
   _t1 = base < _t0 ;
   _t2 = base == _t0 ;
   _t3 = _t1 || _t2 ;
   IfZ _t3 Goto _L1 ;
   Return base ;
   Goto _L2 ;
_L1:
   _t4 = 0 ;
   f0 = _t4 ;
   _t5 = 1 ;
   f1 = _t5 ;
   _t6 = 2 ;
   i = _t6 ;
_L3:
   _t7 = i < base ;
   _t8 = i == base ;
   _t9 = _t7 || _t8 ;
   IfZ _t9 Goto _L4 ;
   _t10 = f0 + f1 ;
   f2 = _t10 ;
   f0 = f1 ;
   f1 = f2 ;
   _t11 = 1 ;
   _t12 = i + _t11 ;
   i = _t12 ;
   Goto _L3 ;
_L4:
   Return f2 ;
_L2:
   EndFunc
```

Control-flow analysis begins by constructing a *control-flow graph*, which is a graph of the different possible paths program flow could take through a function. To build the graph, we first divide the code into *basic blocks*. A basic block is a segment of the code that a program must enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all statements are executed consecutively if the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point. The program will execute every instruction in a basic block sequentially every time if it executes the first instruction.

A basic block begins in one of several ways:

- the entry point into the function

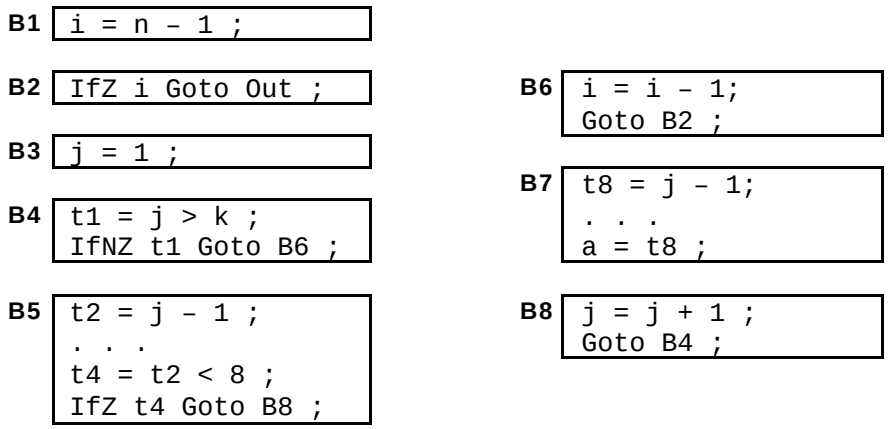
- the target of a branch (in our example, any label)
- the instruction immediately following a branch or a return

A basic block ends in any of the following ways:

- a jump statement
- a conditional or unconditional branch
- a return statement

(There are certain exceptions to these rules that we will never encounter with SOOP and our edition of TAC.) Using the rules above, how would we divide the Fibonacci TAC code into basic blocks?

How is the control-flow graph is constructed from here? Each basic block is a node in the graph, and the possible different routes a program might take are the connections. Connect up the flow graph for the following code already divided into basic blocks:



What does a loop look like in a control-flow graph?

You probably have all seen the gcc warning or javac error about: "Unreachable code at line XXX." How can the compiler tell?

Local optimizations

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information, we just work with the statements within the block. But correspondingly, we also aren't likely to get that much mileage out of optimizations that work on such a narrow scale. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform.

Some of the common local optimizations are constant folding and propagation, algebraic identities, operator strength reduction, and common sub-expression elimination.

Constant folding

Arithmetic expressions can be evaluated at compile time if the operands have values that are themselves known at compile time. Whenever expressions like `10+2*3` are encountered, the compiler can compute the result at compile-time (16) and emit code as if the input contained the result rather than the original expression. The expression had to be evaluated at least once, but if the compiler does it, it means you don't have to do it again as needed during runtime. One thing to be careful about is that the compiler must obey the grammar and semantic rules from the source

language that apply to expression evaluation, which may not necessarily match the language you are writing the compiler in. (For example, if you were writing an APL compiler, you would need to take care that you were respecting its Iversonian precedence rules).

Constant-folding is what allows a language to accept constant expressions where a constant is required (such as a case label or array size):

```
int arr[20 * 4 + 3];

switch (i) {
   case 10 * 5: ...

}
```

In both cases, the expression can be resolved to an integer constant at compile time and thus, we have the information needed to generate code. If either expression involved a variable, though, there would be an error. How could you re-write the grammar to allow the SOOP grammar constant folding in case statements or array declarations? This situation is a classic example of the gray area between syntactic and semantic analysis.

Constant propagation

If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant. Perhaps you noticed the following inefficiency in our earlier Fibonacci example. On the left is the original, on the right is the improved version after constant propagation:

```
_t4 = 0 ;
f0 = _t4 ;
_t5 = 1 ;
f1 = _t5 ;
_t6 = 2 ;
i = _t6 ;
```

```
f0 = 0 ;
f1 = 1 ;
i = 2 ;
```

This code is both smaller and faster than the original. Constant propagation has the potential to expose many additional optimizations. Consider this TAC fragment before and after constant propagation:

```
t1 = 5;
t2 = 6;
t3 = t1 + t2;
```

```
t3 = 5 + 6;
```

Once we're done, we obviously needs a round of constant folding to make it really efficient. But wait! When we're done with that, can't we run another round of constant propagation?…

This is the classic reason that optimization remains such an interesting field: one optimization may expose possibilities for others. How do you know you are done? (The reverse is also true, one optimization may obscure or remove possibilities for others.)

Algebraic identifies

Of course, the best sort of optimization is to remove useless instructions entirely. The rules of arithmetic can come in handy when looking for redundant calculations to eliminate. Whenever you encounter one of the expressions on the left, you can replace with the right side:

```
x+0  = x
0+x  = x
x*1  = x
```

```
1*x  = x
0/x  = 0
x-0  = x
```

You might ask, could I make my intermediate code generator smart enough not to generate such silly things in the first place?

Operator strength reduction

Operator strength reduction replaces an operator by a "less expensive" one. Given each group of identities below,) which operations are the most and least expensive, assuming f is a float and i is an int? (Trick question: it may differ for differing architectures— you need to know your target machine to optimize well!)

1) `i*2  =  2*i  =  i+i  =  i<<2`
2) `i/2  =  (int)(i*0.5)`
3) `0-1  =  -i`
4) `f*2  =  2.0 * f  = f + f`
5) `f/2.0 = f*0.5`

Strength reduction is often performed as part of *loop-induction variable elimination*. An idiomatic loop to zero all the elements of an array might look like this in SOOP and its corresponding TAC:

```
while (i < 100) {
    arr[i] = 0
    i = i + 1;
}
```

```
L0: _t2 = i < 100;
    IfZ _t2 Goto _L1 ;
    _t4 = 4 * i ;
    _t5 = arr + _t4 ;
    *_t5 =  0 ;
    i = i + 1 ;
L1:
```

Each time through the loop, we are multiplying i by 4 (the element size) and adding to the array base. Instead, we could be maintaining the address to the current element and just adding 4 each time instead:

```
    _t4 = arr ;
L0: _t2 = i < 100;
    IfZ _t2 Goto _L1 ;
    *_t5 =  0;
    _t5 = arr + 4;
    i = i + 1 ;
L1:
```

This eliminates the multiplication entirely and reduces the need for an extra temporary.

Copy propagation

This optimization is similar to constant propagation, but generalized to non-constant values. If we have an assignment `a = b` in our instruction stream, we can replace later occurrences of a with b (assuming there are no changes to either variable in-between). Given the way we generate TAC code, this is a valuable optimization since it is able to eliminate a large number of instructions that only serve to copy values from one variable to another.

The code on the left makes a copy of `t1` in `t2`, and a copy of `t4` in `t3`. In the optimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

```
t2 = t1;
t3 = t2 * t1 ;
t4 = t3 ;
t5 = t3 * t2 ;
c = t5 + t4 ;
```

```
t3 = t1 * t1 ;
t5 = t3 * t1 ;
c = t5 + t3 ;
```

Dead code elimination

If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream. So if we have

```
t1 = t2 + t3 ;
```

and t1 is never used again, we can eliminate this instruction altogether. However, we have to a be a little particular about the operation, for example, if t1 holds the result of a function call:

```
t1 = L0();
```

Even if t1 is never used again, we cannot eliminate the instruction because we can't be sure that called function has no side-effects. Dead code can occur in the original source program but is more likely to have resulted from some of the optimization techniques run previously.

Common sub-expression elimination

Two operations are *common* if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time. An expression is *alive* if the operands used to compute the expression have not been changed. An expression that is no longer alive is *dead*.

```
main()
{
    int x, y, z;

    x = (1+20)* -x;
    y = x*x+(x/y);
    y = z = (x/y)/(x*x);
}
```

straight translation:

```
t1 = 1 + 20 ;
t2 = -x ;
x = t1 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
t6 = x * x ;
z = t5 / t6 ;
y = z ;
```

What sub-expressions can be eliminated? How can valid common sub-expressions (live ones) be determined? Here is an optimized version, after constant folding and propagation and elimination of common sub-expressions:

```
t2 = -x ;
x = 21 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
z = t5 / t3 ;
y = z ;
```

Global optimizations and data-flow analysis

So far we were only considering making changes within one basic block. With some additional analysis, we can try out some of these optimizations across basic blocks, making them *global* optimizations. It's worth pointing out that global in this case does not mean across the entire program. We usually only optimize one function at a time. Inter-procedural analysis is an even larger task, one not attempted by most compilers.

The additional analysis the optimizer must do to perform optimizations across basic blocks is called *data-flow analysis*. Data-flow analysis is much more complicated than control-flow analysis, and we can only scratch the surface here, but were you to take CS243 (a wonderful class, in my opinion) you will get to delve much deeper!

Let's consider a global common sub-expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be *available* at that point. Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis.

Each block is a node in the *flow graph* of a program. The successor set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is *defined* at the point where it is assigned a value and *killed* when one of its operands is subsequently assigned a new value. An expression is *available* at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not subsequently killed.

`avail[B]` = set of expressions available on entry to block B
`exit[B]` = set of expressions available on exit from B

`avail[B]` = $\cap_{x \in pred[B]}$ `exit[x]` (i.e. B has available the intersection of the exit of its predecessors)

`killed[B]` = set of the expressions killed in B
`defined[B]` = set of expressions defined in B:

```
exit[B] = avail[B] - killed[B] + defined[B]
avail[B] = ∩ (avail[x] - killed[x] + defined[x])
           x  pred[B]
```

Here is an algorithm for global common sub-expression elimination:

1) First, compute defined and killed sets for each basic block (this does not involve any of its predecessors or successors).
2) Iteratively compute the avail and exit sets for each block by running the following algorithm until you hit a stable fixed point:
   a) Identify each statement `s` of the form `a = b op c` in some block B such that `b op c` is available at the entry to B and neither `b` nor `c` is redefined in B prior to `s`.
   b) Follow flow of control backward in the graph passing back to but not through each block that defines `b o c`. The last computation of `b o c` in such a block reaches `s`.
   c) After each computation `d = b op c` identified in step 2a, add statement `t = d` to that block where `t` is a new temp.
   d) Replace `s` by `a = t`.

Try an example to make things clearer:

```
L0:
   BeginFunc 0 ;
   b = a + 2 ;
   c = 4 * b ;
   t1 = b < c;
   ifNZ t1 goto L1 ;
   b = 1 ;
L1:
   d = a + 2
EndFunc
```

First, divide the code above into basic blocks. Now calculate the available expressions for each block. Then find an expression available in a block and perform step 2c above. What common sub-expression can you share between the two blocks?

What if the above code were:

```
L0:
   BeginFunc 0 ;
   b = a + 2 ;
   c = 4 * b ;
   t1 = b < c ;
   ifNZ t1 goto L1 ;
   b = 1 ;
   z = a + 2 ; <========= an additional line here
L1:
   d = a + 2
   EndFunc ;
```

Code motion

Code motion (also called *code hoisting*) unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation. The most common form of code motion is *loop-invariant* code motion which moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. What statements inside the following TAC code can be moved outside the loop body?

```
L0:
   t1 = t2 + t3;
   t4 = t4 + 1;
   PrintInteger(t4);
   t6 = 10;
   t5 = t4 == t6
   ifZ Goto L0;
```

We have an intuition of what makes a loop in a flowgraph, but here is a more formal definition. A loop is a set of basic blocks which satisfies two conditions:

1. All are *strongly connected*, i.e, there is a path between any two blocks.
2. The set has a unique *entry point*, i.e, every path from outside the loop that reaches any block inside the loop enters through a single node. A block n *dominates* m if all paths from the starting block to m must travel through n. Every block dominates itself.

For loop L, moving invariant statement s in block B which defines variable v outside the loop is a safe optimization if:

1. B dominates all exits from L

2. No other statement assigns a value to v
3. All uses of v inside L are from the definition in s.

Loop invariant code can be moved to just above the entry point to the loop.

Optimization soup

As one compiler textbook author (Pyster) puts it:

> "Adding optimizations to a compiler is a lot like eating chicken soup when you have a cold. Having a bowl full never hurts, but who knows if it really helps. If the optimizations are structured modularly so that the addition of one does not increase compiler complexity, the temptation to fold in another is hard to resist. How well the techniques work together or against each other is hard to determine."

Bibliography

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

R. Mak, Writing Compilers and Interpreters. New York, NY: Wiley, 1991.

S. Muchnick, Advanced Compiler Design and Implementation . San Francisco, CA: Morgan Kaufmann, 1997.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.