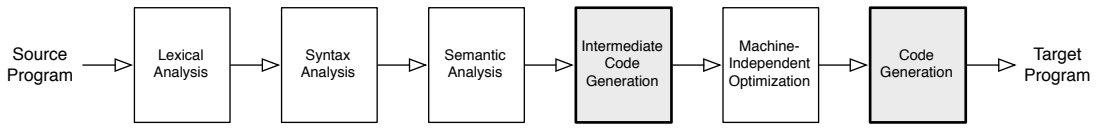
HW 8: TAC, x86 Code, Intro to Optimization

CSCI 434T
Fall, 2007

Overview



This week, we dive into the compiler's back end. The first readings focus on the run-time environment for your program, which describes how it executes machine code and how it manages the stack, the heap, and other computer resources. We will discuss the particulars of the x86 environment in lab, in preparation for writing a basic x86 code generator for IC.

The last two readings introduces the topic of program optimization. They explore the goals, the challenges, and the various forms of optimization. This week we focus on one simple optimization, redundant expression elimination via value numbering. This is a *local* optimization that operates only on small contiguous sequences of instructions. Next week, we will begin to explore a general framework for describing many program analysis and optimizations in a unified at elegant way.

Readings

- Dragon 7 – 7.2, 8.1 – 8.3
- Code Generation Materials on web site
- Cooper and Torczon 8.1 – 8.4
- Dragon 8.4 – 8.5 (You may want to read Cooper and Torczon first, and use this to fill in any remaining details.)

Exercises

I will be most interested in discussing Problems 1, 2, 7, and 8 next week.

1. This question asks you to design the `tac` package for your IC compiler, as outline in the PA 3 handout. Please come to the tutorial meeting with a design detailed enough to discuss the following items:

   - The TAC instruction set for your compiler. Be able to justify your choice of instructions and how they will be used.
   - The top-level design of the `tac` package:
     - What are the main classes, what will they do?
     - How do you represent a TAC instruction?
     - How do you represent TAC operands?

   We will talk about this on Monday, but you need not prepare a written answer to turn in. The relevant details should appear in the compiler write up when you submit the TAC generator.

2. Consider a `switch` statement with the following syntax:

```
switch(e) {
  case v1:  s1;
  case v2:  s2;
  ...
  case vn:  sn;
  default:  sd;
}
```

where e is an expressions, $v_{1..n}$ are constants, and $s_{1..n}$ and s_d are statements. When the value of e matches the constant v_k, the program executes statement s_k; if the program does not match any case, it executes s_d. The execution does not fall through to the next case. One way to implement this construct in TAC is by testing each case sequentially. However, for constructs with a large number of cases, this implementation can be slow ($O(n)$ in n, the number of explicit cases).

(a) If the listed cases represent a dense set in some range of values $[l, u]$, a better implementation is to use a jump table containing case labels and have a single table lookup to figure out the matching case. Give a precise description of the table contents and show a translation of the switch statement to a low-level TAC that implements this approach. You may extend our TAC language if necessary.

(b) The solution proposed above has the drawback that the jump table may be too large if the set of case values is sparse. Describe at least two translation approaches that would permit the program to find the matching case faster, on average, than sequential search, but which use data structures that are linear in the number of cases in the switch.

(c) What complications do your approaches from (a) and (b) bring to the process of building the internal representation for a program used during optimization? How could you address them? (A few sentences is sufficient.)

(d) Explain why switch constructs are less frequent in object-oriented languages.

3. Here is a small IC program:

```
class A {
   int x;
   int y;
   void m(int w, int z) {
      int r = w / z + this.x * this.x;
      {
         int k = w + 1;
      }
   }
   void main(string[] args) { }
}

class B extends A {
   int z;
   void n(string s) { }
   void m(int g, int h) { }
}

class C extends B {
   void m(int g, int h) { }
   void p() { }
}

class D extends A {
   string k;
   void p() { }
}
```

Using Appel, Figure 14.3, as a model, show the object and dispatch vector layouts for the four classes shown here. Include the object offsets for fields and dispatch vector indexes for methods.

Also, lay out the stack frame for method `A.m`. Include in your diagram the stack pointer, frame pointer, locations of parameters, `this`, and local variables. Also show where the TAC temporary variables are stored, assuming that the TAC for this method is:

```
t1 = w / z
t2 = this.x
t3 = this.x
t3 = t2 * t3
t0 = t1 + t2
r = t0
t4 = w + 1
k = t4
```

4. The `gcc` compiler has generated the following x86 code for a method in an object-oriented language:

```
_f:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl 16(%ebp), %eax
  addl 12(%ebp), %eax
  movl %eax, -4(%ebp)
.L2:
  movl 8(%ebp), %eax
  movl 8(%ebp), %edx
  movl 4(%eax), %ecx
  movl 8(%edx), %eax
  cmpl %eax, %ecx
  jge .L5
  movl 8(%ebp), %ecx
  movl 8(%ebp), %edx
  movl -4(%ebp), %eax
  addl 4(%edx), %eax
  movl %eax, 4(%ecx)
  jmp .L2
.L5:
  movl 8(%ebp), %eax
  subl $12, %esp
  movl (%eax), %edx
  addl $32, %edx
  pushl %eax
  movl (%edx), %eax
  call *%eax
  addl $16, %esp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

   (a) Assuming that the stack grows downwards, draw the memory layout during the execution of this method. The layout must contain all of the pieces of memory that the execution of this method accesses.

(b) Show a possible input program that this code may have been generated from.

(c) Is it possible to identify whether this method is static or virtual by inspecting its generated code? Explain.

(d) Would it be safe to remove the instruction "`addl $16, %esp`" at the end of the program? Explain.

5. The translation from TAC to assembly code can be expressed as a function $CG[\ ]$ in much the same was as the TAC translation function $T[\ ]$ was described last week. This function converts one TAC instruction into one or more assembly instructions that implement the TAC operation. In order to properly generate the assembly code, the code generation function will take an "augmented TAC" in which each variable name has been annotated with its offset from the frame pointer, each field access has been annotated with the offset at which the field is located in the object, and so on. (You may wish to think about how your TAC instruction objects will access this information when it is needed in your compiler.) Given these annotations, here is the translation of the TAC add instruction:

$$CG[\ \mathtt{x^a = s^b + t^c}\ ] \equiv \begin{array}{l}\texttt{movl b(\%ebp), \%eax}\\ \texttt{addl c(\%ebp), \%eax}\\ \texttt{movl \%eax, a(\%ebp)}\end{array}$$

Of course, we would translate differently if one or more operands to + were constants instead of variables:

$$CG[\ \mathtt{x^a = J + t^c}\ ] \equiv \begin{array}{l}\texttt{movl J, \%eax}\\ \texttt{addl c(\%ebp), \%eax}\\ \texttt{movl \%eax, a(\%ebp)}\end{array}$$

$$CG[\ \mathtt{x^a = J + K}\ ] \equiv \begin{array}{l}\texttt{movl J, \%eax}\\ \texttt{addl K, \%eax}\\ \texttt{movl \%eax, a(\%ebp)}\end{array}$$

Converting a sequence of TAC instructions $s_1; \ldots; s_n$ is defined in the obvious way: $CG[\ s_1; \ldots; s_n\ ] \equiv CG[\ s_1\ ]; \ldots; CG[\ s_n\ ]$.

(a) Write the compilation function for the following cases. Be sure to use proper x86 instructions and addressing modes!

- $CG[\ \mathtt{s^a = t^b}\ ]$
- $CG[\ \mathtt{x^a = s^b / t^c}\ ]$
- $CG[\ \mathtt{x^a} = (\mathtt{y^b.f})^\mathtt{c}\ ]$, where c is the object offset of field `f`.
- $CG[\ (\mathtt{y^b.f})^\mathtt{c} = J\ ]$, where c is the object offset of field `f` and J is a constant.
- $CG[\ \mathtt{x^a[y^c] = z^b}\ ]$

(b) Augment the TAC for `A.m` with the offset information and show the compilation of those TAC instructions using $CG[\ ]$.

(c) There will be some obvious inefficiencies in your translation. Identify some of them and discuss how your code generator might avoid them. A short list or a few sentences is sufficient.

6. Consider the following TAC code:

```
 x = 2
 y = 3
 z = 11
L0:
 T0 = x < 10
```

```
 fjump T0 L1
 T1 = x < y
 fjump T1 L3
 T2 = x + 1
 x = T2
 jump L2
L3:
 T3 = y < 100
 fjump T3 L2
 T4 = y + 1
 y = T4
 jump L3
L2:
 T5 = z + 3
 z = T5
 jump L0
L1:
```

(a) Build the basic blocks and control flow graph for this code.

(b) Identify the natural loops.

7. Consider the following two basic blocks:

`a = b + c`	`a = b + c`
`d = c`	`e = c + c`
`e = c + d`	`f = a + c`
`f = a + d`	`g = b + e`
`g = b + e`	`h = b + c`
`h = b + d`	

(a) Build a DAG for each block. (The Dragon book and Cooper and Torczon use different notation for the DAGs. I suggest using the Dragon book form — it is a little more flexible.)

(b) Value number each block.

(c) Explain any differences in the redundancies found by these two techniques.

(d) At the end of each block, `f` and `g` have the same value. Why do the algorithms have difficulty discovering this fact?

8. Dragon 8.5.6