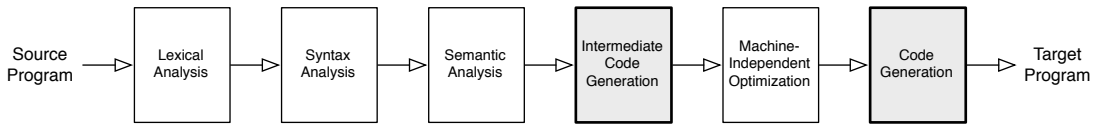
HW 8: TAC, x86 Code, Intro to Optimization

CSCI 434T
Fall, 2015

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. This should hopefully be mostly a review of cs237 material.

We will discuss the particulars of the x86 environment in lab next week, in preparation for writing a basic x86 code generator for IC.

The last two readings introduce 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.

Please focus on questions 1 – 5. Questions 6 and 7 are related to x86 code generation. Think about them before we discuss code generation in lab next week, but don't worry about writing them up.

Readings

- Dragon 7 – 7.2
- Code Generation Materials on web site
- Appel 14 (from last week)
- Cooper and Torczon 8.1 – 8.4 (in my mailbox)
- Dragon 8.1 – 8.3
- Dragon 8.4 – 8.5 (You may want to read Cooper and Torczon first, and use this to fill in any remaining details.)

Exercises

1. This question asks you to design the `tac` package for your IC compiler, as outlined in the PA 3 handout. Please write up the basic design of your TAC package, with enough detail to identify the following items:
   - The TAC instruction set for your compiler (The TAC description in the materials on the web for PA 3 is a good start, although there will be some additions, and possibly a few subtractions).
   - 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?

2. 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 (or vtable) layouts for the four classes shown here. Include the object offsets for fields and dispatch vector indexes for methods. Assume all data is stored as 64-bit values.

Also, lay out the stack frame for method `A.m`, 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
```

Figure 7.5 in the Dragon book is a reasonable starting point. Include in your diagram:

- the stack pointer (%rsp).
- the frame pointer (%rbp), aka "Control Link"
- locations of parameters, including `this`, which is treated as an implicit first parameter to every method.
- local variables, and the TAC temporary variables, assuming there are all stored in the stack frame.

You do not need to keep an "Access link", and the only "Saved machine status" is the return address to jump back to upon return.

3. 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.

4. 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?

5. Dragon 8.5.6

6. **[No need to write solution]** The `gcc` compiler (configured to pass all arguments on the stack) has generated the following x86_64 code for a method in an object-oriented language:

```
_f:
  pushq %rbp
  movq %rsp, %rbp
  subq $8, %rsp
  movq 32(%rbp), %rax
```

```
  addq 24(%rbp), %rax
  movq %rax, -8(%rbp)
.L2:
  movq 16(%rbp), %rax
  movq 16(%rbp), %rdx
  movq 8(%rax), %rcx
  movq 16(%rdx), %rax
  cmpq %rax, %rcx
  jge .L5
  movq 16(%rbp), %rcx
  movq 16(%rbp), %rdx
  movq -8(%rbp), %rax
  addq 8(%rdx), %rax
  movq %rax, 8(%rcx)
  jmp .L2
.L5:
  movq 16(%rbp), %rax
  movq (%rax), %rdx
  addq $64, %rdx
  pushq %rax
  movq (%rdx), %rax
  call *%rax
  addq $8, %rsp
  movq 16(%rbp), %rax
  movq %rbp, %rsp
  popq %rbp
  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) Would it be safe to remove the instruction "`addq $8, %rsp`" at the end of the program? Explain.

7. **[No need to write solution]** 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:

$$CG[\ \mathtt{x}^{\mathtt{a}} = \mathtt{s}^{\mathtt{b}} + \mathtt{t}^{\mathtt{c}}\ ] \quad \equiv \quad \begin{array}{l} \texttt{movq b(\%rbp), \%rax} \\ \texttt{addq c(\%rbp), \%rax} \\ \texttt{movq \%rax, a(\%rbp)} \end{array}$$

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

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

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

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!

- $CG[\ \mathtt{s^a = t^b}\ ]$
- $CG[\ \mathtt{x^a = s^b / t^c}\ ]$

  Hint: compile

  ```
  int main() {
    long x,y,z;
    x = y/z;
  }
  ```

  with `gcc -m64 -S` and look at the resulting assembly code.
- $CG[\ \mathtt{x^a} = (\mathtt{y^b.f})^{\mathtt{c}}\ ]$, where c is the object offset of field `f`.
- $CG[\ \mathtt{x^a[y^c] = z^b}\ ]$

(b) Augment the TAC for `A.m` with the offset information and show the compilation of those TAC instructions using $CG[\ ]$. (Note: This is tedious to do by hand — just do the first couple of instructions so that you can think about and answer the next part of the question.)

(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.