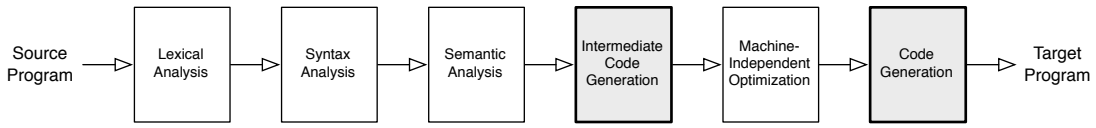
HW 7: TAC, Object Representation, Type-Based Optimizations

CSCI 434T
Fall, 2011

Overview



This problem set explores "Intermediate Code Generation", the phase of the compiler that converts our high-level AST representation into a low-level TAC representation. This lowering makes our representation closer to a realistic machine model and will greatly facilitate the generation of assembly code and optimization. After exploring this translation step, we wrap up the week's material by thinking about how objects are represented at run time, focusing on data layout and dynamic dispatch. The last few questions explore optimizations of dynamically dispatched methods. The first uses semantic information about the class hierarchy. Many optimizations will require building and manipulating control-flow graphs, but this particular optimization is a great example of a semantics-driven optimization, where analyzing a program's types will enable the compiler to generate more efficient code. The last paper examines a run-time optmization technique to reap many of the same benefits as the static optimization. That paper introduced ideas now essential to JIT optimizers.

Next week, we will examine the full details of the run-time environment for IC, which describes memory storage organization, calling conventions, and the particulars of the machine to which we will compile IC programs, namely the x86 processor. Oh yeah...

Readings

- Dragon 6.2 – 6.2.1. *(Skim)*
- Appel, Chapter 14.
- "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," Jeffrey Dean, David Grove, and Craig Chambers, ECOOP 1995.
- "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches," Urs Hölzle, Craig Chambers, David Ungar, ECOOP 1991.

Exercises

1. Be prepared to give me a status update on PA 2.
2. This problem explores how to translate a program represented as an AST into TAC, the lower-level representation that we will then translate into x86 assembly code. The specification of the TAC instruction set for this question is on the web site. As an example, consider the following while loop and its translation:

```
n = 0;
while (n < 10) {
  n = n + 1;
}
```

```
n = 0
label test
t1 = n < 10
t2 = not t1
cjump t2 end
label body
n = n + 1
jump test
label end
```

We would like to develop an automatic way to generate TAC from a program. The automatic method will be a syntax-directed translation, meaning that we will describe it as a function that takes as input a syntactic form from the source language and returns a sequence of TAC instructions that is semantically equivalent. Note that the operands of each TAC instruction are either program variable names (ie, `n`), temporary variable names introduced during translation (`t2`, `t3`), or constants (`0`, `10`, `1`). Branch instructions refer to label names generated during the translation.

More precisely, we will define a function T such that $T[s]$ is the low-level TAC representation for the high-level statement s. If e is an expression, we denote by the

$$t := T[e]$$

the low-level TAC instructions to compute e and store the result value in the variable `t`.

Expressions with subexpressions will be translated by recursively translating the subexpressions and then generating the code to combine their results appropriately. To make this concrete, suppose e is $e_1 + e_2$, then $t := T[e]$ would be:

```
t1 := T[e1]
t2 := T[e2]
t = t1 + t2
```

where the first two lines recursively translate e_1 and e_2 and store the results of those expressions in new temporary variables $t1$ and $t2$, which are then added together and stored in t. Here are a few other general cases:

e	t := $T[e]$
v	t := v (variable)
n	t := n (integer)
e.f	t1 := $T[e]$ (field access) t = t1.f
e_1[e_2] = e_3	t1 := $T[e_1]$ (array assignment) t2 := $T[e_2]$ t3 := $T[e_3]$ t1[t2] := t3

Note that I generate new temporary names whenever necessary. For more complex expressions, we simply recursively apply rules: `T[ a[i] = x * y + 1 ]` becomes the following:

```
t1 := T[a]                t1 = a              t1 = a              t1 = a
t2 := T[i]                t2 = i              t2 = i              t2 = i
t3 := T[x * y + 1]        t4 := T[x * y]      t6 := T[x]          t6 = x
                     ≡                    ≡   t7 := T[y]      ≡   t7 = y
                                              t4 = t6 * t7        t4 = t6 * t7
                          t5 := T[1]          t5 = 1              t5 = 1
                          t3 = t4 + t1        t3 = t4 + t1        t3 = t4 + t1
```

The translation scheme for statements follows the same pattern: T[`while e s`], for example, is

```
label test
t1 := T[e]
t2 = not t1
cjump t2 end
T[s]
jump test
label end
```

(a) Define T for the following syntactic forms:

- `t :=` $T[e_1$ `*` $e_2]$
- `t :=` $T[e_1$ `||` $e_2]$ (where `||` is short-circuit)
- `t :=` $T[e_1$ `&&` $e_2]$ (where `&&` is short-circuit)
- T[`if` e `then` s_1 `else` s_2]
- T[s_1`;` s_2`;` ...`;` s_n]
- `t :=` T[`f(`e_1`,` ...`,` e_n`)`]

(b) Note that these translation rules introduce more copy instructions than strictly necessary. For example, `t4 :=` T[`x * y`] becomes

```
t6 = x
t7 = y
t4 = t6 * t7
```

instead of the single statement

```
t4 = x * y
```

Describe how you would change your translation function to avoid generating these unnecessary copy statements.

(c) The original rules also use more unique temporaries than required, even after changing them to avoid the unnecessary copy instructions. For example,

T[`x = x*x+1; y = y*y-z*z; z = (x+y+w)*(y+z+w)`]

becomes the following:

```
t1 = x * x
t2 = t1 + 1
x = t2
t3 = y * y
t4 = z * z
y = t3 - t4
t5 = x + y
t6 = t5 + w
t7 = y + z
t8 = t7 + w
z = t6 * t8
```

Rewrite this to use as few temporaries as possible. Generalizing from this example, how would you change T to avoid using more temporaries than necessary.

This seems like a good idea, but can you think of any reasons why you may prefer the original translation scheme that uses more temporaries?

(d) **(Optional)** Translation of some constructs, such as nested `if` statements, `while` loops, short-circuit and/or statements may generate adjacent labels in the TAC. This is less than ideal, since the labels are clearly redundant and only one of them is needed. Illustrate an example where this occurs, and devise a scheme for generating TAC that does not generate consecutive labels.

3. Appel, 14.3. (The two questions from Appel are based on ideas from "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis" — Appel provides a good, short summary of the techniques, but you may wish to read that paper as you work on these two.)

4. Appel, 14.4.

5. The paper "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis" introduced many of the ideas behind the previous two questions. Read that paper and ponder the following:

   - When will CHA be most effective in a program? When will it be less effective?
   - What are the main limitations of CHA?
   - Section 2.2.3 discusses how CHA can be used for dynamically-typed languages like Smalltalk. Can you think of how the techniques outlined for Smalltalk could be used for Java?

6. Read "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches."

   - How does this optimization do?
   - When will this optimization be most effective? When will it be less effective?
   - Compare CHA for dynamically-typed languages to this work.
     - What is the same?
     - What is different?