CSE P 501 – Compilers

Code Shape I – Basic Constructs
Hal Perkins
Autumn 2011
Agenda

- Mapping source code to x86
  - Mapping for other common architectures follows same basic pattern
- Now: basic statements and expressions
  - We’ll go quickly since this is probably review for many and pretty straightforward
- Next: Object representation, method calls, and dynamic dispatch
Review: Variables

- For us, all data will be in either:
  - A stack frame (method local variables)
  - An object (instance variables)

- Local variables accessed via ebp
  - `mov eax, [ebp+12]`

- Instance variables accessed via an object address in a register
  - Details later
Conventions for Examples

- Examples show code snippets in isolation
- Real code generator needs to deal with things like:
  - Which registers are busy at which point in the program
  - Which registers to spill into memory (pushed onto stack or stored in stack frame) when a new register is needed and no free ones are available
- Register eax used below as a generic example
  - Rename as needed for more complex code
- Also includes a few peephole optimizations
Code Generation for Constants

- **Source**
  - 17

- **x86**
  - `mov eax,17`
  - Idea: realize constant value in a register

- **Optimization:** if constant is 0
  - `xor eax,eax`
Assignment Statement

- Source
  
  \[ \text{var} = \text{exp}; \]

- x86
  
  \(<\text{code to evaluate exp into, say, eax}>\quad \text{mov} \quad [\text{ebp+offset}_\text{var}],\text{eax}\)
UnaryMinus

- Source
  - exp
- x86
  <code evaluating exp into eax>
  neg eax

- Optimization
  - Collapse -(exp) to exp
  - Unary plus is a no-op
Binary +

- Source
  \[ \text{exp1} + \text{exp2} \]

- x86
  - \text{<code evaluating exp1 into eax>}
  - \text{<code evaluating exp2 into edx>}
  - add eax, edx
Binary +

- Optimizations
  - If exp2 is a simple variable or constant
    - add eax, exp2
  - Change exp1 + (-exp2) into exp1-exp2
  - If exp2 is 1
    - inc eax
  - Surprisingly, the Intel optimization guide recommends against this on newer processors
Binary -, *

- Same as +
  - Use sub for - (but not commutative!)
  - Use imul for *

- Optimizations
  - Use left shift to multiply by powers of 2
    - If your multiplier is really slow or you’ve got free scalar units and multiplier is busy, \(10 \times x = (8 \times x) + (2 \times x)\)
  - Use \(x + x\) instead of \(2 \times x\), etc. (faster)
  - Use dec for \(x - 1\)
Integer Division

- Ghastly on x86
  - Only works on 64 bit int divided by 32-bit int
  - Requires use of specific registers

- Source
  - $\text{exp1} \div \text{exp2}$

- x86
  - `<code evaluating exp1 into eax ONLY>`
  - `<code evaluating exp2 into ebx>`
  - `cdq`; extend to edx:eax, clobbers edx
  - `idiv ebx`; quotient in eax; remainder in edx
Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos

- In the following, $j_{\text{false}}$ is used to mean jump when a condition is false
  - No such instruction on x86
  - Will have to realize with appropriate sequence of instructions to set condition codes followed by conditional jumps
  - Normally wouldn’t actually generate the value “true” or “false” in a register
**While**

- **Source**
  
  `while (cond) stmt`

- **x86**
  
  `test: <code evaluating cond> j_false done`  
  `<code for stmt> jmp test done:`

- **Note:** In generated asm code we’ll need to create a unique label name for each loop, conditional statement, etc.
Optimization for While

- Put the test at the end
  
  ```
  jmp test
  loop: <code for stmt>
  test: <code evaluating cond>
  jtrue loop
  ```

- Why bother?
  - Pulls one instruction (jmp) out of the loop
  - Avoids a pipeline stall on jmp on each iteration
    - But modern processors can predict control flow and avoid stalls
  - Easy to do from AST or other IR; not so easy if generating code on the fly (e.g., recursive descent 1-pass compiler)
Do-While

- Source
  
do stmt while(cond);

- x86
  
  loop: <code for stmt>
  <code evaluating cond>
  j_true loop
If

- **Source**
  
  ```
  if (cond) stmt
  ```

- **x86**
  
  ```
  <code evaluating cond>
  ```
  
  ```
  j_{false} skip
  ```
  
  ```
  <code for stmt>
  ```
  
  ```
  skip:
  ```
If-Else

- **Source**
  ```
  if (cond) stmt1 else stmt2
  ```

- **x86**
  ```
  <code evaluating cond>
  jfalse else
  <code for stmt1>
  jmp done
  else:
  <code for stmt2>
  done:
  ```
Jump Chaining

- Observation: naïve code generation will produce jumps to jumps
- Optimization: if a jump has as its target an unconditional jump, change the target of the first jump to the target of the second
  - Repeat until no further changes
Boolean Expressions

- What do we do with this?
  \[ x > y \]

- Expression evaluates to true or false
  - Could generate the value in a register (0/1 or whatever the local convention is)
  - But normally we don’t want/need the value; we’re only trying to decide whether to jump
Code for exp1 > exp2

- Basic idea: designate jump target, and whether to jump if the condition is true or if false
- Example: exp1 > exp2, target L123, jump on false
  - <evaluate exp1 to eax>
  - <evaluate exp2 to edx>
  - cmp eax, edx
  - jng L123

11/2/2011
Boolean Operators: !

- Source
  
  \( \neg \exp \)

- Context: evaluate exp and jump to L123 if false (or true)

- To compile \( \neg \), reverse the sense of the test: evaluate exp and jump to L123 if true (or false)
Boolean Operators: && and ||

- In C/C++/Java/C#, these are **short-circuit operators**
  - Right operand is evaluated only if needed
- Basically, generate if statements that jump appropriately and only evaluate operands when needed
Example: Code for &&

- **Source**
  
  ```
  if (exp1 && exp2) stmt
  ```

- **x86**

  ```
  <code for exp1>
  j_false skip
  <code for exp2>
  j_false skip
  <code for stmt>
  skip:
  ```
Example: Code for `||`

- **Source**
  
  ```
  if (exp1 || exp2) stmt
  ```

- **x86**
  
  ```
  <code for exp1>
  j_true doit
  <code for exp2>
  j_false skip
  doit:
  <code for stmt>
  skip:
  ```
Realizing Boolean Values

- If a boolean value needs to be stored in a variable or method call parameter, generate code needed to actually produce it.
- Typical representations: 0 for false, +1 or -1 for true.
  - C specifies 0 and 1 if stored; we'll use that.
  - Best choice can depend on machine instructions; normally some convention is established during the primeval history of the architecture.
Boolean Values: Example

- **Source**
  \[
  \text{var} = \text{bexp} ;
  \]

- **x86**
  
  ```
  \text{<code for bexp>}
  \text{j}_{\text{false}} \text{ genFalse}
  \text{mov} \text{ eax,1}
  \text{jmp} \text{ storeIt}
  
  \text{genFalse:}
  \text{mov} \text{ eax,0}
  
  \text{storeIt: mov} \text{ [ebp+offset$_{\text{var}}$],eax ; generated by asg stmt}
  ```
Better, If Enough Registers

- **Source**
  
  \[ \text{var} = \text{bexpr} ; \]

- **x86**
  
  xor eax, eax
  <code for bexp>
  jfalse storeIt
  inc eax
  storeIt: mov [ebp+offset_var], eax ; generated by asg stmt

- Or use conditional move (movecc) instruction – avoids pipeline stalls due to conditional jumps
Better yet: setcc

- **Source**
  \[
  \text{var} = x < y;
  \]

- **x86**
  \[
  \begin{align*}
  \text{mov} & \quad [\text{ebp} + \text{offset}_x], \text{eax} & \quad ; \text{load } x \\
  \text{cmp} & \quad [\text{ebp} + \text{offset}_y], \text{eax} & \quad ; \text{compare to } y \\
  \text{setl} & \quad \text{al} & \quad ; \text{set low byte eax to 0/1} \\
  \text{movzx} & \quad \text{eax, al} & \quad ; \text{zero-extend to 32 bits} \\
  \text{storeIt: mov} & \quad [\text{ebp} + \text{offset}_\text{var}], \text{eax} & \quad ; \text{generated by asg stmt}
  \end{align*}
  \]

- GNU mnemonic for `movzx` (byte->dbl word) is `movzbl`
- Or use conditional move (movecc) instruction for sequences like \( x = y < z \ ? \ y : z \)
Other Control Flow: switch

- Naïve: generate a chain of nested if-else’s
- Better: switch is intended to allow an $O(1)$ selection, provided the set of switch values is reasonably compact
- Idea: create a 1-D array of jumps or labels and use the switch expression to select the right one
  - Need to generate the equivalent of an if statement to ensure that expression value is within bounds
Switch

**Source**

```java
switch (exp) {
    case 0: stmts0;
    case 1: stmts1;
    case 2: stmts2;
}
```

**X86**

```assembly
<put exp in eax>
    “if (eax < 0 || eax > 2)
    jmp defaultLabel”
    mov eax, swtab[eax*4]
    jmp eax
.data
    swtab:
        dd L0
        dd L1
        dd L2
.code
L0:       <stmts0>
L1:       <stmts1>
L2:       <stmts2>
```
Arrays

- Several variations
- C/C++/Java
  - 0-origin; an array with n elements contains variables $a[0]...a[n-1]$
  - Multiple dimensions; row major order
- Key step is to evaluate a subscript expression and calculate the location of the corresponding element
0-Origin 1-D Integer Arrays

- **Source**
  
  \[ \text{exp1}[\text{exp2}] \]

- **x86**

  <evaluate exp1 (array address) in eax>
  <evaluate exp2 in edx>

  address is \([\text{eax}+4*\text{edx}]\) ; 4 bytes per element
2-D Arrays

- Subscripts start with 1 (default)
- C, etc. use row-major order
  - E.g., an array with 3 rows and 2 columns is stored in this sequence: a(1,1), a(1,2), a(2,1), a(2,2), a(3,1), a(3,2)
- Fortran uses column-major order
- Exercises: What is the layout? How do you calculate location of a(i,j)? What happens when you pass array references between Fortran and C/etc. code?

- Java does not have "real" 2-D arrays. A Java 2-D array is a pointer to a list of pointers to the rows

\[
\alpha \times (3 + n \times cols) + 4) \times scale \\
\text{fixed + offset } (3, 4)
\]
a(i,j) in C/C++/etc.

- To find a(i,j), we need to know
  - Values of i and j
  - How many columns the array has
- Location of a(i,j) is
  Location of a + (i-1)*(# of columns) + (j-1)
- Can factor to pull out load-time constant part and evaluate that once – no recalculating at runtime
Coming Attractions

- Code Generation for Objects
  - Representation
  - Method calls
  - Inheritance and overriding
- Strategies for implementing code generators
- Code improvement – optimization