CS412/413
Introduction to Compilers
Radu Rugina

Lecture 17: IR Lowering
03 Mar 04

Intermediate Code

- Usually two IRs:
  - High-level IR
    - Language-independent
    - (but closer to language)
  - Low-level IR
    - Machine independent
    - (but closer to machine)

  C
  Fortran → HIR → LIR
  Pascal
  Pentium
  Java bytecode
  Alpha

High-level IR

- Tree node structure very similar to the AST
- Contains high-level constructs common to many languages
  - Expression nodes
  - Statement nodes
- Expression nodes for:
  - Integers and program variables
  - Binary operations: e1 OP e2
  - Arithmetic operations
  - Logic operations
  - Comparisons
  - Unary operations: OP e
  - Array accesses: e1[e2]

High-level IR

- Statement nodes:
  - Block statements (statement sequences): (s1, ..., sN)
  - Variable assignments: v = e
  - Array assignments: e1[e2] = e3
  - If-then-else statements: if c then s1 else s2
  - If-then statements: if c then s
  - While loops: while (c) s
  - Function call statements: f(e1, ..., eN)
  - Return statements: return or return e
- May also contain:
  - For loop statements: for(v = e1 to e2) s
  - Break and continue statements
  - Switch statements: switch(e) { v1: s1, ..., vN: sN }

Low-Level IR

- Low-level representation is essentially an instruction set for an abstract machine

- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    a = b OP c
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)

Three-Address Code

- In this class: three-address code
  a = b OP c

- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as:
  (a, b, c, OP)

- Example:
  a = (b+c)\(^*(-e)\);
  t1 = b + c
  t2 = - e
  a = t1 * t2
Low IR Instructions

- Assignment instructions:
  - Binary operations: \( a = b \) OP \( c \)
  - Arithmetic: ADD, SUB, MUL, DIV, MOD
  - Logic: AND, OR, XOR
  - Comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
- Unary operation: \( a = \) OP \( b \)
- Arithmetic MINUS or logic NEG
- Copy instruction: \( a = b \)
- Load/store: \( a = \) OP \( b \)
- Other data movement instructions

Low IR Instructions (Ctd)

- Flow of control instructions:
  - Label \( L \) : label instruction
  - Jump \( L \) : Unconditional jump
  - CJump \( a \) \( L \) : conditional jump
- Function call
  - Call \( f(a_1, ..., a_n) \)
  - \( a = \) call \( f(a_1, ..., a_n) \)
  - Is an extension to quads
- ... IR describes the Instruction Set of an abstract machine

Example

\[
\begin{align*}
m &= 0; \\
\text{if } (c == 0) \{ \\
  m &= m + n; \\
\text{\textbf{else}} \{ \\
  m &= m + n; \\
\}
\end{align*}
\]

\[
\begin{align*}
m &= 0 \\
t &= c == 0 \\
t\text{jump t1 trueb} \\
m &= m + n \\
\text{jump end} \\
l\text{label trueb} \\
t2 &= n \times n \\
m &= m + t2 \\
\text{label end}
\end{align*}
\]

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST

Notation

- Use the following notation:
  \( T[e] \) = the low-level IR representation of high-level IR construct \( e \)
- \( T[e] \) is a sequence of Low-level IR instructions
- If \( e \) is an expression (or a statement expression), it represents a value
- Denote by \( t = T[e] \) the low-level IR representation of \( e \), whose result value is stored in \( t \)
- For variable \( v \): \( t = T[v] \) is the copy instruction \( t = v \)

Translating Expressions

- Binary operations: \( t = T[e1 \text{ OP } e2] \)
  (arithmetic operations and comparisons)
  \[
  \begin{align*}
t1 &= T[e1] \\
t2 &= T[e2] \\
t &= t1 \text{ OP } t2
\end{align*}
\]
- Unary operations: \( t = T[e \text{ OP } e] \)
  \[
  \begin{align*}
t1 &= T[e] \\
t &= \text{ OP } t1
\end{align*}
\]
Translating Boolean Expressions

- \( t = T[e_1 \lor e_2] \)

\[
\begin{align*}
  t_1 &= T[e_1] \\
  t_2 &= T[e_2] \\
  t &= t_1 \lor t_2
\end{align*}
\]

- ... how about short-circuit OR?
- Should compute \( e_2 \) only if \( e_1 \) evaluates to false

Translating Short-Circuit OR

- Short-circuit OR: \( t = T[e_1 \lor e_2] \)

\[
\begin{align*}
  t &= T[e_1] \\
  \text{jump } t \text{ Lend} \\
  t &= T[e_2] \\
  \text{label } L\text{end}
\end{align*}
\]

- ... how about short-circuit AND?

Translating Short-Circuit AND

- Short-circuit AND: \( t = T[e_1 \land e_2] \)

\[
\begin{align*}
  t &= T[e_1] \\
  \text{jump } t \text{ Lnext} \\
  \text{jump } L\text{next} \\
  t &= T[e_2] \\
  \text{label } L\text{end}
\end{align*}
\]

Another Translation

- Short-circuit AND: \( t = T[e_1 \land e_2] \)

\[
\begin{align*}
  t &= T[e_1] \\
  \text{jump } t \text{ Lend} \\
  t &= T[e_2] \\
  \text{label } L\text{end}
\end{align*}
\]

Array and Field Accesses

- Array access: \( t = T[v[e]] \)

\[
\begin{align*}
  t_1 &= T[e] \\
  t &= v[t_1]
\end{align*}
\]

- Field access: \( t = T[e_1.f] \)

\[
\begin{align*}
  t_1 &= T[e_1] \\
  t &= t_1.f
\end{align*}
\]

Translating Statements

- Statement sequence: \( T[s_1; s_2; \ldots; s_N] \)

\[
\begin{align*}
  T[s_1] \\
  T[s_2] \\
  \ldots \\
  T[s_N]
\end{align*}
\]

- IR instructions of a statement sequence = concatenation of IR instructions of statements
**Assignment Statements**

- Variable assignment: \( T[\ v = e \] \)
  \[ \text{var-assign} \]
  \[ v = T[\ e \] \]
  \[ v \ e \]

- Array assignment: \( T[\ v[e1] = e2 \] \)
  \[ t1 = T[\ e1 \] \]
  \[ \text{array-assign} \]
  \[ t2 = T[\ e2 \] \]
  \[ v[t1] = t2 \]
  \[ v \ e1 \ e2 \]

**Translating If-Then-Else**

- \( T[\ \text{if} \ (e) \ \text{then} \ s1 \ \text{else} \ s2 \] \)
  \[ t1 = T[\ e \] \]
  \[ \text{fjump t1 Lfalse} \]
  \[ T[\ s1 \] \]
  \[ \text{if-then-else} \]
  \[ \text{jump Lend} \]
  \[ \text{label Lfalse} \]
  \[ T[\ s2 \] \]
  \[ \text{label Lend} \]

**Translating If-Then**

- \( T[\ \text{if} \ (e) \ \text{then} \ s \] \)
  \[ t1 = T[\ e \] \]
  \[ \text{fjump t1 Lend} \]
  \[ T[\ s \] \]
  \[ \text{if-then} \]
  \[ \text{label Lend} \]

**While Statements**

- \( T[\ \text{while} \ (e) \ \{ \ s \} \] \)
  \[ t1 = T[\ e \] \]
  \[ \text{fjump t1 Lend} \]
  \[ T[\ s \] \]
  \[ \text{while} \]
  \[ \text{jump Ltest} \]
  \[ \text{label Ltest} \]

**Switch Statements**

- \( T[\ \text{switch} \ (e) \ \{ \ \text{case} \ v1: \ s1, ..., \text{case} \ vN: \ sN \} \] \)
  \[ t = T[\ e \] \]
  \[ \text{switch} \]
  \[ \text{case} \ v1 \ s1 \ ...
  \[ \text{switch} \]
  \[ \text{vN sN} \]
Statement Expressions

- So far: statements which do not return values
- Easy extensions for statement expressions:
  - Block statements
  - If-then-else
  - Assignment statements
- \( t = T[ s ] \) is the sequence of low IR code for statement \( s \), whose result is stored in \( t \)

Block Statements

- \( t = [ s_1; s_2; \ldots; s_N ] \)
  
  \[
  \begin{array}{c}
  [ s_1 ] \\
  [ s_2 ] \\
  \vdots \\
  t = [ s_N ]
  \end{array}
  \]

- Result value of a block statement = value of last statement in the sequence

Assignment Statements

- \( t = [ v = e ] \)
  
  \[
  \begin{array}{c}
  v = [ e ] \\
  t = v
  \end{array}
  \]

- Result value of an assignment statement = value of the assigned expression