CS42/413

Introduction to Compilers
Radu Rugina

Lecture 16: Efficient Translation to Low IR
25 Feb 02

Intermediate Representation

- High IR: captures high-level language constructs
  - Has a tree structure very similar to AST
  - Has expression nodes (ADD, SUB, etc) and statement nodes (if-then-else, while, etc)
- Low IR: captures low-level machine features
  - Is a instruction set describing an abstract machine
  - Has arithmetic/logic instructions, data movement instructions, branch instructions, function calls

IR Lowering

- Use temporary variables for the translation
- Temporary variables in the Low IR store intermediate values corresponding to the nodes in the High IR

<table>
<thead>
<tr>
<th>High IR</th>
<th>Low IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>t1 = a</td>
</tr>
<tr>
<td>SUB</td>
<td>t2 = b</td>
</tr>
<tr>
<td>ADD</td>
<td>t3 = t1 - t2</td>
</tr>
<tr>
<td></td>
<td>t4 = b</td>
</tr>
<tr>
<td></td>
<td>t5 = c</td>
</tr>
<tr>
<td></td>
<td>t5 = t4 + t5</td>
</tr>
<tr>
<td></td>
<td>t = t3 * t5</td>
</tr>
</tbody>
</table>

Lowering Methodology

- Define simple translation rules for each High IR node
  - Arithmetic: el + e2, el - e2, etc.
  - Logic: el AND e2, el OR e2, etc.
  - Array access expressions: el[e2]
  - Statements: if (e) then sl else s2, while (e) sl, etc.
  - Function calls (f1, ..., en)
- Recursively traverse the High IR trees and apply the translation rules
- Can handle nested expressions and statements

Efficient Lowering Techniques

- How to generate efficient Low IR:
  1. Reduce number of temporaries
     1. Don't use temporaries that duplicate variables
     2. Use "accumulator" temporaries
     3. Reuse temporaries in Low IR
  2. Don't generate multiple adjacent label instructions
  3. Encode conditional expressions in control flow
No Duplicated Variables

- **Basic algorithm:**
  - Translation rules recursively traverse expressions until they reach terminals (variables and numbers).
  - Then translate \( t = [v] \) into \( t = v \) for variables.
  - And translate \( t = [n] \) into \( t = n \) for constants.
- **Better:**
  - Terminate recursion one level before terminals.
  - Need to check at each step if expressions are terminals.
  - Recursively generate code for children only if they are non-terminal expressions.

---

Example

- \( t = [(a+b)c] \)
- Operand \( e1 = a+b \), is not terminal
- Operand \( e2 = c \), is terminal
- Translation: \( t1 = [e1] \)
  - \( t = t1 \star c \)
  - Recursively generate code for \( t1 = [e1] \)
  - For \( e1 = a+b \), both operands are terminals
  - Code for \( t1 = [e1] \) is \( t1 = b + c \)
  - Final result:
    - \( t1 = b + c \)
    - \( t = t1 \star c \)

---

No Duplicated Variables

- \( t = [e1 \ op \ e2] \)
  - \( t = [e1] \), if \( e1 \) is not terminal
  - \( t = [e2] \), if \( e2 \) is not terminal
  - \( t = x1 \ op \ x2 \)
  - where:
    - \( x1 = t \), if \( e1 \) is not terminal
    - \( x1 = e1 \), if \( e1 \) is terminal
    - \( x2 = e2 \), if \( e2 \) is not terminal
    - \( x2 = e2 \), if \( e2 \) is terminal
- Similar translation for statements with conditional expressions: if, while, switch

---

Reuse Temporaries

- **Idea:** in the translation of \( t = [e1 \ op \ e2] \)
  - \( t = [e1] \), \( t' = [e2] \), \( t = t \ op \ t' \)
  - temporary variables from the translation of \( e1 \) can be reused in the translation of \( e2 \).
- **Observation:** temporary variables compute intermediate values, so they have limited lifetime.
- **Algorithm:**
  - Use a stack of temporaries
  - This corresponds to the stack of the recursive invocations of the translation function \( t = [e] \)
  - All the temporaries on the stack are alive

---

Accumulator Temporaries

- Use the same temporary variables for operands and result.
- Translate \( t = [e1 \ op \ e2] \) as:
  - \( t = [e1] \)
  - \( t1 = [e2] \)
  - \( t = t \ op \ t1 \)
- Example: \( t = [(a+b)c] \)
  - \( t = b + c \)
  - \( t = t \star c \)

---

Reuse Temporaries

- **Implementation:** use counter \( c \) to implement the stack.
  - Temporaries \( t(0), \ldots, t(c) \) are alive.
  - Temporaries \( t(c+1), t(c+2), \ldots \) can be reused.
  - Push means increment \( c \), pop means decrement \( c \)
- In the translation of \( t(c) = [e1 \ op \ e2] \)
  - \( t(c) = [e1] \)
  - \( ................. \ c = c+1 \)
  - \( t(c) = [e2] \)
  - \( ................. \ c = c-1 \)
  - \( t(c) = t(c) \ op \ t(c+1) \)
Example

\[ t_0 = ((a*b) + (c*d) - (e*f)) \]
\[ t_0 = a + b \]
\[ \quad C = C + 1 \]
\[ t_1 = e2 \]
\[ \quad C = C + 1 \]
\[ t_2 = e+f \]
\[ t_1 = t_1 - t_2 \]
\[ \quad C = C - 1 \]
\[ t_0 = t_0 + t_1 \]

Trade-offs

- Benefits of fewer temporaries:
  - Smaller symbol tables
  - Smaller analysis information propagated during dataflow analysis
- Drawbacks:
  - Some temporaries store multiple values
  - Some analysis results may be less precise
  - More difficult to reconstruct expression trees (which may be convenient for instruction selection)
- Possible compromise:
  - Use different temporaries for intermediate expression values in each statement
  - Use same temporaries in different statements

No Adjacent Labels

- Translation of control flow constructs (if, while, switch) and short-circuit conditionals generates label instructions
- Nested if/while/switch statements and nested short-circuit AND/OR expressions may generate adjacent labels
- Simple solution: have a second pass that merges adjacent labels
  - And a third pass to adjust the branch instructions
- More efficient: backpatching
  - Directly generate code without adjacent label instructions
  - Code has placeholders for jump labels, fill in labels later

Backpatching

- Keep track of the return label (if any) of translation of each
  High IR node: \( t = [e, L] \)
- No end label for a translation: \( L = \emptyset \)
- Translate \( t = [e1 \text{ SC-OR} e2, L] \) as:
  \[ t_1 = [e_1, l_1] \]
  \[ \text{jump} \ t_1 \]
  \[ t_1 = [e_2, L_2] \]
- If \( L_2 = \emptyset \) \( L \) is new label; add \text{‘label’} to code
- If \( L_2 \neq \emptyset \) \( L_2 \) is new label; don’t add label instruction
- Then fill placeholder \( L \) in jump instruction and set \( L = \text{end label} \) of the SC-OR construct

Example

\[ t = [(a \text{ OR } (c \text{ OR } d \text{ OR } e)), L] \]
\[ t = a < b \]
\[ \text{jump} \ t_1 \]
\[ t = [(c < d \text{ OR } d < e) \text{ OR } L'] \]
\[ t = c < d \]
\[ \text{jump} \ t_1' \]
\[ t = c < d \]
\[ \text{label Lend} \]

- Backpatch \( t = [(c \text{ OR } d \text{ OR } e), L'] : L' = \text{Lend} \)
- Backpatch \( t = [(a \text{ OR } (c \text{ OR } d \text{ OR } e)), L] : L = L' = \text{Lend} \)

Backpatching

- Similar rules for end labels of short-circuit OR, and for control
  flow statements: if-then-else, if-then, while, switch
- Keep track of end labels for each of these constructs
- Translations may begin with a label while statements start
  with a label instruction for the test condition
- For a statement sequence \( s_1; s_2 \) : should merge end label of
  \( s_1 \) with start label of \( s_2 \)
  - Need to pass in the end label of \( s_1 \) to the recursive
    translation of \( s_2 \)
  - Translation of each statement: receives end label of
    previous statement, returns end label of current statement
Encode Booleans in Control-Flow

- Consider \[ (a < b \land c < d) \implies x = y \]\n
  \[
  \begin{align*}
  t &= a < b \\
  \text{if } t < a < b \land c < d \\
  t &= c < d \\
  \text{label } l1 \\
  \text{jump } t2 \\
  x &= y \\
  \text{label } l2
  \end{align*}
  \]
  Condition: \( t = a < b \land c < d \)
  Control flow: \( t = a < b \land c < d \)

  \[ \text{... can we do better?} \]

How It Works

- For each boolean expression \( e \):
  \[
  [ e, L1, L2 ]
  \]
  is the code that computes \( e \) and branches to \( L1 \) if \( e \) evaluates to true, and to \( L2 \) if \( e \) evaluates to false

  - New translation: \( [ \text{if } (e) \text{ then } s ] \)
    \[
    [ e, \text{L1, L2} ]
    \]
    label L1
    \[
    [ s ]
    \]
    label L2
  - Also remove sequences \( \text{jump L1, label L1} \)

Define New Translations

- Must define:
  \[
  [ s ] \text{ for } \text{if, while statements}
  \]
  \[
  [ e, L1, L2 ] \text{ for boolean expressions } e
  \]
  \[
  [ \text{if } (e) \text{ then } s1 \text{ else } s2 ]
  \]
  \[
  [ e, L1, L2 ]
  \]
  label L1
  \[
  [ s1 ]
  \]
  jump Lend
  \[
  [ s2 ]
  \]
  label L2
  \[
  [ \text{Lend} ]
  \]

While Statement

- \( [ \text{while } (e) s ] \)
  \[
  \text{label Ltest}
  [ e, L1, L2 ]
  \]
  label L1
  \[
  [ s ]
  \]
  jump Ltest
  \[
  \text{label L2}
  \]
  Code branches directly to end label when \( e \) evaluates to false

Boolean Expression Translations

- \( [ \text{true, L1, L2} ] : \) jump L1
- \( [ \text{false, L1, L2} ] : \) jump L2
- \( [ e1 \text{ SC-OR } e2, L1, L2 ] \)
  \[
  [ e1, L1, \text{Lnext} ]
  \]
  label Lnext
  \[
  [ e2, L1, L2 ]
  \]
- \( [ e1 \text{ SC-AND } e2, L1, L2 ] \)
  \[
  [ e1, \text{Lnext, L2} ]
  \]
  label Lnext
  \[
  [ e2, L1, L2 ]
  \]