CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 19: Efficient IL Lowering
5 March 08
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

High IR

Low IR

```
MUL
  |
  v
SUB  ADD
  /  /
 a  b  c  d
```

\[ t_1 = a - b \]
\[ t_2 = c + d \]
\[ t = t_1 \times t_2 \]
Lowering Methodology

• Define simple translation rules for each High IR node
  - Arithmetic: e1 + e2, e1 - e2, etc.
  - Logic: e1 AND e2, e1 OR e2, etc.
  - Array access expressions: e1[e2]
  - Statements: if (e) then s1 else s2, while (e) s1, etc.
  - Function calls f(e1, …, eN)

• Recursively traverse the High IR trees and apply the translation rules

• Can handle nested expressions and statements
Notation

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

• In these translations, expressions may be nested;
• Translation recurses on the expression structure

Example: $t = T[ (a - b) \times (c + d) ]$

- $t1 = a$
- $t2 = b$
- $t3 = t1 - t2$
- $t4 = b$
- $t5 = c$
- $t5 = t4 + t5$
- $t = t3 \times t5$

\[ T[ (a - b) ] \]
\[ T[ (c + d) ] \]
Nested Statements

- Same for statements: recursive translation

Example: \( T[ \text{if } c \text{ then } \text{if } d \text{ then } a = b ] \)

\[
\begin{align*}
t1 &= c \\
fjump t1 \text{ Lend1} \\
t2 &= d \\
fjump t2 \text{ Lend2} \\
t3 &= b \\
a &= t3 \\
\text{label Lend2} \\
\text{label Lend1}
\end{align*}
\]
IR Lowering Efficiency

```
t1 = c
fjump t1 Lend1

t2 = d
fjump t2 Lend2

t3 = b
a = t3
label Lend2
label Lend1
```

```
if-then
  c
if-then
  d =
    a
    b
```

```
fjump c Lend
fjump d Lenda = b
Label Lend
```
Efficient Lowering Techniques

• How to generate efficient Low IR:

1. Reduce number of temporaries
   a) Don’t use temporaries that duplicate variables
   b) Use “accumulator” temporaries
   c) Reuse temporaries in Low IR

2. Don’t generate multiple adjacent label instructions

3. Encode conditional expressions in control flow

4. Eliminate jumps to unconditional jumps
No Duplicated Variables

• Basic algorithm:
  - Translation rules recursively traverse expressions until they reach terminals (variables and numbers)
  - Then translate $t = T[v]$ into $t = v$ for variables
  - And translate $t = T[n]$ into $t = n$ for constants

• Better:
  - Terminate recursion one level before terminals
  - Need to check at each step if expressions are terminals and only recursively generate code for child if it is a non-terminal expression
No Duplicated Variables

• \( t = T[e_1 \text{ OP } e_2 ] \)
  \[
  \begin{align*}
  t_1 &= T[e_1], \text{ if } e_1 \text{ is not terminal} \\
  t_2 &= T[e_2], \text{ if } e_2 \text{ is not terminal} \\
  t &= x_1 \text{ OP } x_2 
  \end{align*}
  \]
  where:
  \[
  \begin{align*}
  x_1 &= \textbf{if } e_1 \text{ is terminal } \textbf{then } e_1 \textbf{ else } t_1 \\
  x_2 &= \textbf{if } e_2 \text{ is terminal } \textbf{then } e_2 \textbf{ else } t_2 
  \end{align*}
  \]

• Similar translation for statements with conditional expressions: if, while, switch
Example

- $t = T[ (a+b)\ast c ]$
- Operand $e_1 = a+b$, is not terminal
- Operand $e_2 = c$, is terminal
- Translation: $t_1 = T[ e_1 ]$
  
  $t = t_1 \ast c$

- Recursively generate code for $t_1 = T[ e_1 ]$
- For $e_1 = a+b$, both operands are terminals
- Code for $t_1 = T[ e_1 ]$ is $t_1 = a+b$

- Final result: $t_1 = a + b$
  
  $t = t_1 \ast c$
Accumulator Temporaries

- Use the same temporary variables for operands and result

- Translate $t = T[ e_1 \text{ OP } e_2 ]$ as:
  
  $$
  t = T[ e_1 ] \\
  t_1 = T[ e_2 ] \\
  t = t \text{ OP } t_1
  $$

- Example: $t = T[ (a+b)*c ]$
  
  $$
  t = a + b \\
  t = t * c
  $$
Reuse Temporaries

- **Idea:** in the translation of $t = T[ e_1 \text{ OP } e_2 ]$ as:
  - $t = T[ e_1 ]$, $t' = T[ e_2 ]$, $t = t \text{ OP } t'$
  temporary variables from the translation of $e_1$ can be reused in the translation of $e_2$

- **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 functions $t = T[ e ]$
  - All the temporaries on the stack are alive
Reuse Temporaries

- **Implementation**: use counter c to implement the stack
  - Temporaries t(0), ..., t(c) are alive
  - Temporaries t(c+1), t(c+2), ... can be reused
  - Push means increment c, pop means decrement c

- In the translation of \( t(c) = T[\ e_1 \ OP \ e_2 \ ] \)
  
  \[
  \begin{align*}
  t(c) &= T[\ e_1 ] \\
  \quad &\quad \quad \quad \quad \quad \quad \quad \quad \text{c} = \text{c}+1 \\
  t(c) &= T[\ e_2 ] \\
  \quad &\quad \quad \quad \quad \quad \quad \quad \quad \text{c} = \text{c}-1 \\
  t(c) &= t(c) \ OP \ t(\text{c}+1)
  \end{align*}
  \]
Example

- $t_0 = T[ ((c*d) - (e*f)) + (a*b)]$
  - $c = 0$
  - $t_0 = T[ e_0 ]$
  - $t_0 = c*d$
    - $c = c + 1$
    - $t_1 = e*f$
      - $c = c - 1$
      - $t_0 = t_0 - t_1$
  - $t_1 = a * b$
    - $c = c + 1$
  - $t_0 = t_0 + t_1$

- $\vdots$
Trade-offs

• Benefits of fewer temporaries:
  - Smaller symbol tables
  - Smaller analysis information propagated during dataflow analysis

• Drawbacks:
  - Same temporaries store multiple values
  - Some analysis results may be less precise
  - Also harder to reconstruct expression trees (albeit, possibly more convenient for instruction selection)

• Possible compromise:
  - Different temporaries for intermediate values in each statement
  - Reuse temporaries for 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.
Encode Booleans in Control-Flow

• Consider $T[ \text{if ( } a<b \text{ SC-AND } c<d \text{ ) then } x = y; ]$

\[
\begin{align*}
&\text{Condition: } t = a<b \text{ SC-AND } c<d \\
&\text{Control flow: if (t) } x = y
\end{align*}
\]

• ...can we do better?
Encode Booleans in Control-Flow

• Consider \( T[ \text{if } ( a < b \text{ SC-AND } c < d ) \text{ then } x = y; ] \)

\[
\begin{align*}
\text{t} &= a < b \\
\text{fjump } t \text{ L1} \\
\text{t} &= c < d \\
\text{label L1} \\
\text{fjump } t \text{ L2} \\
\text{x} &= y \\
\text{label L2}
\end{align*}
\]

\[
\begin{align*}
\text{t} &= a < b \\
\text{fjump } t \text{ L2} \\
\text{t} &= c < d \\
\text{fjump } t \text{ L2} \\
\text{x} &= y \\
\text{label L2}
\end{align*}
\]

Condition and control flow

• If \( t = a < b \) is false, program branches to label L2
How It Works

• For each boolean expression e, and b either true or false:

\[ T[e, L, b] \]

is the code that computes e and branches to L if e evaluates to b, and falls through to the next sequential instruction on !b

• Must redefine \( T[s] \) for if and while statements to use \( T[e, L, b] \) for Boolean expressions
Define New Translations

- $T[\text{if}(e) \text{ then } s1 \text{ else } s2 ]$
  
  $T[ e, L, \text{false} ]$
  
  $T[ s1 ]$
  
  jump Lend
  
  label L
  
  $T[ s2 ]$
  
  label Lend

- $T[\text{if}(e) \text{ then } s ]$
  
  $T[ e, L, \text{false} ]$
  
  $T[ s ]$
  
  label L
While Statement

- $T[\, \text{while (e) s } ]$

  label Ltest

  $T[\, e, L, \text{false} ]$

  $T[\, s ]$

  jump Ltest

  label L
SC-Boolean Expression Translations

- \[ T[v, L, b] : \text{if } b \text{ then } \text{tjump } v, L \text{ else } \text{fjump } v, L \]
- \[ T[!e, L, b] : T[e, L, !b] \]
- \[ T[e1 \text{ SC-OR } e2, L, \text{true}] \]
  \[ T[e1, L, \text{true}] \]
  \[ T[e2, L, \text{true}] \]
- \[ T[e1 \text{ SC-AND } e2, L, \text{false}] \]
  \[ T[e1, L, \text{false}] \]
  \[ T[e2, L, \text{false}] \]
- \[ T[e1 \text{ SC-OR } e2, L, \text{false}] \]
  \[ T[e1, L_{\text{next}}, \text{true}] \]
  \[ T[e2, L, \text{false}] \]
  \[ \text{label } L_{\text{next}} \]
- \[ T[e1 \text{ SC-AND } e2, L, \text{true}] \]
  \[ T[e1, L_{\text{next}}, \text{false}] \]
  \[ T[e2, L, \text{true}] \]
  \[ \text{label } L_{\text{next}} \]
Eliminate Jumps to Unconditional Jumps

• Example

\[
T[ \text{if } a \text{ then if } b \text{ then } c=d \text{ else } e=f \text{ else } g=h ]
\]

\[
fjump a \text{ L1}
\]
\[
fjump b \text{ L2}
\]
\[
c = d
\]
\[
\text{jump Lend2}
\]
\[
\text{label L2}
\]
\[
e = f
\]
\[
\text{label Lend2}
\]
\[
\text{jump Lend1}
\]
\[
\text{label L1}
\]
\[
g = h
\]
\[
\text{label Lend1}
\]
Eliminate Jumps to Unconditional Jumps

- Example

\[ T[ \text{if } a \text{ then if } b \text{ then } c=d \text{ else } e=f \text{ else } g=h ] \]

\[
\begin{align*}
&\text{fjump } a \text{ L1} \\
&\text{fjump } b \text{ L2} \\
&\text{c} = \text{d} \\
&\text{jump Lend1} \\
&\text{label L2} \\
&\text{e} = \text{f} \\
&\text{jump Lend1} \\
&\text{label L1} \\
&\text{g} = \text{h} \\
&\text{label Lend1}
\end{align*}
\]
Eliminate Jumps to Unconditional Jumps

- Each set of jumps to jumps that end in the same label form a tree (with the ultimate label as root)
- Traverse tree and retarget all jumps to the root label
Eliminate Jumps to Jumps

- Each set of jumps to jumps that end in the same label form a tree (with the ultimate label as root)
- Traverse tree and retarget all jumps to the root label

```
jump L3
  label L1
    jump L3
  label L2
    jump L3
    label L3
      <not jump>
```