CS412/413

Introduction to Compilers

Radu Rugina

Lecture 16: Efficient Translation to Low IR

25 Feb 02

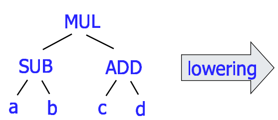
Intermediate Representation

- 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

High IR



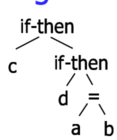
Low IR

```
t1 = a
t2 = b
t3 = t1 – t2
t4 = b
t5 = c
t5 = t4 + t5
t = t3 * t5
```


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

IR Lowering Efficiency




```
t1 = c
fjump t1 Lend1
t2 = d
fjump t2 Lend2
t3 = b
a = t3
label Lend2
label Lend1
```

```
fjump c Lend
fjump d Lend
a = b
Label Lend
```


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

No Duplicated Variables

- t = ⟦ e1 OP e2 ⟧
  t1 = ⟦ e1 ⟧, if e1 is not terminal
  t2 = ⟦ e2 ⟧, if e2 is not terminal
  t = x1 OP x2
  where:
  x1 = t1, if e1 is not terminal
  x1 = e1, if e1 is terminal
  x2 = t2, if e2 is not terminal
  x2 = e2, if e2 is terminal
- Similar translation for statements with conditional expressions: if, while, switch

Example

- t = ⟦ (a+b)*c ⟧
- Operand e1 = a+b, is not terminal
- Operand e2 = c, is terminal
- Translation: t1 = ⟦ e1 ⟧
  t = t1 * 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 * c

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 * c

Reuse Temporaries

- Idea: in the translation of t = ⟦ e1 OP e2 ⟧ as:
  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 functions 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) = ⟦ e1 OP e2 ⟧
  t(c) = ⟦ e1 ⟧
  .... c = c+1
  t(c) = ⟦ e2 ⟧
  .... c = c-1
  t(c) = t(c) OP t(c+1)

Example

- t0 = ⟦ (a*b) + ((c*d) – (e*f)) ⟧

```
t0 = a * b
                 c = c+1
                              t1 = c*d
                                          c = c+1
t1 = ⟦ e2 ⟧                   t2 = e*f
                                          c = c-1
                              t1 = t1 – t2
                 c = c-1
t0 = t0+t1
```


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 (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 = ∅
- Translate t = ⟦ e1 SC-OR e2, L ⟧ as:

```
t1 = ⟦ e1, L1 ⟧
tjump t1 L
t1 = ⟦ e2, L2 ⟧
```

- If L2 = ∅: L is new label; add 'label L' to code
- If L2 ≠ ∅: L = L2; don't add label instruction
- Then fill placeholder L in jump instruction and set L = end label of the SC-OR construct

Example

- t = ⟦ (a < b) OR (c<d OR d<e), L ⟧

```
t = a < b
tjump t1 L
                              t = c < d
t = ⟦ c<d OR d<e, L' ⟧        fjump t L'
                              t = d<e
                              label Lend
```

- Backpatch t = ⟦ c<d OR d<e, L' ⟧ : L' = Lend
- Backpatch t = ⟦ (a<b) OR (c<d OR d<e), L ⟧ : L = L' = 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 s1;s2 : should merge end label of s1 with start label of s2
  - Need to pass in the end label of s1 to the recursive translation of s2
  - Translation of each statement: receives end label of previous statement, returns end label of current statement

Encode Booleans in Control-Flow

- Consider ⟦ if (a<b AND c<d) x = y; ⟧

```
t = a<b
fjump t L1
t = c<d
label L1        } Condition: t = a<b AND c<d
fjump t L2
x = y
label L2        } Control flow: if (t) x = y
```

- … can we do better?

Encode Booleans in Control-Flow

- Consider ⟦ if (a<b AND c<d) x = y; ⟧

```
t = a<b            t = a<b
fjump t L1         fjump t L2
t = c<d            t = c<d
label L1           fjump t L2
fjump t L2         x = y
x = y              label L2    } Condition and control flow
label L2
```

- If t = a<b is false, program branches to label L2
- Encode (a<b) == false to branch directly to the end label

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: ⟦ if(e) then s ⟧

```
⟦ e, L1, L2 ⟧
label L1
⟦ s ⟧
label L2
```

- Also remove sequences 'jump L, label L'

Define New Translations

- Must define:
  ⟦ s ⟧ for if, while statements
  ⟦ e, L1, L2 ⟧ for boolean expressions e
- ⟦ if(e) then s1 else s2 ⟧

```
⟦ e, L1, L2 ⟧
label L1
⟦ s1 ⟧
jump Lend
label L2
⟦ s2 ⟧
label Lend
```


While Statement

- ⟦ while (e) s ⟧

```
label Ltest
⟦ e, L1, L2 ⟧
label L1
⟦ s ⟧
jump Ltest
label L2
```

- Code branches directly to end label when e evaluates to false

Boolean Expression Translations

- ⟦ true, L1, L2 ⟧ : jump L1
- ⟦ false, L1, L2 ⟧ : jump L2
- ⟦ e1 SC-OR e2, L1, L2 ⟧

```
⟦ e1, L1, Lnext ⟧
label lnext
⟦ e2, L1, L2 ⟧
```

- ⟦ e1 SC-AND e2, L1, L2 ⟧

```
⟦ e1, Lnext, L2 ⟧
label lnext
⟦ e2, L1, L2 ⟧
```