CS412/413

Introduction to Compilers
Radu Rugina

Lecture 18: Efficient IR Lowering
28 Feb 03

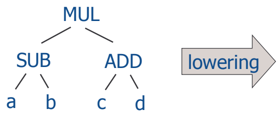
Intermediate Representation

- 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 - b
t2 = c + d
t = t1 * t2

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

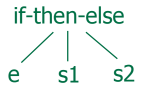
Notation

- 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 If-Then-Else

- T[if (e) then s1 else s2]

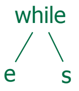
```
t1 = T[ e ]
fjump t1 Lfalse
T[ s1 ]
jump Lend
label Lfalse
T[ s2 ]
label Lend
```



While Statements

- T[while (e) { s }]

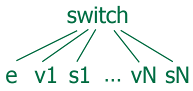
```
label Ltest
t1 = T[ e ]
fjump t1 Lend
T[ s ]
jump Ltest
label Lend
```



Switch Statements

- T[switch (e) { case v1: s1, …, case vN: sN }]

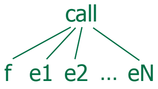
```
t = T[ e ]
c = t != v1
tjump c L2
T[ s1 ]
jump Lend
label L2
c = t != v2
tjump c L3
T[ s2 ]
jump Lend
…
label LN
c = t != vN
tjump c Lend
T[ sN ]
label Lend
```



Call and Return Statements

- T[call f(e1, e2, …, eN)]

```
t1 = T[ e1 ]
t2 = T[ e2 ]
…
tN = T[ eN ]
call f(t1, t2, …, tN)
```



- T[return e]

```
t = T[ e ]
return t
```

Statement Expressions

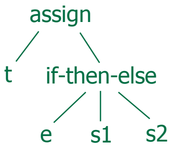
- So far: statements which do not return values
- Easy extensions for statement expressions:
  - Block statements
  - If-then-else
  - Assignment statements
- t = T[s] is the sequence of low IR code for statement s, whose result is stored in t

---

Statement Expressions

- t = T[if (e) then s1 else s2]

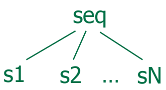
```
t1 = T[ e ]
cjump t1 Ltrue
t = T[ s2 ]
jump Lend
label Ltrue
t = T[ s1 ]
label Lend
```



Block Statements

- t = T[s1; s2; …; sN]

```
T[ s1 ]
T[ s2 ]
 …
t = T[ sN ]
```



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

Assignment Statements

- t = T[v = e]

v = T[e]
t = v

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

---

Lowering Field and Array Accesses

- Can lower field and array accesses to load/store
- Lowering array read:

a = b[i] → t1 = i*s
t2 = b + t1
a = *t2

- Lowering field read:

a = b.f → t1 = b+offset(f)
a = *t1

Nested Expressions

- In these translations, expressions may be nested;
- Translation recurses on the expression structure
- Example: t = T[(a - b) * (c + d)]

t1 = a
t2 = b
t3 = t1 – t2
} T[(a - b)]

t4 = b
t5 = c
t5 = t4 + t5
} T[(c + d)]

t = t3 * t5

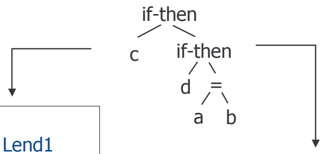
} T[(a - b) * (c + d)]

Nested Statements

- Same for statements: recursive translation
- Example: T[if c then if d then a = b]

t1 = c
fjump t1 Lend1
t2 = d
fjump t2 Lend2
t3 = b
a = t3
} T[a = b]
label Lend2
} T[if d …]
label Lend1
} T[if c then …]

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 = 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
  - Recursively generate code for children only if they are non-terminal expressions

---

No Duplicated Variables

- t = T[e1 OP e2]
  t1 = T[e1], if e1 is not terminal
  t2 = T[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 = T[(a+b)*c]
- Operand e1 = a+b, is not terminal
- Operand e2 = c, is terminal
- Translation: t1 = T[e1]
  t = t1 * c
- Recursively generate code for t1 = T[e1]
- For e1 = a+b, both operands are terminals
- Code for t1 = T[e1] is t1 = a+b
- Final result: t1 = a + b
  t = t1 * c

Accumulator Temporaries

- Use the same temporary variables for operands and result
- Translate t = T[e1 OP e2] as:

  t = T[e1]
  t1 = T[e2]
  t = t OP t1
- Example: t = T[(a+b)*c]
  t = a + b
  t = t * c

Reuse Temporaries

- Idea: in the translation of t = T[e1 OP e2] as:
  t = T[e1], t' = 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 = 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[e1 OP e2]
  t(c) = T[e1]
  ………………… c = c+1
  t(c) = T[e2]
  ………………… c = c-1
  t(c) = t(c) OP t(c+1)

Example

- t0 = T[((c*d) – (e*f)) + (a*b)]

```
            c = 0
t0 = T[ e0 ]
    t0 = c*d
                c = c+1
    t1 = e*f
                c = c-1
    t0 = t0 – t1
            c = c+1
t1 = a * b
            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 (more convenient for instruction selection)
- Possible compromise:
  - Reuse temporaries for intermediate values in each statement
  - Use different 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, L = T[e]
- No end label for a translation: L = ∅
- Translate t, L = T[e1 SC-OR e2] as:

```
t1, L1 = T[ e1 ]
tjump t1 L
t1, L2 = T[ e2 ]
```

- 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, L = T[(a < b) OR (c<d OR d<e)]

```
t = a < b
tjump t1 L
t, L' = T[ c<d OR d<e ]
    t = c < d
    fjump t L'
    t = d<e
    label Lend
```

- Backpatch t, L' = T[c<d OR d<e] : L' = Lend
- Backpatch t, L = T[(a<b) OR (...)] : L = L' = Lend

Encode Booleans in Control-Flow

- Consider T[if (a<b AND c<d) x = y;]

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

- ... can we do better?

Encode Booleans in Control-Flow

- Consider T[if (a<b AND c<d) x = y;]

```
t = a<b
fjump t L1
t = c<d
label L1
fjump t L2
x = y
label L2
```

```
t = a<b
fjump t L2
t = c<d
fjump t L2
x = y
label L2
```

} Condition and control flow

- 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:

  T[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: T[if(e) then s]

```
T[ e, L1, L2 ]
label L1
T[ s ]
label L2
```

- Also remove sequences ‘jump L, label L’

Define New Translations

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

```
T[ e, L1, L2 ]
label L1
T[ s1 ]
jump Lend
label L2
T[ s2 ]
label Lend
```

While Statement

- T[while (e) s]

```
label Ltest
T[ e, L1, L2 ]
label L1
T[ s ]
jump Ltest
label L2
```

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

---

Boolean Expression Translations

- T[true, L1, L2] :   jump L1
- T[false, L1, L2] :   jump L2
- T[e1 SC-OR e2, L1, L2]

```
T[ e1, L1, Lnext ]
label Lnext
T[ e2, L1, L2 ]
```

- T[e1 SC-AND e2, L1, L2]

```
T[ e1, Lnext, L2 ]
label Lnext
T[ e2, L1, L2 ]
```