CS412/CS413

Introduction to Compilers

Tim Teitelbaum

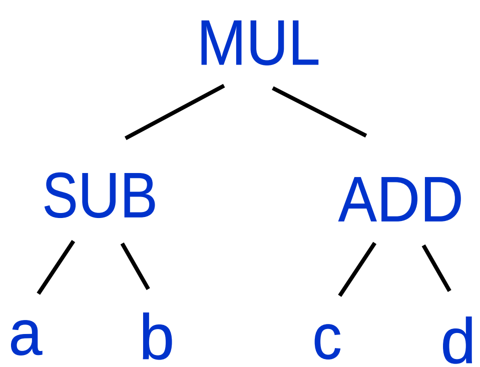
Lecture 19: Efficient IL Lowering

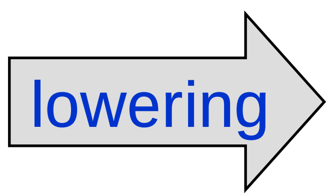
5 March 08

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

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), T[e] computes 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

Nested Expressions

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

- Example: t = T[(a - b) * (c + d)]

$$\left.\begin{array}{l}\left.\begin{array}{l} \mathrm{t1} = \mathrm{a} \\ \mathrm{t2} = \mathrm{b} \\ \mathrm{t3} = \mathrm{t1} - \mathrm{t2} \end{array}\right\} \mathrm{T}[\ (\mathrm{a} - \mathrm{b})\ ] \\ \left.\begin{array}{l} \mathrm{t4} = \mathrm{b} \\ \mathrm{t5} = \mathrm{c} \\ \mathrm{t5} = \mathrm{t4} + \mathrm{t5} \end{array}\right\} \mathrm{T}[\ (\mathrm{c} + \mathrm{d})\ ] \\ \mathrm{t} = \mathrm{t3} * \mathrm{t5} \end{array}\right\} \mathrm{T}[\ (\mathrm{a} - \mathrm{b}) * (\mathrm{c} + \mathrm{d})\ ]$$

Nested Statements

- Same for statements: recursive translation

- Example: T[if c then if d then a = b]

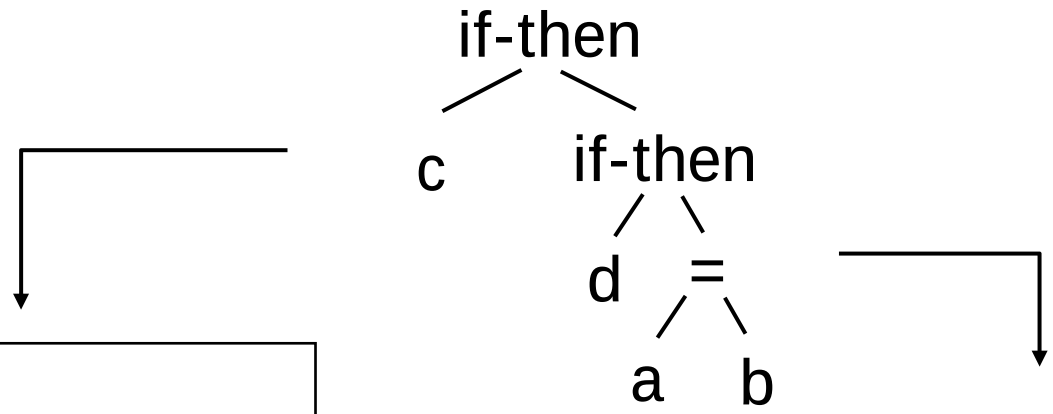
```
t1 = c
fjump t1 Lend1
t2 = d
fjump t2 Lend2
t3 = b
a = t3
label Lend2
label Lend1
```

T[a = b]

T[if d ...]

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
   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[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 = **if** e1 is terminal **then** e1 **else** t1

  x2 = **if** e2 is terminal **then** e2 **else** t2

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

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

Condition: t = a<b SC-AND c<d

Control flow: if (t) x = y

- ... can we do better?

Encode Booleans in Control-Flow

- Consider T[if (a<b SC-AND c<d) then 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

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[if(e) then s1 else s2]

  T[e, L, false]
  T[s1]
  jump Lend
  label L
  T[s2]
  label Lend

- T[if(e) then s]

  T[e, L, false]
  T[s]
  label L

While Statement

- T[while (e) s]

  label Ltest
  T[e, L, false]
  T[s]
  jump Ltest
  label L

SC-Boolean Expression Translations

- T[v, L, b] : if b then tjump v, L else fjump v, L
- T[!e, L, b] : T[e, L, !b]
- T[e1 SC-OR e2, L, true]

  T[e1, L, true]
  T[e2, L, true]
- T[e1 SC-AND e2, L, false]

  T[e1, L, false]
  T[e2, L, false]
- T[e1 SC-OR e2, L, false]

  T[e1, Lnext, true]
  T[e2, L, false]
  label Lnext
- T[e1 SC-AND e2, L, true]

  T[e1, Lnext, false]
  T[e2, L, true]
  label Lnext

Eliminate Jumps to Unconditional Jumps

- Example

  T[if a then if b then c=d else e=f else g=h]

```
fjump a L1
   fjump b L2
      c = d
   jump Lend2
   label L2
      e = f
   label Lend2
jump Lend1
label L1
   g = h
label Lend1
```

Eliminate Jumps to Unconditional Jumps

- Example

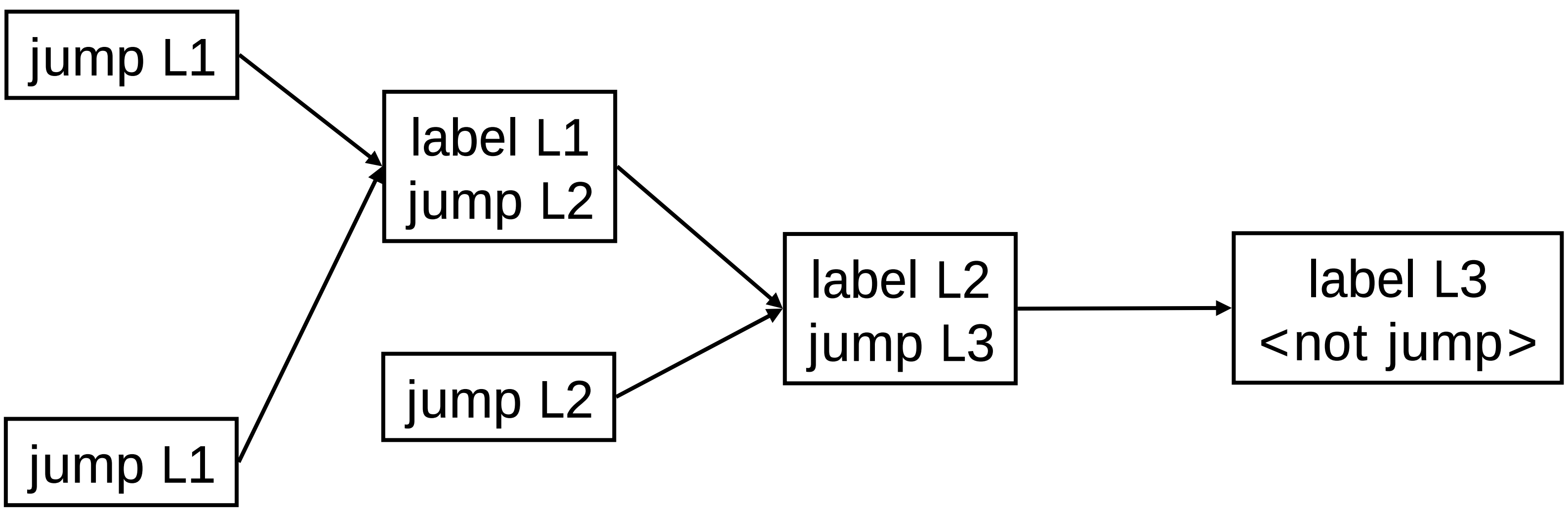
T[if a then if b then c=d else e=f else g=h]

```
fjump a L1
   fjump b L2
       c = d
   jump Lend1
   label L2
       e = f

 jump Lend1
label L1
   g = h
label Lend1
```

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