CS 412/413

Introduction to
Compilers and Translators
Cornell University
Andrew Myers

Lecture 14: Canonical Intermediate Code
23 Feb 00

Administration

- HW3 due Friday
- Prelim 1 in one week
  - covers topics up through this lecture
  - lexical, syntactic analysis
  - type checking and static semantics
  - syntax-directed translation and IR
- Prelim review Monday evening 7-9 PM
  - location TBA

Where we are

source code
parsing
abstract syntax tree (HIR)
syntax-directed translation
intermediate code (MIR)
syntax-directed translation
canonical intermediate code (MIR)
code generation
abstract assembly code (LIR)
register allocation
assembly code

What makes a good IR?

- Easy to translate from AST
- Easy to translate to assembly
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget

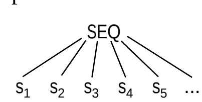
AST (>40 node types)
↓
IR (13 node types)
↓
Pentium (>200 opcodes)

Canonical form

- Intermediate code has general tree form
  - easy to generate from AST, but...
- Hard to translate directly to assembly
  - assembly code is a sequence of statements
  - Intermediate code (IR) has nodes corresponding to assembly statements deep in expression trees
- Canonical form: all statements brought up to top level of tree — generate assembly directly

One SEQ node

- In canonical form, only one SEQ node at the very top of tree



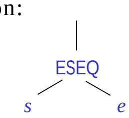
- Function is just a list of statements: SEQ(s_1,s_2,s_3,s_4,s_5,...)
- Can translate to assembly by translating each s_i to assembly statement(s) and concatenating

Canonical form

- Idea: rewrite IR to get rid of constructs incompatible with assembly code
  - arbitrarily deep expression trees — deal with this later as part of *instruction tiling*
  - ESEQ & CALL nodes — rewrite tree so no ESEQ nodes, CALLs moved to top
  - CJUMP is two-way jump rather than fall-through — convert to one-way jumps

no ESEQ nodes

- ESEQ nodes put a statement node underneath an expression:



S ⟦ x = a[(i = i+1)] ⟧ = ?

- Problem: statement can have arbitrary number of side effects; assembly can't
- Canonical form: no ESEQ nodes

  similar to: x = a[(i = i+1)] ⇒ i=i+1; x=a[i];

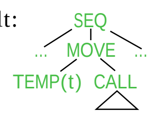
Top-level CALL statements

- CALL nodes have arbitrary side effects
- CALL node deep in expression tree will break translation to assembly

  Example: x = f(g(x) + h(y))

- Solution: move to top level
- Call that discards return result:

- Call that uses result:



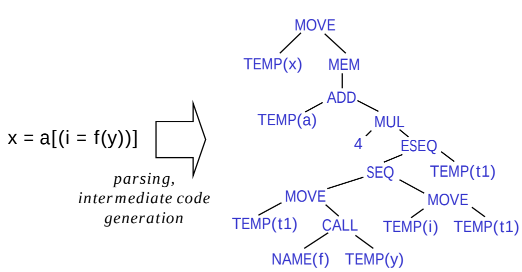
Canonical form

- Canonical tree has top-level SEQ node with following kinds of children:

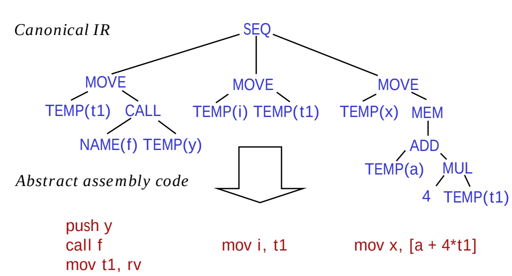
  MOVE(*dest*, *e*)
  MOVE(TEMP(*t*), CALL(...))
  EXP(CALL(...))
  JUMP(*e*)
  CJUMP(*e*, l_1, l_2)
  LABEL(*l*)

⇒Straightforward translation to assembly

Example



Canonical IR to assembly



ESEQ rewriting

- Want to move ESEQ nodes up to top of tree where they can become SEQ nodes
- Idea: define syntax-directed rules that take an IR tree and move ESEQ nodes to top.
- **Goal:** avoid ripping apart expressions more than necessary -- leads to better code because expression patterns can be recognized and mapped to instruction set

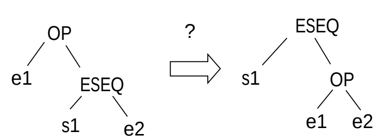
ESEQ rewrite rules

- Example transformations:

ESEQ(s1, ESEQ(s2, e)) $\Rightarrow$
ESEQ(SEQ(s1, s2), e))
MOVE(ESEQ(s1, e), dest) $\Rightarrow$
SEQ(s1, MOVE(e, dest))
OP(ESEQ(s1, e1), e2) $\Rightarrow$
ESEQ(s1, OP(e1, e2))
OP(e1, ESEQ(s1, e2)) $\Rightarrow$?

Rewriting expressions

- OP(e1, ESEQ(s1, e2))

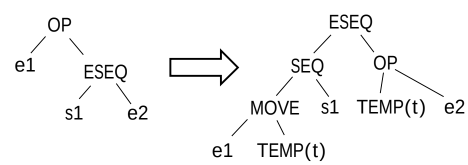


?
e1 + (a=0; e2) $\Rightarrow$ (a=0; e1 + e2)

Introducing temporaries

- If e1 does not *commute* with s1
  - i.e., (s1; e1; e2) $\neq$ (e1; s1; e2)

  must save value of e1 in temporary

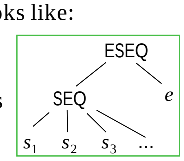


Implementation options

- Option 1: Walk over tree looking for places to apply rewrite rules (can use visitors)
  - "bad" nodes (ESEQ, CALL) percolate upward
  - Problem: need to restart tree traversal at every rewrite
- Option 2: Rewrite whole IR tree in one pass to build canonical IR tree
  - Syntax-directed translation!

General case

- When we move all ESEQ nodes to top, arbitrary expression node looks like:
- ESEQ transformation takes arbitrary expression node, returns list of sub-statements s_i to be executed (all side effects of e) plus final expression *e* **free of side effects.**



- Arbitrary statement node becomes a new SEQ node with no ESEQ nodes (or list of sub-statements s_i)

IR Simplification Interface

```
class CanonExpr {
   IRStmt[ ] pre_stmts;
   IRExpr expr;
}

abstract class IRExpr { CanonExpr simplify(); }
abstract class IRStmt { IRStmt[ ] simplify( ); }
```


Simplification

Two translation functions:

- $T[\![e]\!]$ gives a list of canonical statements s_i and a canonical expression e' where executing the s_i and **then** evaluating e has same side effects and value as e (= IRExpr.simplify)

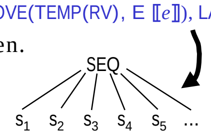
  $$T[\![e]\!] = (s_1, \ldots, s_n)\ ;\ e'$$

- $T[\![s]\!]$ gives a list of canonical statements s_i such that executing the s_i has same side effects as s (= IRStmt.simplify)

  $$T[\![s]\!] = (s_1, \ldots, s_n)$$

Simplifying a function body

- Last time: translate a function definition $f(a_1, \ldots, a_n) = e$ as SEQ(MOVE(TEMP(RV), $E[\![e]\!]$), LABEL(*epilogue*))
- Canonical form: SEQ node with all of $T[\![$ SEQ(MOVE(TEMP(RV), $E[\![e]\!]$), LABEL(*epilogue*)) $]\!]$ as children.



Rules

- Simplify arbitrary expression e:

  $$T[\![\ e\ ]\!] = (s_1, s_2, s_3, \ldots)\ ;\ e'$$

- Goal: define $T[\![\ e\ ]\!]$ and $T[\![\ s\ ]\!]$ for all 13 node types
- 3 trival cases:

  $$T[\![\ \mathrm{CONST}(i)\ ]\!] = (\ )\ ;\ \mathrm{CONST}(i)$$
  $$T[\![\ \mathrm{NAME}(n)\ ]\!] = (\ )\ ;\ \mathrm{NAME}(n)$$
  $$T[\![\ \mathrm{TEMP}(t)\ ]\!] = (\ )\ ;\ \mathrm{TEMP}(t)$$

- These expressions are already free of side effects: already in canonical form

JUMP, CJUMP, MEM

- JUMP(e), CJUMP(e, l_1, l_2), MEM(e)
- Need to make sure e is canonical
- $T[\![\ \mathrm{JUMP}(e)\ ]\!] = (s_1, \ldots, s_n, \mathrm{JUMP}(e'))$

  if $T[\![\ e\ ]\!] = (s_1, \ldots, s_{n,})\ ;\ e'$
- Similarly for CJUMP
- Can write as inference rule:

  $$\frac{T[\![\ e\ ]\!] = (s_1, \ldots, s_n)\ ;\ e'}{T[\![\ \mathrm{MEM}(e)\ ]\!] = (s_1, \ldots, s_n)\ ;\ \mathrm{MEM}(e')}$$

ESEQ

- How to simplify an expression ESEQ(s, e) ?

  $$\frac{T[\![\ e\ ]\!] = (s_1, \ldots, s_n)\ ;\ e'}{T[\![\ \mathrm{ESEQ}(s, e)\ ]\!] = (s, s_1, \ldots, s_n)\ ;\ e'}$$

- What is wrong with this rule?

Correct ESEQ rule

$$\frac{\mathcal{T}[\![e]\!] = (s_1, \ldots, s_n) \,;\, e' \qquad \mathcal{T}[\![s]\!] = (s'_1, \ldots, s'_m)}{\mathcal{T}[\![\mathrm{ESEQ}(s, e)]\!] = (s'_1, \ldots, s'_m, s_1, \ldots, s_n) \,;\, e'}$$

- Assuming $\mathcal{T}[\![e]\!]$, $\mathcal{T}[\![s]\!]$ produce canonical statements s_i, s'_j and canonical expression e, $\mathcal{T}[\![\mathrm{ESEQ}(s, e)]\!]$ works properly.

Translating binary operators

$$\frac{\mathcal{T}[\![e_1]\!] = (s_1, \ldots, s_m) \,;\, e'_1 \qquad \mathcal{T}[\![e_2]\!] = (s'_1, \ldots, s'_n) \,;\, e'_2}{\begin{array}{l}\mathcal{T}[\![\mathrm{OP}(e_1, e_2)]\!] = \\ (s_1, \ldots, s_m, \mathrm{MOVE}(\mathrm{TEMP}(t), e'_1), s'_1, \ldots, s'_n) \,;\, \mathrm{OP}(\mathrm{TEMP}(t), e'_2)\end{array}}$$

- Idea: save value of e_1 in a temporary before executing all the side effects of e_2

SEQ nodes

- How to get rid of SEQ nodes: concatenate canonical versions of all sub-statements

$$\frac{\mathcal{T}[\![s_1]\!] = (s'_1, \ldots, s'_m) \qquad \mathcal{T}[\![s_2]\!] = (s''_1, \ldots, s''_n)}{\mathcal{T}[\![\mathrm{SEQ}(s_1, s_2)]\!] = (s'_1, \ldots, s'_m, s''_1, \ldots, s''_n)}$$

CALL nodes

- CALL nodes call a function which may have side effects
- Overwrites return value register at least; can't be operand at assembly-code level
- Therefore, CALL nodes must move to top

$$\frac{\mathcal{T}[\![e_f]\!] = (s_1, \ldots, s_m) \,;\, e'_f \qquad \mathcal{T}[\![e_1]\!] = (s'_1, \ldots, s'_n) \,;\, e'_1}{\begin{array}{l}\mathcal{T}[\![\mathrm{CALL}(e_f, e_1)]\!] = \\ (s_1, \ldots, s_m, s'_1, \ldots, s'_n, \mathrm{MOVE}(\mathrm{TEMP}(t), \mathrm{CALL}(e'_f, e'_1)) ;\, \mathrm{TEMP}(t)\end{array}}$$

EXP

- EXP(e) evaluates e for its side effects, discards value
- Simplified IR does same:

$$\frac{\mathcal{T}[\![e]\!] = (s_1, \ldots, s_n) \,;\, e'}{\mathcal{T}[\![\mathrm{EXP}(e)]\!] = (s_1, \ldots, s_n)}$$

Canonical intermediate code

- Syntax-directed translation function $\mathcal{T}[\![\cdot]\!]$ simplifies an IR tree into canonical form
- Yields recursive impl. of IRStmt.simplify, IRExpr.simplify
- Canonical form: IR is a sequence of simple IR statements ready for translation to assembly