CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 15: Canonical IR
26 Feb 01

Administration

- HW3 due Friday
- Prelim 1 next Tuesday evening (7:30-9:30PM)
  - location TBA
  - covers topics up through this lecture
  - lexical, syntactic analysis
  - type checking and static semantics
  - syntax-directed translation and IR

Where we are

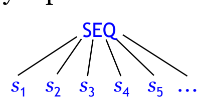
source code
parsing
abstract syntax tree (HIR)
syntax-directed translation
intermediate code (MIR)
syntax-directed translation
canonical intermediate code (MIR)
code generation
abstract assembly code (LIR)
register allocation
assembly code

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



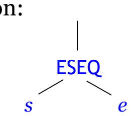
- Function body is just a list of statements: SEQ(s_1,s_2,s_3,s_4,s_5,...)
- Can translate to assembly by translating each s_i to assembly statement(s) and concatenating

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:

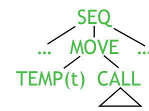


$\mathcal{S}[\![$ x = a[(i = i+1)] $]\!]$ = ?
Problem: statement can have arbitrary number of side effects; assembly can't
Canonical form: *no ESEQ nodes*
similar to: x = a[(i = i+1)] ⇒ i=i+1; x=a[i];

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

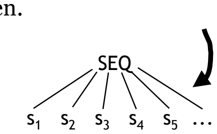
- Canonical tree has top-level SEQ node with following kinds of children:

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

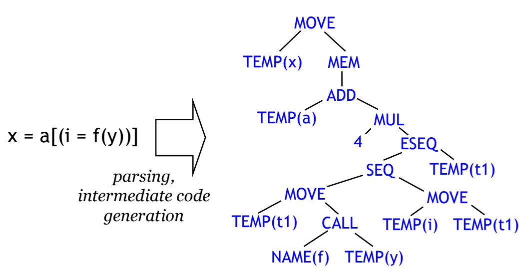
Code is a just sequence of these statements
⇒ Straightforward translation to assembly

Simplifying a function body

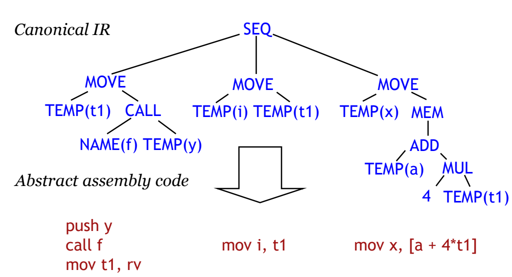
- Last time: translate a function definition $f(a_1, \ldots, a_n) = e$
  as SEQ(MOVE(TEMP(RV), $\mathcal{E}[\![e]\!]$), LABEL(*epilogue*))
- Canonical form: SEQ node with all of
  $\mathcal{T}[\![$ SEQ(MOVE(TEMP(RV), $\mathcal{E}[\![e]\!]$), LABEL(*epilogue*)) $]\!]$
  as children.



Example



Canonical MIR to LIR



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

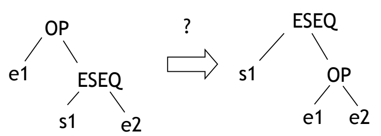
ESEQ rewrite rules

- Example transformations:

ESEQ(s1, ESEQ(s2, e)) ⇒
    ESEQ(SEQ(s1, s2), e))
MOVE(ESEQ(s1, e), dest) ⇒
    SEQ(s1, MOVE(e, dest))
OP(ESEQ(s1, e1), e2) ⇒
    ESEQ(s1, OP(e1, e2))
OP(e1, ESEQ(s1, e2)) ⇒ ?

Rewriting expressions

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



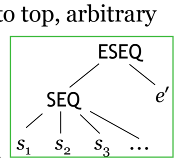
?
e1 + (a=0; e2) ⇒ (a=0; e1 + e2)

Implementation options

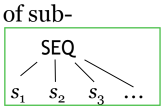
- Option 1: Walk over tree looking for places to apply rewrite rules
  - "bad" nodes (ESEQ, CALL) percolate upward, eventually disappear
  - Problem: may need to restart tree traversal at every rewrite
- Option 2: Rewrite whole IR tree in one pass to build canonical IR tree
  - Syntax-directed translation!

General case

- When we move all ESEQ nodes to top, arbitrary expression node *e* looks like:



- Transformation returns list of sub-statements s_i plus final expression e'
  - each s_i has at most one side-effect
  - e' is *free of side effects*.
- Arbitrary statement node becomes a new SEQ node with no ESEQ nodes (or list of sub-statements s_i)
  - each s_i has at most one side-effect



IR simplification Interface

```
class CanonicalExpr {
   IRStmt[ ] pre_stmts;
   IRExpr expr; }
}
class CanonicalStmt {
   IRStmt[ ] stmts;
}
abstract class IRExpr { CanonicalExpr simplify(); }
abstract class IRStmt { IRStmt[ ] simplify( ); }
```

Simplification

Two translation functions:

- $\mathcal{T}[\![e]\!]$ 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)

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

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

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

Rules

- Simplify arbitrary expression e:

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

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

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

- Already in canonical form!

JUMP, CJUMP, MEM

- JUMP(e), CJUMP(e, l_1, l_2), MEM(e)
  - Need to make sure e is canonical
- $\mathcal{T}[\![\text{JUMP}(e)]\!] = (s_1, \ldots, s_n, \text{JUMP}(e'))$
  if $\mathcal{T}[\![e]\!] = (s_1, \ldots, s_n) ; e'$
- Similarly for CJUMP
- Can write as inference rule:

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

ESEQ

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

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

?

Correct ESEQ rule

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

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

SEQ nodes

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

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

EXP

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

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

Translating binary operators

$$\frac{\mathcal{T}[\![e_1]\!] = (s_1,\ldots, s_m)\ ;\ e'_1 \qquad \mathcal{T}[\![e_2]\!] = (s'_1,\ldots, s'_n)\ ;\ e'_2}{\mathcal{T}[\![\ \mathsf{OP}(e_1, e_2)\ ]\!] = (s_1,\ldots, s_m, s'_1,\ldots,s'_n)\ ;\ \mathsf{OP}(e'_1, e'_2)}$$

- When does this rule work?
  - Note: OP allows either e_1 or e_2 to be evaluated first

Translating binary operators

- Previous rule works if e'_1 commutes with each of s'_i or e'_2 commutes with each of s'_i
- Idea: save value of e_1 in a temporary before executing all the side effects of e_2

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

- Works, but:
  - Introduces extra register t
  - No opportunity to do e'_1 and e'_2 in one instruction

Reordering

Statement s and expression e commute if executing s does not change result of e. False if:

- s overwrites any TEMP used by e
- s overwrites any MEM location that might be the same location as (alias) a MEM in e
  - conservative assumption*:* all memory locations may alias one another
  - less conservative: use *alias analysis* to determine which memory locations may alias

CALL nodes

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

$$\dfrac{\mathcal{T}[\![\ e_f\ ]\!] = (s_1,\ldots, s_m)\ ;\ e_f' \qquad \mathcal{T}[\![\ e_1\ ]\!] = (s_1',\ldots, s_n')\ ;\ e_1'}{\mathcal{T}[\![\ \mathsf{CALL}(e_f, e_1)\ ]\!] = (s_1,\ldots, s_m, s'_1,\ldots, s'_n, \mathsf{MOVE}(\mathsf{TEMP}(t), \mathsf{CALL}(e_f', e'_1));\ \mathsf{TEMP}(t)}$$

- Assumes e_f' commutes with $s'_1,\ldots, s'_n$
- Iota, C: operands may be reordered freely

Canonical intermediate code

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