Programming Languages

Building an Executable Program

Benjamin J. Keller

Department of Computer Science, Virginia Tech
Overview

- General Back-End Structure
- Intermediate Forms
- Code generation with attribute grammar
Compiler Back-End Structure

Front End

AST w/ annotations

Intermediate Code Generation

Flow graph w/ pseudo-instructions in basic blocks

Machine Independent Code Improvement

Modified flow graph

Target Code Generation

(Almost) assembly language

Machine-Specific Code Improvement

Real Assembly Language
Syntax Tree

```
program
  read
  read
  while
    <>
    if
      >
      :=
      :=
```

```
Index | Symbol     | Type
----- | ---------- | ----
1     | INTEGER   | type
2     | TEXTFILE  | type
3     | INPUT     | 2
4     | OUTPUT    | 2
5     | GCD       | program
6     | I         | 1
7     | J         | 1
```
Intermediate Code Generation

- Code improvements easier on form other than AST
- Basic block — maximal length sequence of nonbranch instructions
- Control Flow Graph
  - Vertices: basic blocks
  - Arcs: control flow between graphs
- Instructions for an idealized processor
- Assume unlimited set of virtual registers
Control Flow Graph

Start

a1 := &input
call readint
i := rv
a1 := &input
call readint
j := rv

v1 := i
v2 := j
v3 := v1 <-> v2
test v3

F

a1 := output
v13 := i
a2 := v13
call writeint
a1 := &output
call writeln

End

T

v4 := i
v5 := j
v6 := v4 > v5
test v6

T

v7 := i
v8 := j
v9 := v7 - v8
i := v9

F

v10 := j
v11 := i
v12 := v10 - v11
j := v12

null
Machine Independent Code Improvement

- Transformations on control flow graph
- Local improvements (to basic blocks):
  - Eliminate unnecessary loads and stores
  - Simplify and combine arithmetic expressions
- Global improvements:
  - Eliminating recomputations inside conditional blocks
  - Eliminating computation of invariant value inside loop
Final Phases

- Target code generation:
  - Translate control flow graph into assembly language for target machine
  - Strings together blocks, adding branches as needed

- Machine-specific code improvement:
  - Register allocation — assigning virtual registers to physical registers
  - Instruction scheduling — reordering instructions in basic block to try to keep pipeline(s) of machine full
Intermediate Forms

- Different compilers use different intermediate forms
- Vary in *level* or degree of abstraction from machine language
- High-level:
  - Tree-based forms
  - Stack-based: operands stored on stack
- Mid-level:
  - Control flow graph
  - Quadruples: opcode, two operands and destination
Example: GNU RTL

- Intermediate form for gcc
- Lisp like expressions: operator followed by list of operands
- Implemented using structs and pointers (text form not common)

```
(insn  8  6  10  (set  (reg:SI 2)
    (mem:SI (symbol_ref:SI ("a")))))
(insn  10  8  12  (set  (reg:SI 3)
    (mem:SI (symbol_ref:SI ("b")))))
(insn  12  10  14  (set  (reg:SI 2)
    (plus:SI (reg:SI 2)
      (reg:SI 3))))
(insn  14  12  15  (set  (reg:SI 3)
    (mem:SI (symbol_ref:SI ("c")))))
(insn  15  14  17  (set  (reg:SI 2)
    (mult:SI (reg:SI 2)
      (reg:SI 3))))
(insn  17  15  19  (set  (mem:SI (symbol_ref:SI ("d"))
      (reg:SI 2)))
```
A Simple Compiler

Front End → AST w/ annotations

- Naive Register Allocation

  Syntax Tree w/ more Annotations

  Target Code Generation → Assembly Language
Code Generation

\[
\begin{align*}
\text{program} & \rightarrow \ id \ stmt \\
& \quad \triangleright \ stmt.\text{next\_free\_reg} \ := \ 0 \\
& \quad \triangleright \ program.\text{code} \ := \ ["main:"] + \\
& \quad \quad \quad stmt.\text{code} + \ ["goto\ exit"] \\
& \quad \triangleright \ id.\text{stp} \rightarrow \text{name}
\end{align*}
\]
Code Generation

\[ if : stmt_1 \rightarrow expr \; stmt_2 \; stmt_3 \]

- expr.next_free_reg := stmt_2.next_free_reg
  := stmt_3.next_free_reg
  := stmt_4.next_free_reg
  := stmt_1.next_free_reg

- L1 := new_label(); L2 := new_label()

\[
stmt_1.code := expr.code + \\
["if" \; expr.reg \; "goto" \; L1] + \\
stmt_3.code + ["goto" \; L2] + [L1 ":"] + stmt_2.code \\
+ [L2 ":"] + stmt_4.code
\]
Code Generation

\[ id : expr \rightarrow \epsilon \]

- \[ expr.reg := \text{reg_names}[\text{expr.next.free.reg mod } k] \]
- \[ expr.code := [\text{expr.reg } "\text{:=}" \text{expr.stp->name}] \]
Register Allocation

- Maintain a pool of registers and cycle through them to select
- When return to parent, effectively pop registers off of “stack”
- If run out of registers, must *spill* contents to memory
- This is an extremely naive approach — would not be used
Example

Code for \((a+b) \cdot (c-(d/e))\)

\[
\begin{align*}
  r1 & := a \\
  r2 & := b \\
  r1 & := r1 + r2 \\
  r2 & := c \\
  r3 & := d \\
  r4 & := e \\
  r3 & := r3 / r4 \\
  r2 & := r2 - r3 \\
  r1 & := r1 \cdot r2
\end{align*}
\]