Intermediate Code Cleanup

• Tree intermediate code is fairly low-level
  – Compiler has already done a lot of the work toward generating assembly code
• However, some features are inconvenient when doing code generation
  – Two-way conditional jumps
  – Statements and calls (side-effects) within expressions
  – Nested function calls

Canonical Trees

• Appel: a piece of intermediate code is said to be canonical if it satisfies the following properties:
  – Does not contain SEQ or ESEQ.
  – CALL appears only directly inside EXP(...) or MOVE(TEMP t, ...).
• We can turn every Tree.stm into a sequence (list) of canonical statements.
• Similarly, every Tree.exp corresponds to a sequence of canonical statements followed by a canonical expression.

Example

ESEQ(EXP(CALL(NAME f, [CONST 1, BINOP(MINUS, CALL(NAME g, [CONST 2]), CONST 3)]),
  BINOP(PLUS,
    ESEQ(SEQ(MOVE(TEMP t, CONST 4), MOVE(TEMP u, CONST 5)),
       CONST 6),
    ESEQ(MOVE(TEMP w, CONST 7),
       CALL(NAME h, []))))
Canonical Version of Example

Canonicalization Process

• Give a name to every result of a CALL
  – Turn
    \[ \text{CALL}(f, a) \]
  into
    \[ \text{ESEQ}(\text{MOVE}(\text{TEMP} \ t, \ \text{CALL}(f, a)), \ \text{TEMP} \ t). \]
• Pull out all the statements
  – See Figure 8.1

Basic Blocks

• We now have a linear list of statements.
• Break these up into basic blocks
  – First statement is \text{LABEL}
  – Last statement is \text{JUMP} or \text{CJUMP}
  – No other \text{LABEL} or \text{JUMP}/\text{CJUMP}

Basic Block Generation

• Algorithm: Scan statements sequentially
  – Start new block at every \text{LABEL}
  – End block at every \text{JUMP} or \text{CJUMP}
  • Insert a fresh label next if needed
  – Add a \text{JUMP} to done at the end if needed
Ordering Basic Blocks

- Given definition of basic blocks, the blocks themselves can be arranged in any order.
- So, choose an order where every CJUMP is followed by its false branch.
  - Matches standard fall-through for conditional jump machine instructions.
  - Not possible in general
    - What if two CJUMP’s have the same false label?
    - Need to make small modifications to code
- Also, where possible, have JUMP followed by its destination.

Traces

- Definition: a trace is a sequence of statements that could be consecutively executed
  - Can include jumps
  - Convenient to consider traces as sequence of basic blocks
- We will divide the program into a collection of maximal, disjoint traces

Control Flow Graphs

- Visualize the collection of basic blocks as a graph
  - Each basic block is a node
  - Each node has one successor if it ends in JUMP, and (one or) two successors if it ends in CJUMP
- A trace is a directed path through this graph.

Trace Generation

- Simple algorithm:
  - Keep track of which nodes have been made part of a trace.
  - Follow a path through the graph, traversing only nodes that are not already part of a trace
    - Similar to DFS
    - Stop when you get stuck, make this path into a trace.
  - Repeat until all blocks are part of a trace
Trace Optimizations

• Some compilers are much pickier about the traces they generate
  – Try to generate traces likely to be consecutively executed at run-time
  – May duplicate code if two likely traces are non-disjoint
  – May optimize code on the likely path at the cost of slowing down other traces
    • Fix up instructions to undo code that was speculatively executed in the common trace.
• Particularly common for VLIW or highly superscalar machines.

Trace Cleanup

• The program is now a sequence of traces
  – List of (basic block lists)
  – Equivalent to a list of basic blocks
• Walk through all the blocks
  – If it ends in a CJUMP and is followed by the false label, do nothing
  – If it ends in a CJUMP and is followed by the true label, reverse the CJUMP
  – If it ends in a CJUMP and is followed by neither label, follow it with a new false block that jumps to the old false label.
  – If it ends in a JUMP followed by its destination, delete the jump.

Instruction Selection

• Now that intermediate code is in a nice form, how do we generate machine instructions?
• Idea: tree patterns.
  – See Figures 9.1 and 9.2

Optimal and Optimum Tilings

• An optimum tiling is a tiling with the lowest possible
• An optimal tiling is a one such that no two adjacent tiles can be combined into a single tile of lower total cost.
  – "locally optimum"
Maximal Munch

- Greedy Algorithm.
  - Always pick the largest possible tile at the root of the tree
  - Then recursively tile the remaining subtrees
- Always succeeds under reasonable assumptions
  - e.g., if there is a single-node tile for every sort of node
  - Results in tiling that is optimal but not the optimum.

Dynamic Programming

- Finds optimum tiling, not just optimal
- Idea:
  - Examine all possible tiles at the root
  - See what the resulting cost of the tree will be assuming optimum tiling of the remaining subtrees
  - Use dynamic programming (memoizing) to avoid recomputing the optimal tiling of a subtree.
- Two-pass algorithm:
  - First find optimum tiling of every subtree
  - Then emit the code for the optimum tiling of entire tree.

Dynamic Programming Issues

- Can be messy to program efficiently
  - Particularly for non-RISC machines
  - There exist automatic tools for generating code generators
  - Input looks like a context-free grammar
- Optimum only under unrealistic cost model
  - Assumes the cost of a tiling is the sum of the costs of the individual tiles.

Efficiency

- Assumptions
  - There are $T$ tiles (e.g., 50)
  - Average tile has $K$ nonleaf nodes (e.g., 2)
  - At most $K'$ nodes must be examined to see which patterns match the tree (e.g., 4)
  - On average, $T'$ patterns match each tree node (e.g., 5)
  - $N$ nodes in the input tree
- Maximal Munch: $(K'+T')N K$
  - Must match $N K'$ times, each time costing $K'+T'$
- Dynamic Programming: $(K'+T')N$
  - Must check matches at every node.
  - Bigger constant (walks the tree twice)
  - Usually not the bottleneck (faster even than lexing)
Handling CISC Machines

- Hard to generate good code for CISC machines
  - Few registers
  - Two-address instructions: add r1, r2
  - Registers not always interchangable
    - Address vs. data registers
    - Hard-coded source or destination: mul eax, reg
  - Fancy addressing modes, autoincrement

Appel's Suggestions

- Depend heavily on the register allocator
  - Few registers? Generate code assuming infinite number of registers, just as for RISC machines
  - Instruction-specific registers and/or two-address instructions? Insert extra moves.
  - tl ← t2 * t3 becomes
    - mov eax, t2 (eax ← t2)
    - mul t3 (eax ← eax * t3)
    - mov tl, eax (tl ← eax)
  - Hope that register allocator eliminates moves.
  - Ignore autoincrement, fancy addressing modes