Instruction Selection by Tree Matching

- Work on AST or on DAG for basic block
- Use patterns to describe semantics of machine instructions
- Cover tree with smallest/cheapest set of patterns
- Use a greedy strategy to find good covering, or
- Use dynamic programming to find optimal covering
- Generate patterns automatically from machine description files
Greedy strategy: the maximal munch approach

- Construct tree patterns for each machine instruction
- Cover (tile) the AST with matching tree patterns.
- Proceed top-down, and try larger patterns first, to minimize total number of instructions.
- Traverse tree top-down and emit instructions (in reverse order)
- Algorithm is fast but does not guarantee optimality
Instructions and their pattern trees

- **ADD** \( r_j = r_j + r_k \)

- **ADDI** \( r_i = r_i + c \)

- Express commutativity, and note that an addition can effect the load of a constant

- **LOAD** \( r_i = M[r_j + c] \) 4 trees.

- Assume frame pointer in register, and \( r_0 = 0 \) always.
Example: tiling for $a[i] = x$

```
move

mem

+  mem

  mem

    +

      mem

        *

          mem

            +

              fp

                x_{off}

                i (in register)

                Const 4

fp  a_{off}
```
Naïve tiling: 10 instructions

- **ADDI** \( r1 = r0 + a \)  // load offset of a
- **ADD** \( r1 = fp + r1 \)  // address of a
- **LOAD** \( r1 = M[r1 + 0] \)  // load a
- **ADDI** \( r2 = r0 + 4 \)  // load constant 4
- **MUL** \( r2 = ri * r2 \)  // scale i by 4
- **ADD** \( r1 = r1 + r2 \)  // base + index
- **ADDI** \( r2 = r0 + x \)  // load offset of x
- **ADD** \( r2 = fp + r2 \)  // address of x
- **LOAD** \( r2 = M [r2 + 0] \)  // load x
- **STORE** \( M [r1 + 0] = r2 \)  // store x in array
Better tiling: 3 memory accesses
6 instructions

- LOAD \( r_1 = M[fp+a] \)
- ADDI \( r_2 = r_0 + 4 \)
- MUL \( r_2 = r_i \times r_2 \)
- ADD \( r_1 = r_1 + r_2 \)
- LOAD \( r_2 = M[fp+x] \)
- STORE \( M[r_1 +0] = r_2 \)
Best tiling: two memory instructions

- **LOAD** \( r_1 = M [fp + a] \) // load a
- **ADDI** \( r_2 = r_0 + 4 \) // load 4
- **MUL** \( r_2 = r_i * r_2 \) // scale i
- **ADD** \( r_1 = r_1 + r_2 \) // address of a[i]
- **ADDI** \( r_2 = fp + x \) // address of x
- **MOVEM** \( M[r_1] = M[r_2] \) // memory copy
The maximal munch algorithm

- Top-down procedure.
- Order pattern trees by size, largest first
- Tree matches if operator matches and if descendants match (recursive)
- Different procedure for statements and for expressions.
Example: matching a move subtree

- **if** (op = "+")
  - && kind (right_operand) == Constant){
  - Munch (left_operand);
  - Emit (store_instruction);
- **else if** (op = "+")
  - && kind (left_operand) == Constant) { 
  - Munch (right_operand);
  - Emit (store_instruction);
- **else** ...

BURS
bottom-up rewriting System

- Describe machine instructions by tree patterns:
  
  \[
  \begin{align*}
  & \text{Rx} \\
  & \quad | \quad \text{Load}_\text{Const} \quad \text{cost} = 1 \quad | \quad \text{Load}_\text{Mem} \quad \text{Mem}, \text{Rx}, \text{cost} = 3 \\
  & \quad \text{cst} \quad \text{Mem} \\
  \\
  & \quad | \quad \text{Add}_\text{Mem} \quad \text{Mem},\text{Regx}, \text{cost} = 3 \\
  & \quad \text{Rx} \quad \text{Mem} \\
  \\
  & \quad | \quad \text{Mult}_\text{Mem} \quad \text{Mem},\text{Regx}, \text{cost} = 6 \\
  & \quad \text{Rx} \quad \text{Mem}
  \end{align*}
  \]
All operations have costs

```
Rx
+  
Rx       Ry
```

Add_Reg Rx, Ry cost = 2

```
Rx
+  
Rx       Ry
```

Mult_Mem Mem, Regx, cost = 3

```
Rx
+  
Rx       Ry
```

Rx

Three-pass algorithm

• Instruction-collecting: bottom-up tree matching, using cost information.
  – Each node has set of candidates
• Instruction selection: top-down pass to select single interpretation for node
• Code generation: bottom-up traversal to emit instructions in proper order
Tree Matching

• Similar approach to LR parsing:
  – Introduce **items**: patterns with dot, indicating partial match.
  – Label tree nodes with sets of items
  – Extend match by moving dot
  – Item with dot at root represents full match of instruction (**like accept state**)
Computing label sets for nodes

- Tree for $a \times b$:

  $\{ \text{load\_mem} \rightarrow R1 \}$

  $\{ \text{mem} \rightarrow \text{R1}, \text{mem} \rightarrow \text{R2} \}$

  $\{ \text{Mult\_mem} \rightarrow \text{R1, mem} \}$

  $\{ \text{Mult\_Reg} \rightarrow \text{R1, R2} \}$
Bottom-up pattern matching

To annotate node $N$ with a set of possible matches (labels):

- Match left operand
- Match right operand
- For each tree pattern $P$
  
  - For each label of left operand $L_1$
    
    - For each label of right operand $L_2$
      
      - If matches $(P, L_1, L_2)$ then
        
        - Compute cost of $(L_1, L_2)$ and add label to $N$
      
      - At each level, keep cheapest labelling (there maybe several of them)
Automating the process

- Describe semantics of instruction set in a machine description file
- Generate tree patterns automatically
- Generate automaton to guide pattern match
- Pattern-matching code is machine-independent, produces machine-specific code.
Example: some patterns for lcc on X86

- Describe a memory location:
  - mrc1: mem %0 1 // load, cost 1
  - mrc1: rc %0
- Describe addition with one operand in memory
  - reg:ADDI (reg, mrc1) (pattern on X86)
  - ?mov %c, %0, // may need a load
  - add %c, %1 1 // cost 1
References

- **Maximal munch:**
  - [Andrew Appel](#): modern compiler implementation in X (for various values of X)
- **BURS:**
  - [Brune, Bal, et al.](#): Modern Compiler Design
- **Original paper:**
  - [Proebsting](#): TOPLAS 17 (3) AMY 1995
- **LCC:**
  - [Fraser & Hanson](#): A retargetable C compiler