Lecture 19

Instruction Selection:
Tree-pattern matching

(EaC-11.3)
Many compilers use tree-structured IRs
• Abstract syntax trees generated in the parser
• Trees or DAGs for expressions
These systems might well use trees to represent target ISA

Consider the add operators

\[
\begin{align*}
\text{add } r_i, r_j & \Rightarrow r_k \\
\text{addI } r_i, c_j & \Rightarrow r_k
\end{align*}
\]

If we can match these “pattern trees” against IR trees, ...
The Concept

Low-level AST for \( w \leftarrow (*x) - 2 \times y \)

ARP: \( r_{\text{arp}} \)

NUM: constant

LAB: ASM label

w: at ARP+4
x: at ARP-26
y: at @G+12

(Ref ≈ Load)
The Concept

Low-level AST for \( w \leftarrow (\ast x) - 2 \ast y \)

ARP: \( r_{\text{arp}} \)
NUM: constant
LAB: ASM label

\( w \): at ARP+4
\( x \): at ARP-26
\( y \): at \(@G+12\)

(Ref \( \approx \) Load)
The Concept

Low-level AST for \( w \leftarrow (*x) - 2 \times y \)

Activation Record Pointer
(a.k.a. frame pointer)
Tree-pattern matching

Goal is to “tile” AST with operation trees

- A tiling is collection of \(<ast, op>\) pairs
  - \(ast\) is a node in the AST
  - \(op\) is an operation tree
  - \(<ast, op>\) means that \(op\) could implement the subtree at \(ast\)

- A tiling ‘implements” an AST if it covers every node in the AST and the overlap between any two trees is limited to a single node
  - \(<ast, op>\in\) tiling means \(ast\) is also covered by a leaf in another operation tree in the tiling, unless it is the root
  - Where two operation trees meet, they must be compatible (expect the value in the same location)
Tiling the Tree

Each tile corresponds to a sequence of operations

Emitting those operations in an appropriate order implements the tree.
Generating Code

Given a tiled tree

• Postorder treewalk, with node-dependent order for children
  → Right child of ← before its left child
  → Might impose “most demanding first” rule ...

• Emit code sequence for tiles, in order

• Tie boundaries together with register names
  → Tile 6 uses registers produced by tiles 1 & 5
  → Tile 6 emits “store \( r_{tile\ 5} \Rightarrow r_{tile\ 1} \)”
  → Can incorporate a “real” register allocator or just use virtual registers
So, What’s Hard About This?

Finding the matches to tile the tree
• Compiler writer connects operation trees to AST subtrees
  → Encode tree syntax, in linear form
  → Provides a set of rewrite rules
  → Associated with each is a code template
To describe these trees, we need a concise notation

\[
\begin{align*}
(r_i, r_j) + (c_j, r_i) \\
(r_i, c_j) + (r_i, r_j)
\end{align*}
\]

Linear prefix form
To describe these trees, we need a concise notation.
To describe these trees, we need a concise notation
### Rewrite rules: LL Integer AST into ILOC

<table>
<thead>
<tr>
<th>Rule</th>
<th>Cost</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Goal → Assign</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
| 2 Assign → ST(Reg₁,Reg₂) | 1 | store  
\[ r₂ \Rightarrow r₁ \] |
| 3 Assign → ST+(Reg₁,Reg₂),Reg₃) | 1 | storeAO  
\[ r₃ \Rightarrow r₁,r₂ \] |
| 4 Assign → ST+(Reg₁,NUM₂),Reg₃) | 1 | storeAI  
\[ r₃ \Rightarrow r₁,n₂ \] |
| 5 Assign → ST+(NUM₁,Reg₂),Reg₃) | 1 | storeAI  
\[ r₃ \Rightarrow r₂,n₁ \] |
| 6 Reg → LAB₁ | 1 | loadI  
\[ l₁ \Rightarrow r_{new} \] |
| 7 Reg → VAL₁ | 0 |  |
| 8 Reg → NUM₁ | 1 | loadI  
\[ n₁ \Rightarrow r_{new} \] |
| 9 Reg → REF(Reg₁) | 1 | load  
\[ r₁ \Rightarrow r_{new} \] |
| 10 Reg → REF+(Reg₁,Reg₂)) | 1 | loadAO  
\[ r₁,r₂ \Rightarrow r_{new} \] |
| 11 Reg → REF+(Reg₁,NUM₂)) | 1 | loadAI  
\[ r₁,n₂ \Rightarrow r_{new} \] |
| 12 Reg → REF+(NUM₁,Reg₂)) | 1 | loadAI  
\[ r₂,n₁ \Rightarrow r_{new} \] |
## Rewrite rules: LL Integer AST into ILOC (part II)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Cost</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 ( \text{Reg} \rightarrow \text{REF}(+ (\text{Reg}_1, \text{Lab}_2)) )</td>
<td>1</td>
<td>loadAI ( r_1, l_2 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>14 ( \text{Reg} \rightarrow \text{REF}(+ (\text{Lab}_1, \text{Reg}_2)) )</td>
<td>1</td>
<td>loadAI ( r_2, l_1 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>15 ( \text{Reg} \rightarrow + (\text{Reg}_1, \text{Reg}_2) )</td>
<td>1</td>
<td>addI ( r_1, r_2 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>16 ( \text{Reg} \rightarrow + (\text{Reg}_1, \text{NUM}_2) )</td>
<td>1</td>
<td>addI ( r_1, n_2 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>17 ( \text{Reg} \rightarrow + (\text{NUM}_1, \text{Reg}_2) )</td>
<td>1</td>
<td>addI ( r_2, n_1 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>18 ( \text{Reg} \rightarrow + (\text{Reg}_1, \text{Lab}_2) )</td>
<td>1</td>
<td>addI ( r_1, l_2 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>19 ( \text{Reg} \rightarrow + (\text{Lab}_1, \text{Reg}_2) )</td>
<td>1</td>
<td>addI ( r_2, l_1 \Rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>20 ( \text{Reg} \rightarrow - (\text{NUM}_1, \text{Reg}_2) )</td>
<td>1</td>
<td>rsubI ( r_2, n_1 \Rightarrow r_{\text{new}} )</td>
</tr>
</tbody>
</table>

... ... ... ...

A real set of rules would cover more than signed integers ...
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

What rules match tile 3?

```
REF
  └──+
    │   
    │   └──LAB
    │       @G
    │   └──NUM
    │       12
```
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

What rules match tile 3?

6: Reg → LAB₁ tiles the lower left node
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

What rules match tile 3?

6: $\text{Reg} \rightarrow \text{LAB}_1$ tiles the lower left node
8: $\text{Reg} \rightarrow \text{NUM}_1$ tiles the bottom right node
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

What rules match tile 3?
6: Reg → LAB₁ tiles the lower left node
8: Reg → NUM₁ tiles the bottom right node
15: Reg → + (Reg₁,Reg₂) tiles the + node
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

What rules match tile 3?

6: \( \text{Reg} \rightarrow \text{LAB}_1 \) tiles the lower left node
8: \( \text{Reg} \rightarrow \text{NUM}_1 \) tiles the bottom right node
15: \( \text{Reg} \rightarrow + (\text{Reg}_1, \text{Reg}_2) \) tiles the + node
9: \( \text{Reg} \rightarrow \text{REF}(\text{Reg}_1) \) tiles the REF
So, What’s Hard About This?

Need an algorithm to AST subtrees with the rules

Consider tile 3 in our example

What rules match tile 3?

6: $\text{Reg} \rightarrow \text{LAB}_1$ tiles the lower left node
8: $\text{Reg} \rightarrow \text{NUM}_1$ tiles the bottom right node
15: $\text{Reg} \rightarrow + (\text{Reg}_1, \text{Reg}_2)$ tiles the + node
9: $\text{Reg} \rightarrow \text{REF} (\text{Reg}_1)$ tiles the REF

We denote this match as $<6,8,15,9>$
Of course, it implies $<8,6,15,9>$
Both have a cost of 4
Finding matches

Many Sequences Match Our Subtree

<table>
<thead>
<tr>
<th>Cost</th>
<th>Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6,11</td>
</tr>
<tr>
<td>3</td>
<td>6,8,10</td>
</tr>
<tr>
<td>4</td>
<td>6,8,15,9</td>
</tr>
</tbody>
</table>

In general, we want the low cost sequence
• Each unit of cost is an operation (1 cycle)
• We should favour short sequences
Finding matches

Low Cost Matches

These two are equivalent in cost

6,11 might be better, because @G may be longer than the immediate field
Tiling the Tree

Still need an algorithm

• Assume each rule implements one operator
• Assume operator takes 0, 1, or 2 operands

Now, ...
Tiling the Tree

\[ \text{Tiling Tree(n)} \]

\[ \text{Label(n)} \leftarrow \emptyset \]

if \( n \) has two children then

\[ \text{Tile (left child of n)} \]

\[ \text{Tile (right child of n)} \]

for each rule \( r \) that implements \( n \)

\[ \text{if (left(r) \in Label(left(n)) and} \]

\[ \text{(right(r) \in Label(right(n))} \]

\[ \text{then Label(n) \leftarrow Label(n) \cup \{ r \}} \]

else if \( n \) has one child

\[ \text{Tile(child of n)} \]

for each rule \( r \) that implements \( n \)

\[ \text{if (left(r) \in Label(child(n))} \]

\[ \text{then Label(n) \leftarrow Label(n) \cup \{ r \}} \]

else /* \( n \) is a leaf */

\[ \text{Label(n) \leftarrow \{ all \ rules \ that \ implement \ n \}} \]

Notes:

- left and right refer to the children of the AST node or right-hand sides of a rule
- implements: e.g. rule 9 implements REF
Tiling the Tree

This algorithm
• Finds all matches in rule set
• Labels node \( n \) with that set
• Can keep lowest cost match at each point
• Leads to a notion of local optimality — lowest cost at each point
• Spends its time in the two matching loops

\[
\begin{align*}
\text{Tile}(n) & \\
\text{Label}(n) & \leftarrow \emptyset \\
\text{if } n \text{ has two children then} & \\
\text{Tile (left child of } n) & \\
\text{Tile (right child of } n) & \\
\text{for each rule } r \text{ that implements } n & \\
& \text{if } (\text{left}(r) \in \text{Label(left}(n))) \text{ and } \\
& \quad (\text{right}(r) \in \text{Label(right}(n))) \\
& \quad \text{then } \text{Label}(n) \leftarrow \text{Label}(n) \cup \{r\} \\
\text{else if } n \text{ has one child} & \\
\text{Tile(child of } n) & \\
\text{for each rule } r \text{ that implements } n & \\
& \text{if } (\text{left}(r) \in \text{Label(child}(n))) \\
& \quad \text{then } \text{Label}(n) \leftarrow \text{Label}(n) \cup \{r\} \\
\text{else /* } n \text{ is a leaf */} & \\
\text{Label}(n) & \leftarrow \{\text{all rules that implement } n\} 
\end{align*}
\]
The Big Picture

- Tree patterns represent AST and ASM
- Can use matching algorithms to find low-cost tiling of AST
- Can turn a tiling into code using templates for matched rules
- Techniques (& tools) exist to do this efficiently

| Hand-coded matcher like *Tile* | Avoids large sparse table  
|                               | Lots of work               |
| Encode matching as an automaton | O(1) cost per node  
|                               | Tools like BURS (bottom-up rewriting system), BURG |
| Use parsing techniques         | Uses known technology  
|                               | Very ambiguous grammars    |
| Linearize tree into string and use string searching algorithm (Aho-Corasick) | Finds all matches |