Lecture 14

Instruction Selection:
Tree-pattern matching

(EaC-11.3)
The Concept

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 \( w \leftarrow (\ast x) - 2 \ast y \)

ARP: \( r_{arp} \)
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)
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
  - \(<ast, op >\) ∈ 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)
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 \( \rightarrow \) 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 “\texttt{store } r_{\text{tile 5}} \Rightarrow r_{\text{tile 1}} ”
  - Can incorporate a “real” allocator or can use “NextRegister++”
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
Notation

To describe these trees, we need a concise notation

Linear prefix form

\[ + \]
\[ r_i \quad c_j \rightarrow +(r_i, c_j) \]

\[ + \]
\[ r_i \quad r_j \rightarrow +(r_i, r_j) \]
To describe these trees, we need a concise notation

\[
\begin{align*}
ST & \rightarrow + \\
    & \rightarrow VAL \\
    & \rightarrow ARP \\
    & \rightarrow NUM \\
    & \rightarrow 4 \\
    & \rightarrow REF \\
    & \rightarrow - \\
    & \rightarrow * \\
    & \rightarrow NUM \\
    & \rightarrow 2 \\
    & \rightarrow REF \\
    & \rightarrow + \\
    & \rightarrow VAL \\
    & \rightarrow ARP \\
    & \rightarrow NUM \\
    & \rightarrow -26 \\
    & \rightarrow REF \\
    & \rightarrow LAB \\
    & \rightarrow @G \\
    & \rightarrow NUM \\
    & \rightarrow 12
\end{align*}
\]
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 $\rightarrow$ Assign</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 Assign $\rightarrow$ ST(Reg$_1$,Reg$_2$)</td>
<td>1</td>
<td>store $r_2 \Rightarrow r_1$</td>
</tr>
<tr>
<td>3 Assign $\rightarrow$ ST(+(Reg$_1$,Reg$_2$),Reg$_3$)</td>
<td>1</td>
<td>storeAO $r_3 \Rightarrow r_1, r_2$</td>
</tr>
<tr>
<td>4 Assign $\rightarrow$ ST(+(Reg$_1$,NUM$_2$),Reg$_3$)</td>
<td>1</td>
<td>storeAI $r_3 \Rightarrow r_1, n_2$</td>
</tr>
<tr>
<td>5 Assign $\rightarrow$ ST(+(NUM$_1$,Reg$_2$),Reg$_3$)</td>
<td>1</td>
<td>storeAI $r_3 \Rightarrow r_2, n_1$</td>
</tr>
<tr>
<td>6 Reg $\rightarrow$ LAB$_1$</td>
<td>1</td>
<td>loadI $l_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>7 Reg $\rightarrow$ VAL$_1$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8 Reg $\rightarrow$ NUM$_1$</td>
<td>1</td>
<td>loadI $n_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>9 Reg $\rightarrow$ REF(Reg$_1$)</td>
<td>1</td>
<td>load $r_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>10 Reg $\rightarrow$ REF(+(Reg$_1$,Reg$_2$))</td>
<td>1</td>
<td>loadAO $r_1, r_2 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>11 Reg $\rightarrow$ REF(+(Reg$_1$,NUM$_2$))</td>
<td>1</td>
<td>loadAI $r_1, n_2 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>12 Reg $\rightarrow$ REF(+(NUM$_1$,Reg$_2$))</td>
<td>1</td>
<td>loadAI $r_2, n_1 \Rightarrow r_{new}$</td>
</tr>
</tbody>
</table>
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</td>
<td>1</td>
<td>loadAI $r_1,l_2 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>loadAI $r_2,l_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>addI $r_1,r_2 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>addI $r_1,n_2 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>addI $r_2,n_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>addI $r_1,l_2 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>addI $r_2,l_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>rsubI $r_2,n_1 \Rightarrow r_{new}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</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  NUM
  @G  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: \text{Reg} \rightarrow \text{LAB}_1 \text{ 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: Reg → LAB₁ tiles the lower left node
8: Reg → NUM₁ 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: \( \text{Reg} \rightarrow \text{LAB}_1 \) 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
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
9: Reg → REF(Reg₁) 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: Reg → LAB₁ tiles the lower left node
8: Reg → NUM₁ tiles the bottom right node
15: Reg → + (Reg₁, Reg₂) tiles the + node
9: Reg → REF(Reg₁) 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
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

\textit{Tile}(n)
\texttt{Label}(n) \leftarrow \emptyset
\textbf{if} \ n \ \textbf{has two children} \ \textbf{then}
\hspace{10pt} \textit{Tile} (\texttt{left child of} \ n)
\hspace{10pt} \textit{Tile} (\texttt{right child of} \ n)
\hspace{10pt} \textbf{for each rule} \ r \ \textbf{that implements} \ n
\hspace{20pt} \textbf{if} \ (\texttt{left}(r) \in \texttt{Label}(\texttt{left}(n))) \ \textbf{and}
\hspace{30pt} (\texttt{right}(r) \in \texttt{Label}(\texttt{right}(n)))
\hspace{20pt} \texttt{then} \ \texttt{Label}(n) \leftarrow \texttt{Label}(n) \cup \{ r \}
\textbf{else if} \ n \ \textbf{has one child}
\hspace{10pt} \textit{Tile}(\texttt{child of} \ n)
\hspace{10pt} \textbf{for each rule} \ r \ \textbf{that implements} \ n
\hspace{20pt} \textbf{if} \ (\texttt{left}(r) \in \texttt{Label}(\texttt{child}(n)))
\hspace{30pt} \texttt{then} \ \texttt{Label}(n) \leftarrow \texttt{Label}(n) \cup \{ r \}
\textbf{else} /* \ n \ \textbf{is a leaf} */
\hspace{10pt} \texttt{Label}(n) \leftarrow \{ \texttt{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

$\text{Tile}(n)$
$\text{Label}(n) \leftarrow \emptyset$
if $n$ has two children then
    $\text{Tile}$ (left child of $n$)
    $\text{Tile}$ (right child of $n$)
    for each rule $r$ that implements $n$
        if (left($r$) $\in$ Label(left($n$)) and
            right($r$) $\in$ Label(right($n$))
            then $\text{Label}(n) \leftarrow \text{Label}(n) \cup \{ r \}$
else if $n$ has one child
    $\text{Tile}$ (child of $n$)
    for each rule $r$ that implements $n$
        if (left($r$) $\in$ Label(child($n$))
            then $\text{Label}(n) \leftarrow \text{Label}(n) \cup \{ r \}$
else /* $n$ is a leaf */
    $\text{Label}(n) \leftarrow \{ \text{all rules that implement } n \}$
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand-coded matcher like <em>Tile</em></td>
<td>Avoids large sparse table, lots of work</td>
</tr>
<tr>
<td>Encode matching as an automaton</td>
<td>$O(1)$ cost per node, tools like BURS (bottom-up rewriting system), BURG</td>
</tr>
<tr>
<td>Use parsing techniques</td>
<td>Uses known technology, very ambiguous grammars</td>
</tr>
<tr>
<td>Linearize tree into string and use string searching algorithm (Aho-Corasick)</td>
<td>Finds all matches</td>
</tr>
</tbody>
</table>
Next Lecture

- Register Allocation