Compiler Design

Lecture 19:
Instruction Selection via Tree-pattern matching

Christophe Dubach
Winter 2022

(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

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

What if we could match these “pattern trees” against IR tree?
The Concept

AST for \( w \leftarrow (*x) - 2 \times y \)
The Concept

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

ARP: \$fp
NUM: constant
LAB: ASM label

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

ARP = Activation Record Pointer = frame pointer
The Concept

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

ARP = Activation Record Pointer = **frame pointer**
Tree-pattern matching

Goal is to “tile” AST with operation trees

• A tiling is collection of \(<ast, op \rangle\) pairs
  → \(ast\) is a node in the low-level AST
  → \(op\) is an operation tree
  → \(<ast, op \rangle\) 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
  → \(<ast, op \rangle \in \text{tiling}\) 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 \( \leftarrow \) 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 “\( \text{store } r_{\text{tile 5}} \Rightarrow r_{\text{tile 1}} \)”
  - 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
Notation

To describe these trees, we need a concise notation

\[ + \]
\[ r_i \quad c_j \]

\[ + \]
\[ r_i \quad r_j \]

\[ + (r_i, c_j) \]

\[ + (r_i, r_j) \]

Linear prefix form
To describe these trees, we need a concise notation
To describe these trees, we need a concise notation.
To describe these trees, we need a concise notation.
To describe these trees, we need a concise notation.
Notation

To describe these trees, we need a concise notation
To describe these trees, we need a concise notation

\[
ST(+(VAL_1, NUM_1), -(REF(REF(+(VAL_2, NUM_2))), *(NUM_3, (REF(+(LAB_1, NUM_3)))))
\]
## 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>
<tr>
<td>2 Assign → ST(Reg₁,Reg₂)</td>
<td>1</td>
<td>store r₂ → r₁</td>
</tr>
<tr>
<td>3 Assign → ST(+(Reg₁,Reg₂),Reg₃)</td>
<td>1</td>
<td>storeAO r₃ → r₁,r₂</td>
</tr>
<tr>
<td>4 Assign → ST(+(Reg₁,NUM₂),Reg₃)</td>
<td>1</td>
<td>storeAI r₃ → r₁,n₂</td>
</tr>
<tr>
<td>5 Assign → ST(+(NUM₁,Reg₂),Reg₃)</td>
<td>1</td>
<td>storeAI r₃ → r₂,n₁</td>
</tr>
<tr>
<td>6 Reg → LAB₁</td>
<td>1</td>
<td>loadI l₁ → rₙₑw</td>
</tr>
<tr>
<td>7 Reg → VAL₁</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8 Reg → NUM₁</td>
<td>1</td>
<td>loadI n₁ → rₙₑw</td>
</tr>
<tr>
<td>9 Reg → REF(Reg₁)</td>
<td>1</td>
<td>load r₁ → rₙₑw</td>
</tr>
<tr>
<td>10 Reg → REF(+(Reg₁,Reg₂))</td>
<td>1</td>
<td>loadAO r₁,r₂ → rₙₑw</td>
</tr>
<tr>
<td>11 Reg → REF(+(Reg₁,NUM₂))</td>
<td>1</td>
<td>loadAI r₁,n₂ → rₙₑw</td>
</tr>
<tr>
<td>12 Reg → REF(+(NUM₁,Reg₂))</td>
<td>1</td>
<td>loadAI r₂,n₁ → rₙₑw</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 $r_1,l_2 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>loadAI $r_2,l_1 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>addI $r_1,r_2 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>addI $r_1,n_2 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>addI $r_2,n_1 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>addI $r_1,l_2 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>addI $r_2,l_1 \Rightarrow r_{\text{new}}$</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>rsupI $r_2,n_1 \Rightarrow r_{\text{new}}$</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: Reg \rightarrow 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: 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: 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></td>
<td>8,14</td>
</tr>
<tr>
<td>3</td>
<td>6,8,10</td>
</tr>
<tr>
<td></td>
<td>8,6,10</td>
</tr>
<tr>
<td></td>
<td>6,16,9</td>
</tr>
<tr>
<td></td>
<td>8,19,9</td>
</tr>
<tr>
<td>4</td>
<td>6,8,15,9</td>
</tr>
<tr>
<td></td>
<td>8,6,15,9</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

Sequences with Cost of 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6: Reg → LAB₁</td>
<td>loadI @G ⇒ r_i</td>
</tr>
<tr>
<td>11: Reg → REF(+(Reg₁,NUM₂))</td>
<td>loadAI r_i,12 ⇒ r_j</td>
</tr>
<tr>
<td>8: Reg → NUM₁</td>
<td>loadI 12 ⇒ r_i</td>
</tr>
<tr>
<td>14: Reg → REF(+(LAB₁,Reg₂))</td>
<td>loadAI r_i,@G ⇒ r_j</td>
</tr>
</tbody>
</table>

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)

\hspace{1em} \textit{Label}(n) \leftarrow \emptyset

\hspace{1em} \textbf{if} \ n \ \textbf{has two children} \ \textbf{then}

\hspace{2em} \textit{Tile} \ (\text{left child of } n)

\hspace{2em} \textit{Tile} \ (\text{right child of } n)

\hspace{1em} \textbf{for each rule } r \ \textbf{that implements } n

\hspace{2em} \textbf{if} \ (\text{left}(r) \in \text{Label}(\text{left}(n))) \ \textbf{and}

\hspace{3em} (\text{right}(r) \in \text{Label}(\text{right}(n)))

\hspace{2em} \textbf{then} \ \textit{Label}(n) \leftarrow \text{Label}(n) \cup \{ r \}

\hspace{1em} \textbf{else if} \ n \ \textbf{has one child}

\hspace{2em} \textit{Tile}(\text{child of } n)

\hspace{1em} \textbf{for each rule } r \ \textbf{that implements } n

\hspace{2em} \textbf{if} \ (\text{left}(r) \in \text{Label}(\text{child}(n)))

\hspace{3em} \textbf{then} \ \textit{Label}(n) \leftarrow \text{Label}(n) \cup \{ r \}

\hspace{1em} \textbf{else} \ /* \ n \ is a leaf */

\hspace{2em} \textit{Label}(n) \leftarrow \{ \text{all rules that implement } n \}
Tiling the Tree

Tile(n)
Label(n) ← Ø
if n has two children then
  Tile (left child of n)
  Tile (right child of n)
  for each rule r that implements n
    if (left(r) ∈ Label(left(n)) and
        (right(r) ∈ Label(right(n)))
    then Label(n) ← Label(n) ∪ { r }
else if n has one child
  Tile(child of n)
  for each rule r that implements n
    if (left(r) ∈ Label(child(n))
    then Label(n) ← Label(n) ∪ { r }
else /* n is a leaf */
  Label(n) ← { all rules that implement n }

Label(Ref) =
Label(+) =
Label(Lab) =
Label(Num) =
Tiling the Tree

Tile(n)
Label(n) ← Ø
if n has two children then
  Tile (left child of n)
  Tile (right child of n)
  for each rule r that implements n
    if (left(r) ∈ Label(left(n)) and
    (right(r) ∈ Label(right(n))
    then Label(n) ← Label(n) ∪ { r }
else if n has one child
  Tile(child of n)
  for each rule r that implements n
    if (left(r) ∈ Label(child(n))
    then Label(n) ← Label(n) ∪ { r }
else /* n is a leaf */
  Label(n) ← {all rules that implement n }

This algorithm

• Finds all matches in rule set
• Labels node n with that set
• Can keep lowest cost match at each point for each type of nodes
→ Dynamic programming
• Spends its time in the two matching loops
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 |
Next Lecture

- Object Oriented Programming Support