Compiler Design

Lecture 19:
Instruction Selection via Tree-pattern matching

Christophe Dubach
Winter 2023

(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

\[
\text{add } r_i, r_j \Rightarrow r_k
\]

\[
\text{addI } r_i, j \Rightarrow r_k
\]

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

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

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

ARP = Activation Record Pointer = **frame pointer**
The Concept

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

- **ARP**: $fp
- **NUM**: constant
- **LAB**: ASM label

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

**Abbreviations**
- VAL: Activation Record Pointer = **frame pointer**
- ARP: Activation Record Pointer
- NUM: constant
- LAB: ASM label
- ST: Store
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 low-level 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 \( \leftarrow \) 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

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

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

Linear prefix form
Notation

To describe these trees, we need a concise notation

```
+ VAL ARP + NUM 4 - REF + NUM -26 - REF + LAB @G + NUM 12
```
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

\[
\text{ST}(+(\text{VAL}_1, \text{NUM}_1), -\text{REF}(\text{REF}(+(\text{VAL}_2, \text{NUM}_2))), *(\text{NUM}_3, (\text{REF}(+(\text{LAB}_1, \text{NUM}_3))))))
\]
## 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  ( r_2 \rightarrow r_1 )</td>
</tr>
<tr>
<td>3 Assign → ST(+ (Reg₁,Reg₂),Reg₃)</td>
<td>1</td>
<td>storeAO  ( r_3 \rightarrow r_1, r_2 )</td>
</tr>
<tr>
<td>4 Assign → ST(+ (Reg₁,NUM₂),Reg₃)</td>
<td>1</td>
<td>storeAI  ( r_3 \rightarrow r_1, n_2 )</td>
</tr>
<tr>
<td>5 Assign → ST(+ (NUM₁,Reg₂),Reg₃)</td>
<td>1</td>
<td>storeAI  ( r_3 \rightarrow r_2, n_1 )</td>
</tr>
<tr>
<td>6 Reg → LAB₁</td>
<td>1</td>
<td>loadI  ( l_1 \rightarrow r_{\text{new}} )</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  ( n_1 \rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>9 Reg → REF(Reg₁)</td>
<td>1</td>
<td>load  ( r_1 \rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>10 Reg → REF(+ (Reg₁,Reg₂))</td>
<td>1</td>
<td>loadAO  ( r_1, r_2 \rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>11 Reg → REF(+ (Reg₁,NUM₂))</td>
<td>1</td>
<td>loadAI  ( r_1, n_2 \rightarrow r_{\text{new}} )</td>
</tr>
<tr>
<td>12 Reg → REF(+ (NUM₁,Reg₂))</td>
<td>1</td>
<td>loadAI  ( r_2, n_1 \rightarrow r_{\text{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 Reg → REF(+ (Reg&lt;sub&gt;1&lt;/sub&gt;,Lab&lt;sub&gt;2&lt;/sub&gt;))</td>
<td>1</td>
<td>loadAI r&lt;sub&gt;1&lt;/sub&gt;,l&lt;sub&gt;2&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>14 Reg → REF(+ (Lab&lt;sub&gt;1&lt;/sub&gt;,Reg&lt;sub&gt;2&lt;/sub&gt;))</td>
<td>1</td>
<td>loadAI r&lt;sub&gt;2&lt;/sub&gt;,l&lt;sub&gt;1&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>15 Reg → + (Reg&lt;sub&gt;1&lt;/sub&gt;,Reg&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>1</td>
<td>addI r&lt;sub&gt;1&lt;/sub&gt;,r&lt;sub&gt;2&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>16 Reg → + (Reg&lt;sub&gt;1&lt;/sub&gt;,NUM&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>1</td>
<td>addI r&lt;sub&gt;1&lt;/sub&gt;,n&lt;sub&gt;2&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>17 Reg → + (NUM&lt;sub&gt;1&lt;/sub&gt;,Reg&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>1</td>
<td>addI r&lt;sub&gt;2&lt;/sub&gt;,n&lt;sub&gt;1&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>18 Reg → + (Reg&lt;sub&gt;1&lt;/sub&gt;,Lab&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>1</td>
<td>addI r&lt;sub&gt;1&lt;/sub&gt;,l&lt;sub&gt;2&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>19 Reg → + (Lab&lt;sub&gt;1&lt;/sub&gt;,Reg&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>1</td>
<td>addI r&lt;sub&gt;2&lt;/sub&gt;,l&lt;sub&gt;1&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</td>
</tr>
<tr>
<td>20 Reg → - (NUM&lt;sub&gt;1&lt;/sub&gt;,Reg&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>1</td>
<td>rsupI r&lt;sub&gt;2&lt;/sub&gt;,n&lt;sub&gt;1&lt;/sub&gt; ⇒ r&lt;sub&gt;new&lt;/sub&gt;</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: $\text{Reg} \rightarrow \text{LAB}_1$ 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: $\text{Reg} \rightarrow \text{LAB}_1$ 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: Reg $\rightarrow$ LAB$_1$ tiles the lower left node
8: Reg $\rightarrow$ NUM$_1$ tiles the bottom right node
15: Reg $\rightarrow$ + (Reg$_1$,Reg$_2$) tiles the + node
9: Reg $\rightarrow$ REF(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></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

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

Tile(n)

Label(n) ← ∅

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 }

Notes:
- left and right refer to the children of the AST node or left/right-hand sides of a rule
- implements: e.g. rule 9 implements REF
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 \}

<table>
<thead>
<tr>
<th>Rule</th>
<th>$</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Assign</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Assign → ST(Reg₁,Reg₂)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Assign → ST(+(Reg₁,Reg₂),Reg₃)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Assign → ST(+(Reg₁,NUM₂),Reg₃)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Assign → ST(+(NUM₁,Reg₂),Reg₃)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Reg → LAB₁</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Reg → VAL₁</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Reg → NUM₁</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Reg → REF(Reg₁)</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Reg → REF(+(Reg₁,Reg₂))</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Reg → REF(+(Reg₁,NUM₂))</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Reg → REF(+(NUM₁,Reg₂))</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>Reg → REF(+(Reg₁,Lab₂))</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>Reg → REF(+(Lab₁,Reg₂))</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>Reg → +(Reg₁,Reg₂)</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>Reg → +(Reg₁,NUM₂)</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>Reg → +(NUM₁,Reg₂)</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>Reg → +(Reg₁,Lab₂)</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>Reg → +(Lab₁,Reg₂)</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>Reg → -(NUM₁,Reg₂)</td>
<td>1</td>
</tr>
</tbody>
</table>

Label(Ref) = 
Label(+) = 
Label(Lab) = 
Label(Num) =
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 for each type of nodes → Dynamic programming

• Spends its time in the two matching loops

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

• Object Oriented Programming Support