Instruction Selection: Tree-pattern matching

EaC-11.3

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
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 ILOC add operators

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

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

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

ARP: \( r_{arp} \)

NUM: constant

LAB: ASM label

w: at ARP+4
x: at ARP-26
Y: at @G+12
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
To describe these trees, we need a concise notation.

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

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

\[ +(r_i, c) \]

\[ +(r_i, r_j) \]

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

\[
\text{GETS}(+(\text{VAL}_1,\text{NUM}_1), -(\text{REF}(\text{REF}(+(\text{VAL}_2,\text{NUM}_2))), *(\text{NUM}_3,(\text{REF}(+(\text{LAB}_1,\text{NUM}_3))))))
\]
Tree-pattern matching

Goal is to “tile” AST with operation trees

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

• 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
  \( \rightarrow <\text{ast}, \text{op}> \in \text{tiling} \) means \( \text{ast} \) is also covered by a leaf in another operation tree in the tiling, unless it is the root
  \( \rightarrow \) 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 GETS before its left child
  - Might impose “most demanding first” rule ...  
    \[ \text{(Sethi)} \]

- 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” 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
  - Provides a set of rewrite rules
  - Encode tree syntax, in linear form
  - Associated with each is a code template
## 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</td>
<td>Goal → Assign</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Assign → GETS(Reg₁,Reg₂)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Assign → GETS(+ (Reg₁,Reg₂),Reg₃)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Assign → GETS(+ (Reg₁,NUM₂),Reg₃)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Assign → GETS(+ (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>
</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>add</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>addI</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>addI</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>sub</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>subI</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>rsubI</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>mult</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>multI</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>multI</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 $\rightarrow$ LAB$_1$ tiles the lower left node
8: Reg $\rightarrow$ 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: \( \text{Reg} \rightarrow \text{LAB}_1 \) tiles the lower left node
8: \( \text{Reg} \rightarrow \text{NUM}_1 \) tiles the bottom right node
13: \( \text{Reg} \rightarrow + (\text{Reg}_1, \text{Reg}_2) \) 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
13: \( \text{Reg} \rightarrow + (\text{Reg}_1, \text{Reg}_2) \) 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 \text{LAB}_1 \text{ tiles the lower left node}
8: Reg \rightarrow \text{NUM}_1 \text{ tiles the bottom right node}
13: Reg \rightarrow + (\text{Reg}_1,\text{Reg}_2) \text{ tiles the + node}
9: Reg \rightarrow \text{REF}(\text{Reg}_1) \text{ tiles the REF}

We denote this match as \langle 6, 8, 13, 9 \rangle
Of course, it implies \langle 8, 6, 13, 9 \rangle
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 8,12</td>
</tr>
<tr>
<td>3</td>
<td>6,8,10 8,6,10 6,14,9 8,15,9</td>
</tr>
<tr>
<td>4</td>
<td>6,8,13,9 8,6,13,9</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

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

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

\[ \text{if } n \text{ has two children then} \]

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

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

\[ \text{for each rule } r \text{ that implements } n \]

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

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

\[ \text{else if } n \text{ has one child} \]

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

\[ \text{for each rule } r \text{ that implements } n \]

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

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

\[ \text{else /* } n \text{ is a leaf */} \]

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

**Match binary nodes against binary rules**

**Match unary nodes against unary rules**

**Handle leaves with lookup in rule table**
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

```
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 }
```
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 }

Oversimplifications
1. Only handles 1 storage class
2. Must track low cost sequence in each class
3. Must choose lowest cost for subtree, across all classes

The extensions to handle these complications are pretty straightforward.
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 }

Can turn matching code (inner loop) into a table lookup
Table can get huge and sparse
|op trees| x |labels| x |labels|
  200    x 1000 x 1000
leads to 200,000,000 entries
Fortunately, they are quite sparse & have reasonable encodings (e.g., Chase’s work)
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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lots of work</td>
</tr>
<tr>
<td>Encode matching as an automaton</td>
<td>O(1) cost per node</td>
</tr>
<tr>
<td></td>
<td>Tools like BURS (bottom-up rewriting system), BURG</td>
</tr>
<tr>
<td>Use parsing techniques</td>
<td>Uses known technology</td>
</tr>
<tr>
<td></td>
<td>Very ambiguous grammars</td>
</tr>
<tr>
<td>Linearize tree into string and use Aho-Corasick</td>
<td>Finds all matches</td>
</tr>
</tbody>
</table>
Extra Slides Start Here
Other Sequences

Two operator rule

6, 11
6: Reg → LAB₁
11: Reg → REF( + (Reg₁, NUM₂))
Other Sequences

8,12

8: Reg → NUM\(_1\)
12: Reg → REF( + (NUM\(_1\),Reg\(_2\)))

Two operator rule
Other Sequences

6,8,10

6:  Reg $\rightarrow$ LAB$_1$
8:  Reg $\rightarrow$ NUM$_1$
11: Reg $\rightarrow$ REF( + (Reg$_1$,Reg$_2$))

8,6,10 looks the same
Other Sequences

6, 14, 9

6: Reg → LAB₁
14: Reg → + (Reg₁, NUM₂)
9: Reg → REF(Reg₁)

All single operator rules
Other Sequences

8, 15, 9

8: \( \text{Reg} \rightarrow \text{NUM}_1 \)
15: \( \text{Reg} \rightarrow + (\text{NUM}_1, \text{Reg}_2) \)
9: \( \text{Reg} \rightarrow \text{REF} (\text{Reg}_1) \)

All single operator rules
Other Sequences

6,8,13,9

6: Reg → LAB₁
8: Reg → NUM₁
13: Reg → + (Reg₁, Reg₂)
9: Reg → REF(Reg₁)

8,6,13,9 looks the same