Lecture 7

More Examples of Data Flow Analysis:
Global Common Subexpression Elimination:
Constant Propagation/Folding

I. Available Expressions Analysis
II. Eliminating CSEs
III. Constant Propagation/Folding

Reading: 9.2.6, 9.4

Formulating the Problem

• Domain:
  • a bit vector, with a bit for each textually unique expression in the program
  • Forward or Backward?
  • Lattice Elements?
  • Meet Operator?
    • check commutative, idempotent, associative
  • Partial Ordering
• Top?
• Bottom?
• Boundary condition: entry/exit node?
• Initialization for iterative algorithm?

Global Common Subexpressions

add t1 = x, y
add t2 = c, d

ldc t3 = 0
cpy x = t3
add t4 = x, y
cpy m = t4

sub t5 = a, b
ldc t6 = -1
cpy c = t6

sub t7 = a, b
cpy m = t7
add t8 = x, y
add t9 = c, d

• Availability of an expression E at point P
  • DEFINITION: Along every path to P in the flow graph:
    – E must be evaluated at least once
    – no variables in E redefined after the last evaluation
  • Observations: E may have different values on different paths

Transfer Functions

• Can use the same equation as reaching definitions
  • out[b] = gen[b] \cup (in[b] - kill[b])
• Start with the transfer function for a single instruction
  • When does the instruction generate an expression?
  • When does it kill an expression?
• Calculate transfer functions for complete basic blocks
  • Compose individual instruction transfer functions
Composing Transfer Functions

- Derive the transfer function for an entire block
  \[ \text{in1} \]
  \[ \downarrow \]
  \[ \text{out1} = \text{gen1} \cup (\text{in1} \setminus \text{kill1}) = \text{in2} \]
  \[ \downarrow \]
  \[ \text{out2} = \text{gen2} \cup (\text{in2} \setminus \text{kill2}) \]

- Since \( \text{out1} = \text{in2} \) we can simplify:
  - \( \text{out2} = \text{gen2} \cup (\text{gen1} \cup (\text{in1} \setminus \text{kill1}) \setminus \text{kill2}) \)
  - \( \text{out2} = \text{gen2} \cup (\text{gen1} \setminus \text{kill2}) \cup (\text{in1} \setminus (\text{kill1} \cup \text{kill2})) \)

- Result
  - \( \text{gen} = \text{gen2} \cup (\text{gen1} \setminus \text{kill2}) \)
  - \( \text{kill} = \text{kill2} \cup (\text{kill1} \setminus \text{gen2}) \)

Example Revisited

- \( \text{add } t_1 = x, y \)
- \( \text{add } t_2 = c, d \)
- \( \text{ldc } t_3 = 0 \)
- \( \text{cpy } x = t_3 \)
- \( \text{add } t_4 = x, y \)
- \( \text{cpy } m = t_4 \)
- \( \text{sub } t_5 = a, b \)
- \( \text{ldc } t_6 = -1 \)
- \( \text{cpy } c = t_6 \)
- \( \text{sub } t_7 = a, b \)
- \( \text{cpy } m = t_7 \)
- \( \text{add } t_8 = x, y \)
- \( \text{add } t_9 = c, d \)

II. Eliminating CSEs

- Available expressions (across basic blocks)
  - provides the set of expressions available at the start of a block

- Value Numbering (within basic block)
  - Initialize Values table with available expressions

- If CSE is an "available expression", then transform the code
  - Original destination may be:
    - a temporary register
    - overwritten
    - different from the variables on other paths
  - One solution: Copy the expression to a new variable at each evaluation reaching the redundant use

III. Limitation: Textually Identical Expressions

- Commutative operations
  - \( \text{add } t_1 = x, y \)
  - \( \text{add } t_2 = y, x \)
  - sort the operands
  - \( \text{add } t_3 = x, y \)
Further Improvements

• Examples
  – Expressions with more than two operands
    
    | add t1 = x, y |
    | add t2 = t1, z |
    | add t3 = y, x |
    | add t4 = t3, z |
    | add t5 = x, y |
    | add t6 = t5, z |
  – Textually different expressions may be equivalent
    
    | add t1 = x, y |
    | beq t1, t2, t1 |
    | cpy z = x |
    | add t3 = z, y |

Another Example

x = 1
y = 1

x = x + 1
y = y + 1
x = x + 1
y = y + 1

Summary

<table>
<thead>
<tr>
<th>Domain</th>
<th>Reaching Definitions</th>
<th>Available Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets of definitions</td>
<td>Sets of expressions</td>
</tr>
<tr>
<td>Transfer function</td>
<td>f_b(x)</td>
<td>Generate U Propagate.</td>
</tr>
<tr>
<td>direction of function</td>
<td>forward: out[b] = f_b(in[b])</td>
<td>forward: out[b] = f_b(in[b])</td>
</tr>
<tr>
<td>Generate</td>
<td>Gen_b: exposed definitions</td>
<td>Gen_b: expressions evaluated</td>
</tr>
<tr>
<td>Propagate</td>
<td>in[b]-Kill_b: definitions killed</td>
<td>in[b]-Kill_b: expressions killed</td>
</tr>
<tr>
<td>Meet operation</td>
<td>U(in[b]: U out[predecessors]) ( \cap ) U(in[b]: U out[predecessors])</td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>out[entry] = ( \emptyset )</td>
<td>out[entry] = ( \emptyset )</td>
</tr>
<tr>
<td></td>
<td>out[b] = ( \emptyset )</td>
<td>out[b] = all expressions</td>
</tr>
</tbody>
</table>

III. Constant Propagation/Folding

• At every basic block boundary, for each variable v
  • determine if v is a constant
  • if so, what is the value?

\( a = 1 \)
\( x = 2 \)
\( m = x + e \)
\( a = 3 \)
\( p = a + e + 4 \)
Semi-lattice Diagram

- Finite domain?
- Finite height?

Equivalent Definition

- Meet Operation:

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v1 \land v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>undef</td>
<td>undef</td>
<td>undef</td>
</tr>
<tr>
<td>c1</td>
<td>c2</td>
<td>c1, if c1 = c2</td>
</tr>
<tr>
<td>c2</td>
<td>c2</td>
<td>c2</td>
</tr>
<tr>
<td>NAC</td>
<td>NAC</td>
<td>NAC</td>
</tr>
<tr>
<td>undef</td>
<td>undef</td>
<td>undef</td>
</tr>
<tr>
<td>c1</td>
<td>c2</td>
<td>c2</td>
</tr>
<tr>
<td>NAC</td>
<td>NAC</td>
<td>NAC</td>
</tr>
</tbody>
</table>

- Note: undef \land c2 = c2!

Example

\[
\begin{array}{c}
\text{x = 2} \\
\text{p = x}
\end{array}
\]

Transfer Function

- Assume a basic block has only 1 instruction
- Let IN[b, x], OUT[b, x]
  - be the information for variable x at entry and exit of basic block b

- OUT[entry, x] = undef, for all x.
- Non-assignment instructions: OUT[b, x] = IN[b, x]
- Assignment instructions: (next page)
Constant Propagation (Cont.)

- Let an assignment be of the form \( x_3 = x_1 + x_2 \)
  - ‘+’ represents a generic operator
  - \( \text{OUT}[b,x] = \text{IN}[b,x] \), if \( x \neq x_3 \)

<table>
<thead>
<tr>
<th>( \text{IN}[b,x_1] )</th>
<th>( \text{IN}[b,x_2] )</th>
<th>( \text{OUT}[b,x_3] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>undef</td>
<td>undef</td>
<td>undef</td>
</tr>
<tr>
<td></td>
<td>( c_2 )</td>
<td>( c_2 )</td>
</tr>
<tr>
<td></td>
<td>NAC</td>
<td>NAC</td>
</tr>
<tr>
<td>( c_1 )</td>
<td>undef</td>
<td>( c_2 )</td>
</tr>
<tr>
<td></td>
<td>NAC</td>
<td>NAC</td>
</tr>
<tr>
<td>NAC</td>
<td>undef</td>
<td>( c_2 )</td>
</tr>
<tr>
<td></td>
<td>NAC</td>
<td>NAC</td>
</tr>
</tbody>
</table>

- Use: \( x \leq y \) implies \( f(x) \leq f(y) \) to check if framework is monotone
  - \( [v_1, v_2, \ldots] \leq [v'_1, v'_2, \ldots] \), \( f([v_1, v_2, \ldots]) \leq f([v'_1, v'_2, \ldots]) \)

Summary of Constant Propagation

- A useful optimization
- Illustrates:
  - abstract execution
  - an infinite semi-lattice
  - a non-distributive problem

Distributive?

\[
\begin{align*}
n & = 2 \\
\text{y} & = 3
\end{align*}
\]

\[
\begin{align*}
x & = 3 \\
\text{y} & = 2
\end{align*}
\]

\[
z = x + y
\]

Other Optimizations

- Copy Propagation:

- Dead Code Elimination: