Lesson 33

- Data-Flow analysis for global optimization
Data-Flow Analysis

• A data-flow analysis schema defines a value at each point in the program, IN[s] and OUT[s] for each statement s
• Values are abstractions of all program states reachable in that point with an arbitrary computation path
• Statements of the program have associated transfer functions that relate the value before the statement to the value after
  – Forward OUT[s] = f (IN[s]) or backward IN[s] = f (OUT[s])
• Statements with more than one predecessor must have their value defined by combining the values at the predecessors, using a meet operator.
• Often basic blocks are annotated with values instead of individual statements: OUT[B] and IN[B]
• Useful for annotating the code with info needed for local or global optimization.
Data-Flow Analysis Framework

• A *Data-Flow Analysis Framework* \((D, V, \land, F)\) consists of:
  • A *direction* \(D\) in \{FORWARDS, BACKWARDS\}
  • A *domain of values* \((V, \land)\) which forms a *meet semilattice*:
    – A partial order with a *top element* and a binary operation *meet* \((\land, \text{greatest lower bound})\) such that
      
      \[
      x \land y \leq x \text{ and } x \land y \leq y \text{ and } (\forall z. z \leq x \text{ and } z \leq y \Rightarrow z \leq x \land y)
      \]
  • A family \(F\) of *transfer functions* from \(V\) to \(V\), including the identity function and closed under composition

• A framework is *monotone* if if for all \(f\) in \(F\)

\[
x \leq y \Rightarrow f(x) \leq f(y)
\]

• It is *distributive* if for all \(f\) in \(F\)

\[
f(x \land y) = f(x) \land f(y)
\]
Data-Flow Iterative Algorithm

- [Forward] Given:
  - a data-flow graph with ENTRY and EXIT nodes
  - one transfer function $f_B$ for each basic block $B$
  - A “boundary condition” $v_{\text{ENTRY}}$
- Computes values IN[B] and OUT[B] for all blocks

1) $\text{OUT}[\text{ENTRY}] = v_{\text{ENTRY}}$
2) while (changes to any OUT occur)
3) for (each basic block $B$ other than ENTRY){
4) $\text{IN}[B] = \bigwedge_p \text{a predecessor of } B \ \text{OUT}[P]$
5) $\text{OUT}[B] = f_B(\text{IN}[B])$
}
Example: Dataflow analysis for Reaching Definitions

• Each point in the program is associated with the set of definitions that are active at that point
• Semilattice:
  – Powerset of definitions (assignments)
  – Meet operator: union. Top element: empty set
• The *transfer function* for a block kills definitions of variables that are redefined in the block and adds definitions of variables that occur in the block:
  \[ f_B(x) = gen_B \cup (x - kill_B) \]
• The confluence operator is union.
Reaching Definitions

Then, the data-flow equations for $S$ are:

- $\text{gen}[S] = \{d\}$
- $\text{kill}[S] = D_a - \{d\}$
- $\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$

where $D_a$ = all definitions of $a$ in the region of code.
Reaching Definitions: Iterative solution

1) \( \text{OUT}[\text{ENTRY}] = \{ \} \);
2) for (each basic block \( B \)) \( \text{OUT}[B] = \{ \} \)
3) while (changes to any \( \text{OUT} \) occur)
4) for (each basic block \( B \) other than \( \text{ENTRY} \))
5) \( \text{IN}[B] = \bigcup_{\text{a predecessor of } B} \text{OUT}[P] \)
6) \( \text{OUT}[B] = gen_B \bigcup (\text{IN}[B] - \text{kill}_B) \)

- Visiting order in line 4) influences convergence
- Very efficient implementations with bit vectors
- Non-iterative solutions possible: Syntax-directed, and region-based
Dataflow analysis for Reaching Definitions towards a syntax directed algorithm

\[ d_1: i := m-1; \]
\[ d_2: j := n; \]
\[ d_3: a := u_1; \]
\[ \text{do} \]
\[ d_4: i := i+1; \]
\[ d_5: j := j-1; \]
\[ \text{if } e_1 \text{ then} \]
\[ d_6: a := u_2 \]
\[ \text{else} \]
\[ d_7: i := u_3 \]
\[ \text{while } e_2 \]

Figure 9.13: Flow graph for illustrating reaching definitions
Reaching Definitions

\[
\begin{align*}
gen[S] &= \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2]) \\
kill[S] &= \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{align*}
\]

is of the form

\[S \rightarrow S_1 \rightarrow S_2\]
Reaching Definitions

\[
\begin{align*}
gen[S] &= gen[S_1] \cup gen[S_2] \\
n\text{kill}[S] &= \text{kill}[S_1] \cap \text{kill}[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{align*}
\]
Reaching Definitions

\[ \text{gen}[S] = \text{gen}[S_1] \]
\[ \text{kill}[S] = \text{kill}[S_1] \]
\[ \text{in}[S_1] = \text{in}[S] \cup \text{gen}[S_1] \]
\[ \text{out}[S] = \text{out}[S_1] \]
Reaching Definitions: Computing Gen/Kill

d_1: \text{i} := m-1;
d_2: \text{j} := n;
d_3: \text{a} := u_1;
  \text{do}
  d_4: \text{i} := i+1;
d_5: \text{j} := j-1;
  \text{if} \ e_1 \ \text{then}
  d_6: \text{a} := u_2
  \text{else}
  d_7: \text{i} := u_3
\text{while} \ e_2;
Using Bit-Vectors to Compute Reaching Definitions

\[ d_1 \cdot d_2 \cdot d_3 \cdot d_4 \cdot d_5 \cdot d_6 \cdot d_7 \]

\[ \text{gen=} \quad 0011111 \]
\[ \text{kill=} \quad 1100000 \]

\[ d_1 \]

\[ d_2 \]

\[ d_3 \]

\[ \text{do} \]

\[ d_4 \]

\[ d_5 \]

\[ \text{if} \]

\[ e_1 \]

\[ e_2 \]

\[ d_6 \]

\[ d_7 \]

\[ \text{gen=} \]
\[ \text{kill=} \]

\[ d_1: i := m-1; \]
\[ d_2: j := n; \]
\[ d_3: a := u1; \]
\[ \quad \text{do} \]
\[ d_4: i := i+1; \]
\[ d_5: j := j-1; \]
\[ \quad \text{if} \; e_1 \; \text{then} \]
\[ d_6: a := u2 \]
\[ \quad \text{else} \]
\[ d_7: i := u3 \]
\[ \quad \text{while} \; e_2 \]
Reaching Definitions: Computing In/Out (non-iterative)

\[\begin{align*}
&d_1: i := m-1; \\
&d_2: j := n; \\
&d_3: a := u1; \\
&\text{do} \\
&d_4: i := i+1; \\
&d_5: j := j-1; \\
&\text{if } e1 \text{ then} \\
&d_6: a := u2 \\
&\text{else} \\
&d_7: i := u3 \\
&\text{while } e2
\end{align*}\]
Accuracy, Safeness, and Conservative Estimations

- **Conservative**: refers to making safe assumptions when insufficient information is available at compile time, i.e. the compiler has to guarantee not to change the meaning of the optimized code.
- **Safe**: refers to the fact that a superset of reaching definitions is safe (some may have been killed).
- **Accuracy**: more and better information enables more code optimizations.
Reaching Definitions are a Conservative (Safe) Estimation

Suppose this branch is never taken

Estimation:
- $gen[S] = gen[S_1] \cup gen[S_2]$
- $kill[S] = kill[S_1] \cap kill[S_2]$

Accurate:
- $gen'[S] = gen[S_1] \subseteq gen[S]$
- $kill'[S] = kill[S_1] \supseteq kill[S]$
Example: Dataflow analysis for Live Variables [backwards!]

• Each point in the program is associated with the set of variables that are *live* at that point, i.e. such that their value will be used later

• Semilattice:
  – Powerset of variables
  – Meet operator: union. Top element: empty set

• A variable is *live* at the beginning of a block if it is either used before definition in the block or is live at the end of the block and not redefined in the block.

• The *transfer function*: \( f_B(x) = \text{use}_B \cup (x - \text{def}_B) \)

• The confluence operator is union.
Data-Flow Analysis for Live Variables: an example

Solution:

\[ \text{in}[B1] = \{m, n\} \cup (\{i, j\} - \{i, j\}) = \{m, n\} \]

\[ \text{out}[B1] = \text{in}[B2] = \{i, j\} \]

\[ \text{in}[B2] = \{j\} \cup (\{i, j\} - \{j\}) = \{i, j\} \]

\[ \text{out}[B2] = \text{in}[B3] = \{i, j\} \]
Live variables: Iterative solution

1) \( \text{IN(EXIT)} = \{ \} \); 
2) for (each basic block B) \( \text{IN}[B] = \{ \} \) 
3) while (changes to any IN occur) 
4) for (each basic block B other than EXIT) 
5) \( \text{OUT}[B] = U_{S \text{ a successor of } B} \text{IN}[S] \); 
6) \( \text{IN}[B] = use_B U (\text{OUT}[B] - \text{def}_B) \) 
   
}
Constant Propagation/Folding

• Unbounded set of values:
  – All constants for the relevant type
  – NAC: not-a-constant
  – UNDEF: no info about any value of the variable

• The semilattice:
Constant Propagation/Folding

• Transfer function for statements:
  1. Identity, if it is not an assignment
  2. If it is an assignment to x:
     1.  $m'(v) = m(v)$ for $v \neq x$
     2.  If the RHS is a constant $c$, $m'(x) = c$
     3.  If the RHS is “$y \ op \ z$”,
         1.  If $m(y)$ and $m(z)$ are constant, $m'(x) = m(y) \ op \ m(z)$
         2.  If $m(y) = \text{NAC}$ or $m(z) = \text{NAC}$, then $m'(x) = \text{NAC}$
         3.  $m'(x) = \text{UNDEF}$, otherwise
     4.  If the RHS is anything else (e.g. function call) $m'(x) = \text{NAC}$
Constant Propagation/Folding

- Transfer functions are monotonic but not distributive

```
+---+     +---+  +---+
|   | →   |   |→ |
| B1|     | B2|   | B3|
+---+     +---+  +---+
|   |     |   |→ |
| x = 2 | x = 3 | y = 2 | y = 3 |
+---+     +---+  +---+
|     |     |   |→ |
| z = x + y |   |   |→ |
+---+     +---+  +---+
|     |     |     |→ |
|   |     |     | EXIT |
```

<table>
<thead>
<tr>
<th>m</th>
<th>m(x)</th>
<th>m(y)</th>
<th>m(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_0</td>
<td>UNDEF</td>
<td>UNDEF</td>
<td>UNDEF</td>
</tr>
<tr>
<td>f_1(m_0)</td>
<td>2</td>
<td>3</td>
<td>UNDEF</td>
</tr>
<tr>
<td>f_2(m_0)</td>
<td>3</td>
<td>2</td>
<td>UNDEF</td>
</tr>
<tr>
<td>f_1(m_0) ∧ f_2(m_0)</td>
<td>NAC</td>
<td>NAC</td>
<td>UNDEF</td>
</tr>
<tr>
<td>f_3(f_1(m_0) ∧ f_2(m_0))</td>
<td>NAC</td>
<td>NAC</td>
<td>NAC</td>
</tr>
<tr>
<td>f_3(f_1(m_0))</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>f_3(f_2(m_0))</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>f_3(f_1(m_0)) ∧ f_3(f_2(m_0))</td>
<td>NAC</td>
<td>NAC</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ f_3(f_1(m_0) ∧ f_2(m_0)) < f_3(f_1(m_0)) ∧ f_3(f_2(m_0)) \]
Figure 9.30: Examples of (a) global common subexpression, (b) loop-invariant code motion, (c) partial-redundancy elimination.
On Partial-Redundancy Elimination

• Four step “Lazy Code Motion” algorithm
  – Find blocks where evaluation of an expression can be anticipated (backwards)
  – Check availability of expressions along all paths leading to a block needing it (forwards)
  – Postpone the expression as much as possible (forwards)
  – Eliminate assignments to temporaries that are used only once (backwards)
Determining Loops in Flow Graphs

• In absence of loops data-flow analysis converges in one pass, if performed according to topological order
• Study of loops needed also to evaluate convergence speed
• For some values semi-lattices, loops do not modify values, so they can be ignored
• For others, several iterations in loops are needed: eg, constant folding

L:   x = y;
     y = z;
     z = 1;
     goto L
Determining Loops in Flow Graphs: Dominators

• Dominators: $d \text{ dom } n$
  – Node $d$ of a CFG dominates node $n$ if every path from the initial node of the CFG to $n$ goes through $d$
  – The loop entry dominates all nodes in the loop

• The immediate dominator $m$ of a node $n$ is the last dominator on the path from the initial node to $n$
  – If $d \neq n$ and $d \text{ dom } n$ then $d \text{ dom } m$
Dominator Trees

CFG

Dominator tree
Data-Flow analysis for Dominators

• Computes $D(n)$, set of dominators for each node $n$ (forwards)
• Semilattice: powerset of CFG nodes
• Transfer function: $f_B(x) = x \cup \{B\}$
• Meet operator: intersection
• Boundary: $\text{OUT}[\text{ENTRY}] = \{\text{ENTRY}\}$
• Initialization: $\text{OUT}[B] = \text{NODES}$
Natural Loops

• A back edge is an edge $a \rightarrow b$ whose head $b$ dominates its tail $a$

• Given a back edge $n \rightarrow d$
  – The natural loop consists of $d$ plus the nodes that can reach $n$ without going through $d$
  – The loop header is node $d$

• In other words
  – A natural loop must have a single-entry node $d$
  – There must be a back edge that enters node $d$
Natural Inner/Outer Loops

- Unless two loops have the same header, they are disjoint or one is nested within the other.
- A nested loop is an *inner loop* if it contains no other loops.
- A loop is an *outer loop* if it is not contained within another loop.
Natural Inner Loops Example

CFG

Dominator tree

Natural loop for 3 \textit{dom} 4
Natural loop for 4 \textit{dom} 7
Natural loop for 7 \textit{dom} 10
Natural Outer Loops Example

Natural loop for 1 \textit{dom} 9

Natural loop for 3 \textit{dom} 8

CFG

Dominator tree
Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion (of loop invariant code), strength reduction, and other loop transformations populate the preheader
Reducible Flow Graphs

- *Reducible graph* = disjoint partition in forward and back edges such that the forward edges form an acyclic (sub)graph

**Example of a reducible CFG**

**Example of a nonreducible CFG**
(not a natural loop: no back edge to dominator 1)
Speed of convergence of data-flow analysis

• Maximum number of iterations: (height of the lattice) x (number of nodes)

• If value of interest can be propagated along acyclic path (reaching definitions, available expressions, live variables), few passes are sufficient in general, depending on the depth of the graph (~ number of loop nesting).