Data Flow Analysis

CS 544
Baris Aktemur

• Contents from
  – Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman

• And, where noted as EaC2e, from
  – Keith D. Cooper and Linda Torczon
    *Engineering a Compiler*, Second Edition
    Morgan Kaufmann, 2011
Data Flow Analysis (DFA)

- Find opportunities for improving the efficiency of the code
- We must be sure that a particular transformation is safe
- DFA: Compile-time reasoning about the runtime flow of values
- Performed on Control Flow Graph (CFG)

Figure 9.30: Examples of (a) global common subexpression, (b) loop-invariant code motion, (c) partial-redundancy elimination.
Dominance (from EaC2e)

- Forward data-flow problem:
  - Compute a node's data based on its predecessors'

\[
\text{\textbf{Iterative approach to solve the Dominance problem}}
\]

\[
\text{Initial conditions:}\quad \text{Dom}(n_0) = \{n_0\}, \quad \forall n \neq n_0, \quad \text{Dom}(n) = N
\]
Dominance (from EaC2e)

- A 3-step process
  - Form a CFG
  - Compute initial information for each block
  - Solve the equations to find final information for each block

- Will see this process for any data-flow problem
Dominance (from EaC2e)

- Termination
- Correctness
- Efficiency

Termination of Dominance (from EaC2e)

- Dom sets monotonically shrink
- For the initial node, start with itself; for all others, start with N.
- A Dom set cannot grow (check the algorithm)
- A Dom set cannot be smaller than a single-element set.
- Hence, the while-loop eventually terminates
Correctness of Dominance (from EaC2e)

- There exists a unique fixed-point for the equations we solved
- The algorithm finds that unique solution
- Details are beyond our scope. Food for thought…

Efficiency of Dominance (from EaC2e)

- Unique solution => Order of computing the sets is irrelevant.
- Pick your favorite traversal
- A reverse postorder (rpo) traversal of the graph is particularly effective
- Idea: visit a node before its successors.
Efficiency of Dominance (from EaC2e)

- For a forward data-flow problem, use an RPO computed on the CFG.
- For a backward data-flow, use an RPO computed on the reverse CFG.
- Look up the definition of preorder, postorder, and reverse postorder traversal in your favorite graph theory course/book.

Two passes, rather than three.
Efficiency of Dominance (from EaC2e)

More than two passes needed.

Data Flow Analysis

- Dominance is based only on the structure of the graph.
  - a form of control-flow analysis.
- Behavior of the code is ignored.
- Most data-flow problems reason about the behavior of the code.
Data Flow Abstraction

• Program State:
  – Values of all the variables
  – Value of the program counter
• Execution of a program
  – Series of transformations of the program state
• Each statement transforms an input state to an output state

Data Flow Abstraction

• Data Flow Analysis
  – Extract information for all the possible program states
  – Regarding the problem we’re trying to solve
• Must consider all the possible paths
• An abstraction of the all possible executions
• Complex problems: Interprocedural
• This lecture: Intraprocedural
Data Flow Abstraction

- Program points: just before or after executing a statement
- Program state/data are associated with program points
- Within one basic block, the program point after a statement is the same as the program point before the next statement.
- Execution path: sequence of program points

Data Flow Abstraction

- In general, there is an infinite number of possible execution paths
- No finite upper bound on the length of an execution path
- Program analyses summarize all the possible program states that can occur at a point in the program with a finite set of facts
- Summary is analysis-dependent
Reaching Definitions

• A definition of a variable x is a statement that assigns a value to x. (ignore aliasing for simplicity)
• “What definitions of the variable x may be reaching at point p?”

• The first time program point (5) is executed, the value of a is 1 due to definition d1.
• In subsequent iterations, d3 reaches point (5) and the value of a is 243.
• At point (5), the value of a is one of {1,243}.
• It may be defined by one of {d1,d3}.
Reaching Definitions Exercise

- What def. of i/j/a are reaching specified points?
  - \{d1,d2,d3,d5,d6,d7\}
  - \{d3,d4,d5,d6\}
  - \{d3,d4,d5,d6\}

Data Flow Abstraction

- Reaching definitions: The definitions that may reach a program point along some path.
- Constant propagation: The unique definition that reaches a point, AND that has a constant value.
  - Distinguish def's as constant vs. non-constant
  - Same information, different summary
DFA Schema

- **Domain**: The set of possible DFA values
  - Analysis-specific
- **IN[s]**: data-flow values before statement s
- **OUT[s]**: data-flow values after statement s
- The data-flow problem is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s, for all statements s.

DFA Schema

- **Transfer function**: How a statement changes the data-flow values
  - Analysis- and statement-specific
- **Forward flow**:
  - \( \text{OUT}[s] = f_s(\text{IN}[s]) \)
- **Backward flow**
  - \( \text{IN}[s] = f_s(\text{OUT}[s]) \)
DFA Schema

• Data flow values within a basic block
  – $\text{IN}[s_{i+1}] = \text{OUT}[s_i]$
  – Note that this is an equality; no difference for forward vs. backward

• Suppose block $B$ consists of statements $s_1, \ldots, s_n$, in that order
  – $\text{IN}[B] = \text{IN}[s_1]$
  – $\text{OUT}[B] = \text{OUT}[s_n]$
  – $\text{OUT}[B] = f_B(\text{IN}[B])$ where $f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$

DFA Schema

• Constraints due to control flow between basic blocks
  – E.g: Definitions that may reach a point (a forward problem)
    \[ \text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P] \]
  – E.g: Live variables (backward problem)
    \[
    \text{IN}[B] = f_B(\text{OUT}[B]) \\
    \text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]
    \]
Reaching Definitions

• A definition $d$ reaches a point $p$ if there is a path from the point immediately following $d$ to $p$, such that $d$ is not “killed” along that path.

• We kill a definition of a variable $x$ if there is any other definition of $x$ anywhere along the path.

Transfer Equations

$$d: u = v + w$$

• This statement “generates” a definition $d$ of variable $u$ and "kills" all the other definitions in the program that define variable $u$, while leaving the remaining incoming definitions unaffected.

$$f_d(x) = gen_d \cup (x - kill_d)$$

where $gen_d$ is $\{d\}$ and $kill_d$ is the set of all other definitions of $u$. 
Transfer Equations

• Composition.
  – Suppose we have
  \[ f_1(x) = \text{gen}_1 \cup (x - \text{kill}_1) \text{ and } f_2(x) = \text{gen}_2 \cup (x - \text{kill}_2) \]
  then
  \[ f_2(f_1(x)) = \text{gen}_2 \cup (\text{gen}_1 \cup (x - \text{kill}_1) - \text{kill}_2) \]
  \[ = (\text{gen}_2 \cup (\text{gen}_1 - \text{kill}_2)) \cup (x - (\text{kill}_1 \cup \text{kill}_2)) \]

Transfer Equations

• Composition.
  – Transfer function of a block B with n statements:
  \[ f_B(x) = \text{gen}_B \cup (x - \text{kill}_B), \]
  where
  \[ \text{kill}_B = \text{kill}_1 \cup \text{kill}_2 \cup \cdots \cup \text{kill}_n \]
  and
  \[ \text{gen}_B = \text{gen}_n \cup (\text{gen}_{n-1} - \text{kill}_n) \cup (\text{gen}_{n-2} - \text{kill}_{n-1} - \text{kill}_n) \cup \cdots \cup (\text{gen}_1 - \text{kill}_2 - \text{kill}_3 - \cdots - \text{kill}_n) \]
Figure 9.13: Flow graph for illustrating reaching definitions
Control Flow Equations

\[ \text{IN}[B] = \bigcup_P \text{a predecessor of } B \ \text{OUT}[P] \]

\[ \text{OUT}[B] = \text{gen}_B \cup (\text{IN}[B] - \text{kill}_B) \]

\[ \text{OUT}[\text{ENTRY}] = \emptyset \quad \text{(Initial condition)} \]

- Solution to the equations above is a fixed-point of the system. We are interested in finding the least fixed-point.

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

Figure 9.14: Iterative algorithm to compute reaching definitions

- Note the three-step process
  - Build a CFG (already done)
  - Initialize local information
  - Compute global information (i.e. propagate local info until the fixed-point)
Represent sets by bit-vectors

<table>
<thead>
<tr>
<th>Block B</th>
<th>( \text{OUT}[B]_0 )</th>
<th>( \text{IN}[B]_1 )</th>
<th>( \text{OUT}[B]_1 )</th>
<th>( \text{IN}[B]_2 )</th>
<th>( \text{OUT}[B]_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_1 )</td>
<td>000 0000</td>
<td>000 0000</td>
<td>111 0000</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>( B_2 )</td>
<td>000 0000</td>
<td>111 0000</td>
<td>001 1100</td>
<td>111 0111</td>
<td>001 1110</td>
</tr>
<tr>
<td>( B_3 )</td>
<td>000 0000</td>
<td>001 1100</td>
<td>000 1110</td>
<td>001 1110</td>
<td>000 1110</td>
</tr>
<tr>
<td>( B_4 )</td>
<td>000 0000</td>
<td>001 1110</td>
<td>001 0111</td>
<td>001 1110</td>
<td>001 0111</td>
</tr>
<tr>
<td>EXIT</td>
<td>000 0000</td>
<td>001 0111</td>
<td>001 0111</td>
<td>001 0111</td>
<td>001 0111</td>
</tr>
</tbody>
</table>

E.g:

\[
\text{IN}[B_2]^1 = \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 = 111 0000 + 000 0000 = 111 0000
\]

\[
\text{OUT}[B_2]^1 = \text{gen}[B_2] \cup (\text{IN}[B_2]^1 - \text{kill}[B_2])
\]

\[
= 000 1100 + (111 0000 - 110 0001) = 001 1100
\]

Reaching Definitions

- Detecting uses before definitions (i.e. uninitialized variables)
  - Introduce a dummy definition for each variable \( x \) in the entry to the flow graph. If the dummy definition of \( x \) reaches a point \( p \) where \( x \) might be used, then there might be an opportunity to use \( x \) before definition.
Live Variables

- Can the value of x at p be used along some path in the flow graph starting at p?
- If so, x is *live*, otherwise, *dead* at p.
- Important analysis for register allocation.
- Backward analysis.

Transfer Functions

- \( \text{def}_B \): the set of variables *defined* (i.e., definitely assigned values) in B
- \( \text{use}_B \): the set of variables whose values may be used in B prior to any definition of the variable. (i.e. upwards exposed variables)
Live Variables

- Constraints
  \[
  \text{IN}[B] = \text{use}_B \cup (\text{OUT}[B] - \text{def}_B) \\
  \text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]
  \]

- Initial condition
  \[
  \text{IN}[\text{EXIT}] = \emptyset
  \]
LV vs. RD

• Both have union as the meet operator: In each, we care only about whether a path with desired properties exists, rather than whether something is true along all paths.

• Information flow for liveness travels "backward," whereas “forward” in reachability.

• gen/kill vs use/def.

\[
\begin{align*}
\text{IN}[\text{EXIT}] &= \emptyset; \\
\text{for (each basic block } B \text{ other than } \text{EXIT}) \quad \text{IN}[B] &= \emptyset; \\
\text{while (changes to any } \text{IN occur}) \\
\quad \text{for (each basic block } B \text{ other than } \text{EXIT}) \\
\qquad \text{OUT}[B] &= \bigcup S \text{ a successor of } B \text{ IN}[S]; \\
\qquad \text{IN}[B] &= \text{use}_B \cup (\text{OUT}[B] - \text{def}_B); \\
\end{align*}
\]

Figure 9.16: Iterative algorithm to compute live variables
RPO on CFG:

<table>
<thead>
<tr>
<th>LiveOut(n)</th>
<th>B₀</th>
<th>B₁</th>
<th>B₂</th>
<th>B₃</th>
<th>B₄</th>
<th>B₅</th>
<th>B₆</th>
<th>B₇</th>
<th>B₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td>1</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>[a, b, c, d, l]</td>
<td>ø</td>
</tr>
<tr>
<td>2</td>
<td>ø</td>
<td>[a, l]</td>
<td>[a, b, c, d, l]</td>
<td>[l]</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>[a, c, d, l]</td>
<td>[a, b, d, c, l]</td>
</tr>
<tr>
<td>3</td>
<td>[l]</td>
<td>[a, l]</td>
<td>[a, b, c, d, l]</td>
<td>[l]</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>[a, c, d, l]</td>
<td>[a, c, d, l]</td>
</tr>
<tr>
<td>4</td>
<td>[l]</td>
<td>[a, c, l]</td>
<td>[a, b, c, d, l]</td>
<td>[l]</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>[a, c, d, l]</td>
<td>[a, c, d, l]</td>
</tr>
<tr>
<td>5</td>
<td>[l]</td>
<td>[a, c, l]</td>
<td>[a, b, c, d, l]</td>
<td>[l]</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
<td>[a, c, d, l]</td>
<td>[a, c, d, l]</td>
</tr>
</tbody>
</table>

- If computed on RPO of the reverse CFG

From EaC2e
Uninitialized Variables

• How can you use Live Variable analysis to detect if there may be uninitialized variables? (i.e. variables that are being used before being defined)
  – Check OUT[entry]. If non-empty, there may be a problem.

Of course, this is a conservative analysis. There may be false positives. Consider the following code (Taken from EaC2e)

```c
main() {
    int i, n, s;
    scanf("%d", &n);
    i = 1;
    while (i<=n) {
        if (i==1)
            s = 0;
        s = s + i++;
    }
}
```
Available Expressions

• Expression $x+y$ is available at point $p$ if
  – every path from the entry node to $p$ evaluates $x+y$, and
  – after the last such evaluation prior to reaching $p$, there are no assignments to $x$ or $y$

• Useful for common subexpression elimination
Available Expressions

- A block *kills* expression $x+y$ if it assigns $x$ or $y$ and does not subsequently recompute $x+y$.
- A block *generates* expression $x+y$ if it definitely evaluates $x+y$ and does not subsequently define $x$ or $y$.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Available Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b + c$</td>
<td>${b + c}$</td>
</tr>
<tr>
<td>$b = a - d$</td>
<td>${a - d}$</td>
</tr>
<tr>
<td>$c = b + c$</td>
<td>${a - d}$</td>
</tr>
<tr>
<td>$d = a - d$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

OUT[ENTRY] = $\emptyset$;
for (each basic block $B$ other than ENTRY) OUT[$B$] = $U$;
while (changes to any OUT occur)
  for (each basic block $B$ other than ENTRY) {
    IN[$B$] = $\bigcap_P$ a predecessor of $B$ OUT[$P$];
    OUT[$B$] = $e_{gen_B} \cup (IN[$B$] - e_{kill_B})$;
  }

Figure 9.20: Iterative algorithm to compute available expressions

- Meet operation is intersection.
- OUT[B] are set to $U$, except the entry node.
  - $U$ is the universal set of expressions.
Summary

<table>
<thead>
<tr>
<th>Reaching Definitions</th>
<th>Live Variables</th>
<th>Available Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>Sets of definitions</td>
<td>Sets of variables</td>
</tr>
<tr>
<td>Direction</td>
<td>Forwards</td>
<td>Backwards</td>
</tr>
<tr>
<td>Transfer function</td>
<td>$\text{gen}_B \cup (x - \text{kill}_B)$</td>
<td>$\text{use}_B \cup (x - \text{def}_B)$</td>
</tr>
<tr>
<td>Boundary</td>
<td>$\text{OUT}[\text{ENTRY}] = \emptyset$</td>
<td>$\text{IN}[\text{EXIT}] = \emptyset$</td>
</tr>
<tr>
<td>Meet ($\wedge$)</td>
<td>$\bigcup$</td>
<td>$\bigcap$</td>
</tr>
<tr>
<td>Equations</td>
<td>$\text{OUT}[B] = f_B(\text{IN}[B])$</td>
<td>$\text{IN}[B] = f_B(\text{OUT}[B])$</td>
</tr>
<tr>
<td></td>
<td>$\text{IN}[B] = \bigwedge_{P, \text{pred}(B)} \text{OUT}[P]$</td>
<td>$\text{OUT}[B] = \bigwedge_{S, \text{succ}(B)} \text{IN}[S]$</td>
</tr>
<tr>
<td>Initialize</td>
<td>$\text{OUT}[B] = \emptyset$</td>
<td>$\text{IN}[B] = \emptyset$</td>
</tr>
</tbody>
</table>

Exercise: Compute

- def, use, IN and OUT for LV analysis.
- e_gen, e_kill, IN and OUT for AE analysis.
Interprocedural Summary Problems (from EaC2e)

• Function calls significantly degrade the information collected by an analysis
  – For safety, we have to assume that the callee function may modify any global or pass-by-ref variable

• Interprocedural may modify problem:
  – Determine which variables may be modified by called functions.
  – A data-flow analysis on the call graph
    • Flow insensitive

\[
\text{MAYMOD}(p) = \text{LOCALMOD}(p) \cup \bigcup_{e=(p,q)} \text{unbind}_e(\text{MAYMOD}(q))
\]

• \text{unbind} function maps one set of variables into another
• \( e \) is an edge in the call graph