Data-Flow Analysis

Dragon Book, Chapter 9, Section 9.2, 9.3, 9.4
Data-Flow Analysis

• Data-flow analysis is a sub-area of **static program analysis** (aka **compile-time** analysis)
  – Used in the compiler back end for optimizations of three-address code and for generation of target code
  – For software engineering tools: software understanding, restructuring, testing, verification

• Attaches to each CFG node some information that describes **properties** of the program at that point
  – Based on **lattice theory**

• Defines algorithms for inferring these properties
  – e.g., **fixed-point computation**
Example: Reaching Definitions

• A classical example of a data-flow analysis
  – We will consider *intraprocedural* analysis: only inside a single procedure, based on its CFG

• For ease of discussion, pretend that the CFG nodes are individual instructions, not basic blocks
  – Each node defines two *program points*: immediately before and immediately after

• Goal: identify all connections between variable definitions (“write”) and variable uses (“read”)
  – $x = y + z$ has a *definition* of $x$ and *uses* of $y$ and $z$
Reaching Definitions

• A definition $d$ reaches a program point $p$ if there exists a CFG path that
  – starts at the program point immediately after $d$
  – ends at $p$
  – does not contain a definition of $d$ (i.e., $d$ is not “killed”)

• The CFG path may be impossible (*infeasible*) at run time
  – Any compile-time analysis has to be *conservative*, so we consider all paths in the CFG

• For a CFG node $n$
  – $\text{IN}[n]$ is the set of definitions that reach the program point immediately before $n$
  – $\text{OUT}[n]$ is the set of definitions that reach the program point immediately after $n$
  – Reaching definitions analysis computes $\text{IN}[n]$ and $\text{OUT}[n]$
ENTRY

\( i = m-1 \)

\( j = n \)

\( a = u_1 \)

\( i = i + 1 \)

\( j = j - 1 \)

if \( (i < a) \)

\( a = u_2 \)

\( i = u_3 \)

if \( (j < a) \)

EXIT

\[ \text{OUT}[n_1] = \{ \} \]
\[ \text{IN}[n_2] = \{ \} \]
\[ \text{OUT}[n_2] = \{ \text{d}1 \} \]
\[ \text{IN}[n_3] = \{ \text{d}1 \} \]
\[ \text{OUT}[n_3] = \{ \text{d}1, \text{d}2 \} \]
\[ \text{IN}[n_4] = \{ \text{d}1, \text{d}2 \} \]
\[ \text{OUT}[n_4] = \{ \text{d}1, \text{d}2, \text{d}3 \} \]
\[ \text{IN}[n_5] = \{ \text{d}1, \text{d}2, \text{d}3, \text{d}5, \text{d}6, \text{d}7 \} \]
\[ \text{OUT}[n_5] = \{ \text{d}2, \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{IN}[n_6] = \{ \text{d}2, \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{OUT}[n_6] = \{ \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{IN}[n_7] = \{ \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{OUT}[n_7] = \{ \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{IN}[n_8] = \{ \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{OUT}[n_8] = \{ \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{IN}[n_9] = \{ \text{d}3, \text{d}4, \text{d}5, \text{d}6 \} \]
\[ \text{OUT}[n_9] = \{ \text{d}3, \text{d}5, \text{d}6, \text{d}7 \} \]
\[ \text{IN}[n_{10}] = \{ \text{d}3, \text{d}5, \text{d}6, \text{d}7 \} \]
\[ \text{OUT}[n_{10}] = \{ \text{d}3, \text{d}5, \text{d}6, \text{d}7 \} \]
\[ \text{IN}[n_{11}] = \{ \text{d}3, \text{d}5, \text{d}6, \text{d}7 \} \]
Uses of Reaching Definitions Analysis

• Def-use (du) chains
  – For a given definition (i.e., write) of a variable, which statements read the value created by the def?

• Use-def (ud) chains
  – For a given use (i.e., read) of a variable, which statements performed the write of this value?
  – The reverse of du-chains

• Goal: potential write-read (flow) data dependences
  – Compiler optimizations
  – Program understanding (e.g., slicing)
  – Data-flow-based testing: coverage criteria
  – Semantic checks: e.g., use of uninitialized variables
ENTRY

\( i = m - 1 \)

\( j = n \)

\( a = u_1 \)

\( i = i + 1 \)

\( j = j - 1 \)

\( a = u_2 \)

\( i = u_3 \)

\( \text{if } (i < a) \)

\( \text{if } (j < a) \)

EXIT

OUT\[n1\] = \{ \}

IN\[n2\] = \{ \}

OUT\[n2\] = \{ d1 \}

IN\[n3\] = \{ d1 \}

OUT\[n3\] = \{ d1, d2 \}

IN\[n4\] = \{ d1, d2 \}

OUT\[n4\] = \{ d1, d2, d3 \}

IN\[n5\] = \{ d1, d2, d3, d5, d6, d7 \}

OUT\[n5\] = \{ d2, d3, d4, d5, d6 \}

IN\[n6\] = \{ d2, d3, d4, d5, d6 \}

OUT\[n6\] = \{ d3, d4, d5, d6 \}

IN\[n7\] = \{ d3, d4, d5, d6 \}

OUT\[n7\] = \{ d3, d4, d5, d6 \}

IN\[n8\] = \{ d3, d4, d5, d6 \}

OUT\[n8\] = \{ d4, d5, d6 \}

IN\[n9\] = \{ d3, d4, d5, d6 \}

OUT\[n9\] = \{ d3, d5, d6, d7 \}

IN\[n10\] = \{ d3, d5, d6, d7 \}

OUT\[n10\] = \{ d3, d5, d6, d7 \}

IN\[n11\] = \{ d3, d5, d6, d7 \}

Def-use chains for \( d_1 \):

DU(\( d_1 \)): uses of \( i \) in nodes with \( d_1 \in \text{IN}[n] \)

DU(\( d_1 \)) = \{ n_5 \}

Other examples:

DU(\( d_2 \)) = \{ n_6 \}

DU(\( d_3 \)) = \{ n_7, n_{10} \}

DU(\( d_4 \)) = \{ n_7 \}

DU(\( d_5 \)) = \{ n_{10}, n_6 \}

DU(\( d_6 \)) = \{ n_{10}, n_7 \}

DU(\( d_7 \)) = \{ n_5 \}

Use-def chains:

UD(\( i @ n_5 \)) = \{ d_1, d_7 \}

UD(\( j @ n_6 \)) = \{ d_2, d_5 \}

UD(\( i @ n_7 \)) = \{ d_4 \}

UD(\( a @ n_7 \)) = \{ d_3, d_6 \}

UD(\( j @ n_{10} \)) = \{ d_5 \}

UD(\( a @ n_{10} \)) = \{ d_3, d_6 \}
Example: Live Variables

• A variable $v$ is **live** at a program point $p$ if there exists a CFG path that
  – starts at $p$
  – ends immediately before some statement that reads $v$
  – does **not** contain a definition of $v$

• Thus, the value that $v$ has at $p$ could be used later
  – “could” because the CFG path may be infeasible
  – If $v$ is not live at $p$, we say that $v$ is **dead** at $p$

• For a CFG node $n$
  – $\text{IN}[n]$ is the set of variables that are live at the program point immediately before $n$
  – $\text{OUT}[n]$ is the set of variables that are live at the program point immediately after $n$
ENTRY

\[ i = m - 1 \]

\[ j = n \]

\[ a = u_1 \]

\[ i = i + 1 \]

\[ j = j - 1 \]

if (…)

\[ a = u_2 \]

\[ i = u_3 \]

if (…)

EXIT

\[ \text{OUT}[n1] = \{ m, n, u_1, u_2, u_3 \} \]

\[ \text{IN}[n2] = \{ m, n, u_1, u_2, u_3 \} \]

\[ \text{OUT}[n2] = \{ n, u_1, i, u_2, u_3 \} \]

\[ \text{IN}[n3] = \{ n, u_1, i, u_2, u_3 \} \]

\[ \text{OUT}[n3] = \{ u_1, i, j, u_2, u_3 \} \]

\[ \text{IN}[n4] = \{ u_1, i, j, u_2, u_3 \} \]

\[ \text{OUT}[n4] = \{ i, j, u_2, u_3 \} \]

\[ \text{IN}[n5] = \{ i, j, u_2, u_3 \} \]

\[ \text{OUT}[n5] = \{ j, u_2, u_3 \} \]

\[ \text{IN}[n6] = \{ j, u_2, u_3 \} \]

\[ \text{OUT}[n6] = \{ u_2, u_3, j \} \]

\[ \text{IN}[n7] = \{ u_2, u_3, j \} \]

\[ \text{OUT}[n7] = \{ u_2, u_3, j \} \]

\[ \text{IN}[n8] = \{ u_2, u_3, j \} \]

\[ \text{OUT}[n8] = \{ u_3, j, u_2 \} \]

\[ \text{IN}[n9] = \{ u_3, j, u_2 \} \]

\[ \text{OUT}[n9] = \{ i, j, u_2, u_3 \} \]

\[ \text{IN}[n10] = \{ i, j, u_2, u_3 \} \]

\[ \text{OUT}[n10] = \{ i, j, u_2, u_3 \} \]

\[ \text{IN}[n11] = \{ \} \]

Uses of Live Variables
- Dead code elimination: e.g., when \( x \) is not live at \( x = y + z \)
- Register allocation
Example: Constant Propagation

• Can we guarantee that the value of a variable $v$ at a program point $p$ is always a known constant?

• Compile-time constants are quite useful
  – **Constant folding**: e.g., if we know that $v$ is always 3.14 immediately before $w = 2*v$; replace it $w = 6.28$
  – Often due to symbolic constants
  – **Dead code elimination**: e.g., if we know that $v$ is always false at \textbf{if} ($v$) ...  
  – Program understanding, restructuring, verification, testing, etc.

• Very similar to the abstract interpretation we discussed earlier
Basic Ideas

• At each CFG node $n$, $\text{IN}[n]$ is a map $\text{Vars} \rightarrow \text{Values}$
  – Each variable $v$ is mapped to a value $x \in \text{Values}$
  – $\text{Values} = \text{all possible constant values} \cup \{\text{any}\}$

• Special value $\text{any}$ (not-a-constant) means that the variable cannot be definitely proved to be a compile-time constant at this program point
  – E.g., the value comes from user input, file I/O, network
  – E.g., the value is 5 along one branch of an if statement, and 6 along another branch of the if statement
  – E.g., value comes from some variable with $\text{any}$ value
Formulation as a System of Equations

- OUT[ENTRY] = empty map

- For any other CFG node $n$
  - $IN[n] = \text{Merge}(OUT[m])$ for all predecessors $m$ of $n$
  - $OUT[n] = \text{Update}(IN[n])$

- Merging two maps: if $v$ is mapped to $c_1$ and $c_2$ respectively, in the merged map $v$ is mapped to:
  - if $c_1 = \text{any}$ or $c_2 = \text{any}$, the result is $\text{any}$
  - Else if $c_1 \neq c_2$, the result is $\text{any}$
  - Else the result is $c_1$ (in this case we know that $c_1 = c_2$)
  - Remember IfStmt from Project 4?
Formulation as a System of Equations

- **Updating** a map at an assignment \( \mathbf{v} = \ldots \)
  - If the statement is not an assignment, \( \text{OUT}[n] = \text{IN}[n] \)
- The map does not change for any \( \mathbf{w} \neq \mathbf{v} \)
- If we have \( \mathbf{v} = \mathbf{c} \), where \( \mathbf{c} \) is a constant: in \( \text{OUT}[n] \), \( \mathbf{v} \) is now mapped to \( \mathbf{c} \)
- If we have \( \mathbf{v} = \mathbf{p} + \mathbf{q} \) (or similar binary operators) and \( \text{IN}[n] \) maps \( \mathbf{p} \) and \( \mathbf{q} \) to \( \mathbf{c}_1 \) and \( \mathbf{c}_2 \) respectively
  - If both \( \mathbf{c}_1 \) and \( \mathbf{c}_2 \) are constants: result is \( \mathbf{c}_1 + \mathbf{c}_2 \)
  - Else, \( \mathbf{c}_1 \) or \( \mathbf{c}_2 \) or both are \textit{any} and the result is \textit{any}
ENTRY

\[
a = 1
\]

\[
b = 2
\]

\[
c = a + b
\]

\[
\text{if (…)}
\]

\[
a = 1 + c
\]

\[
b = 4 + c
\]

\[
d = a + b
\]

\[
a = a + b
\]

\[
b = a + c
\]

EXIT

OUT[n1] = \{
\}

OUT[n2] = \{ a → 1 \}

OUT[n3] = \{ a → 1, b → 2 \}

OUT[n4] = \{ a → 1, b → 2, c → 3 \}

OUT[n6] = \{ a → 4, b → 2, c → 3 \}

OUT[n7] = \{ a → 4, b → 7, c → 3 \}

OUT[n8] = \{ a → 4, b → 7, c → 3, d → 11 \}

OUT[n9] = \{ a → 5, b → 2, c → 3 \}

OUT[n10] = \{ a → 5, b → 6, c → 3 \}

IN[n11] = \{ a → any, b → any, c → 3 \}

OUT[n11] = \{ a → any, b → any, c → 3 \}

OUT[n12] = \{ a → any, b → any, c → 3 \}

Note: at the exit node a and b are compile-time constants, but this analysis is not powerful enough to infer this