Data-Flow Analysis

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 $\text{OUT}[n]$
\[
\begin{align*}
\text{ENTRY} & \quad \text{n1} \\
\text{d1} \quad i & = m - 1 \quad \text{n2} \\
\text{d2} \quad j & = n \quad \text{n3} \\
\text{d3} \quad a & = u_1 \quad \text{n4} \\
\text{d4} \quad i & = i + 1 \quad \text{n5} \\
\text{d5} \quad j & = j - 1 \quad \text{n6} \\
\text{d6} \quad a & = u_2 \quad \text{n8} \\
\text{d7} \quad i & = u_3 \quad \text{n9} \\
\text{d8} & \quad \text{n10} \\
\text{EXIT} & \quad \text{n11}
\end{align*}
\]

Examples of relationships:

\[
\begin{align*}
\text{IN}[n2] & = \text{OUT}[n1] \\
\text{IN}[n5] & = \text{OUT}[n4] \cup \text{OUT}[n10] \\
\text{OUT}[n7] & = \text{IN}[n7] \\
\text{OUT}[n9] & = (\text{IN}[n9] - \{d_1, d_4, d_7\}) \cup \{d_7\}
\end{align*}
\]
Formulation as a System of Equations

• For each CFG node $n$

\[
\text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} \text{OUT}[m] \quad \text{OUT}[\text{ENTRY}] = \emptyset
\]

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

– $\text{GEN}[n]$ is a singleton set containing the definition $d$ at $n$
– $\text{KILL}[n]$ is the set of all defs of the variable written by $d$

• It can be proven that the “smallest” sets $\text{IN}[n]$ and $\text{OUT}[n]$ that satisfy this system are exactly the solution for the Reaching Definitions problem
  – To ponder: how do we know that this system has any solutions? how about a unique smallest one?
Iteratively Solving the System of Equations

OUT\([n]\) = \(\emptyset\) for each CFG node \(n\)

\(change = true\)

While (\(change\))

1. For each \(n\) other than ENTRY and EXIT
   \(OUT_{\text{old}}[n] = OUT[n]\)

2. For each \(n\) other than ENTRY
   \(IN[n] = \text{union of } OUT[m] \text{ for all predecessors } m \text{ of } n\)

3. For each \(n\) other than ENTRY and EXIT
   \(OUT[n] = (IN[n] − KILL[n]) \cup GEN[n]\)

4. \(change = false\)

5. For each \(n\) other than ENTRY and EXIT
   If (\(OUT_{\text{old}}[n] \neq OUT[n]\)) \(change = true\)
Worklist Algorithm

\[ \text{IN}[n] = \emptyset \text{ for all } n \]

Put the successor of ENTRY on worklist

While (worklist is not empty)

1. Remove any CFG node \( m \) from the worklist
2. \( \text{OUT}[m] = (\text{IN}[m] - \text{KILL}[m]) \cup \text{GEN}[m] \)
3. For each successor \( n \) of \( m \)
   
   \[ \text{old} = \text{IN}[n] \]
   
   \[ \text{IN}[n] = \text{IN}[n] \cup \text{OUT}[m] \]
   
   If (\( \text{old} \neq \text{IN}[n] \)) add \( n \) to worklist

This is “chaotic” iteration

• The order of adding-to/removing-from the worklist is unspecified
  • e.g., could use stack, queue, set, etc.
• The order of processing of successor nodes is unspecified

Regardless of order, the resulting solution is always the same
A Simpler Formulation

• In practice, an algorithm will only compute $\text{IN}[n]$

\[
\text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} (\text{IN}[m] \setminus \text{KILL}[m]) \cup \text{GEN}[m]
\]

– Ignore predecessor $m$ if it is ENTRY

• Worklist algorithm
  – $\text{IN}[n] = \emptyset$ for all $n$
  – Put the successor of ENTRY on the worklist
  – While the worklist is not empty, remove any $m$ from the worklist; for each successors $n$ of $m$, do
    • $old = \text{IN}[n]$
    • $\text{IN}[n] = \text{IN}[n] \cup (\text{IN}[m] \setminus \text{KILL}[m]) \cup \text{GEN}[m]$
    • If ($old \neq \text{IN}[n]$) add $n$ to worklist
A Few Notes

• We sometimes write

\[
\text{IN}[n] = \bigcup_{m \in \text{Predecessors}(n)} (\text{IN}[m] \cap \text{PRES}[m]) \cup \text{GEN}[m]
\]

• PRES\([n]\): the set of all definitions “preserved” (i.e., not killed) by \(n\); the complement of KILL\([n]\)

• Efficient implementation: bitvectors
  – Sets are presented by bitvectors; set intersection is bitwise AND; set union is bitwise OR
  – GEN\([n]\) and PRES\([n]\) are computed once, at the very beginning of the analysis
  – IN\([n]\) are computed iteratively, using a worklist
Reaching Definitions and Basic Blocks

• For space/time savings, we can solve the problem for basic blocks (i.e., CFG nodes are basic blocks)
  – Program points are before/after basic blocks
  – $IN[n]$ is still the union of $OUT[m]$ for predecessors $m$
  – $OUT[n]$ is still $(IN[n] - KILL[n]) \cup GEN[n]$

• $KILL[n] = KILL[s_1] \cup KILL[s_2] \cup \ldots \cup KILL[s_k]$
  – $s_1, s_2, \ldots, s_k$ are the statements in the basic blocks

• $GEN[n] = GEN[s_k] \cup (GEN[s_{k-1}] - KILL[s_k]) \cup (GEN[s_{k-2}] - KILL[s_{k-1}] - KILL[s_k]) \cup \ldots \cup (GEN[s_1] - KILL[s_2] - KILL[s_3] - \ldots - KILL[s_k])$
  – $GEN[n]$ contains any definition in the block that is downward-exposed (i.e., not killed by a subsequent definition in the block)
KILL[n2] = \{ d1, d2, d3, d4, d5, d6, d7 \}  
GEN[n2] = \{ d1, d2, d3 \}  
KILL[n3] = \{ d1, d2, d4, d5, d7 \}  
GEN[n3] = \{ d4, d5 \}  
KILL[n4] = \{ d3, d6 \}  
GEN[n4] = \{ d6 \}  
KILL[n5] = \{ d1, d4, d7 \}  
GEN[n5] = \{ d7 \}  

IN[n2] = \{ \}  
OUT[n2] = \{ d1, d2, d3 \}  
IN[n3] = \{ d1, d2, d3, d5, d6, d7 \}  
OUT[n3] = \{ d3, d4, d5, d6 \}  
IN[n4] = \{ d3, d4, d5, d6 \}  
OUT[n4] = \{ d4, d5, d6 \}  
IN[n5] = \{ d3, d4, d5, d6 \}  
OUT[n5] = \{ d3, d5, d6, d7 \}
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?
  – For basic blocks: need all upward-exposed uses (use of variable does not have preceding def in the same basic block)

• 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
Upward exposed uses:
USES[n2] = { m@d1, n@d2, u1@d3 }
USES[n3] = { i@d4, j@d5, a@c1 }
USES[n4] = { u2@d6 }
USES[n5] = { u3@d7, j@c2, a@c2 }

Reaching definitions:
IN[n3]  = { d1, d2, d3, d5, d6, d7 }
IN[n4]  = { d3, d4, d5, d6 }
IN[n5]  = { d3, d4, d5, d6 }

Def-use chains across basic blocks:
DU[d1] = upward exposed uses of variable i in all basic blocks n such that d1 ∈ IN[n] = { i@d4 }
DU[d2] = { j@d5 }
DU[d3] = { a@c1, a@c2 }
DU[d4] = { }
DU[d5] = { j@d5, j@c2 }
DU[d6] = { a@c1, a@c2 }
DU[d7] = { i@d4 }

Def-use chains inside basic blocks:
DU[d4] = { i@c1 }

Use-def chains:
UD[m@d1]= { }
UD[n@d2]= { }
UD[u1@d3]= { }
UD[i@d4]= { d1,d7 }
UD[j@d5]= { d2,d5 }
UD[i@c1]= { d4 }
UD[a@c1]= { d3,d6 }
UD[u2@d6]= { }
UD[u3@d7]= { }
UD[j@c2]= { d5 }
UD[a@c2]= { d3,d6 }
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 $n$
Examples of relationships:

OUT[n1] = IN[n2]

OUT[n7] = IN[n8] ∪ IN[n9]

IN[n10] = OUT[n10]

IN[n2] = (OUT[n2] − {i}) ∪ {m}
Formulation as a System of Equations

• For each CFG node $n$

\[
\text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} \text{IN}[m]
\]

\[
\text{IN}[n] = (\text{OUT}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

– $\text{GEN}[n]$ is the set of all variables that are read by $n$
– $\text{KILL}[n]$ is a singleton set containing the variable that is written by $n$ (even if this variable is live immediately after $n$, it is not live immediately before $n$)

• The smallest sets $\text{IN}[n]$ and $\text{OUT}[n]$ that satisfy this system are exactly the solution for the Live Variables problem
Iteratively Solving the System of Equations

\[ \text{IN}[n] = \emptyset \text{ for each CFG node } n \]

\[ \textit{change} = \text{true} \]

While (\textit{change})

1. For each \( n \) other than ENTRY and EXIT
   \[ \text{IN}_{\text{old}}[n] = \text{IN}[n] \]
2. For each \( n \) other than EXIT
   \[ \text{OUT}[n] = \text{union of IN}[m] \text{ for all successors } m \text{ of } n \]
3. For each \( n \) other than ENTRY and EXIT
   \[ \text{IN}[n] = ( \text{OUT}[n] \setminus \text{KILL}[n] ) \cup \text{GEN}[n] \]
4. \textit{change} = false
5. For each \( n \) other than ENTRY and EXIT
   If (\( \text{IN}_{\text{old}}[n] \neq \text{IN}[n] \)) \textit{change} = true
Worklist Algorithm

OUT\[n\] = \emptyset for all \( n \)

Put the predecessors of EXIT on worklist

While (worklist is not empty)

1. Remove any CFG node \( m \) from the worklist
2. IN\([m]\) = (OUT\([m]\) – KILL\([m]\)) \cup GEN\([m]\)
3. For each predecessor \( n \) of \( m \)
   
   \( old = OUT\[n\] \)
   
   OUT\[n\] = OUT\[n\] \cup IN\([m]\)
   
   If \( old \neq OUT\[n\] \) add \( n \) to worklist

As with the worklist algorithm for Reaching Definitions, this is chaotic iteration. But, regardless of order, the resulting solution is always the same.
A Simpler Formulation

• In practice, an algorithm will only compute OUT[n]

\[ \text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m] \]

  – Ignore successor \( m \) if it is EXIT

• Worklist algorithm
  – OUT[n] = \( \emptyset \) for all \( n \)
  – Put the predecessors of EXIT on the worklist
  – While the worklist is not empty, remove any \( m \) from the worklist; for each predecessor \( n \) of \( m \), do
    • old = OUT[n]
    • OUT[n] = OUT[n] \( \cup \) (OUT[m] – KILL[m]) \( \cup \) GEN[m]
    • If (old \( \neq \) OUT[n]) add \( n \) to worklist
A Few Notes

• We sometimes write

$$\text{OUT}[n] = \bigcup_{m \in \text{Successors}(n)} (\text{OUT}[m] \cap \text{PRES}[m]) \cup \text{GEN}[m]$$

- PRES\([n]\): the set of all variables “preserved” (i.e., not written) by \(n\); the complement of KILL\([n]\)
- Efficient implementation: bitvectors

• Comparison with Reaching Definitions
  - Reaching Definitions is a forward data-flow problem and Live Variables is a backward data-flow problem
  - Other than that, they are basically the same

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

21
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 $\text{if (v)}$ ...
  – Program understanding, restructuring, verification, testing, etc.
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 \( x \in \text{Values} \)
  - \( \text{Values} = \text{all possible constant values} \cup \{ nac, \text{undef} \} \)

- Special “value” \( nac \) (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., the value comes from some \( nac \) variable

- Special “value” \( \text{undef} \) (undefined): used temporarily during the analysis
  - Means “we have no information about \( v \) yet”
Formulation as a System of Equations

• OUT[ENTRY] = a map which maps each v to undef

• For any other CFG node n
  – IN[n] = Merge(OUT[m]) for all predecessors m of n
  – OUT[n] = 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 $c_1 = undef$, the result is $c_2$
  – Else if $c_2 = undef$, the result is $c_1$
  – Else if $c_1 = nac$ or $c_2 = nac$, the result it nac
  – Else if $c_1 \neq c_2$, the result is nac
  – Else the result is $c_1$ (in this case we know that $c_1 = c_2$)
Formulation as a System of Equations

• **Updating** a map at an assignment $v = \ldots$
  – If the statement is not an assignment, $\text{OUT}[n] = \text{IN}[n]$  
• The map does not change for any $w \neq v$
• If we have $v = c$, where $c$ is a constant: in $\text{OUT}[n]$, $v$ is now mapped to $c$
• If we have $v = p + q$ (or similar binary operators) and $\text{IN}[n]$ maps $p$ and $q$ to $c_1$ and $c_2$ respectively  
  – If both $c_1$ and $c_2$ are constants: result is $c_1 + c_2$
  – Else if either $c_1$ or $c_2$ is $\text{nac}$: result is $\text{nac}$
  – Else: result is $\text{undef}$
ENTRY

\[ a = 1 \]

\[ b = 2 \]

\[ c = a + b \]

\[ a = 1 + c \]

\[ b = 4 + c \]

\[ d = a + b \]

\[ a = a + b \]

\[ b = a + c \]

EXIT

OUT\[n1\] = \{a → \text{undefined}, b → \text{undefined}, c → \text{undefined}, d → \text{undefined} \}

OUT\[n2\] = \{a → 1, b → \text{undefined}, c → \text{undefined}, d → \text{undefined} \}

OUT\[n3\] = \{a → 1, b → 2, c → \text{undefined}, d → \text{undefined} \}

OUT\[n4\] = \{a → 1, b → 2, c → 3, d → \text{undefined} \}

OUT\[n6\] = \{a → 4, b → 2, c → 3, d → \text{undefined} \}

OUT\[n7\] = \{a → 4, b → 7, c → 3, d → \text{undefined} \}

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

OUT\[n9\] = \{a → 5, b → 2, c → 3, d → \text{undefined} \}

OUT\[n10\] = \{a → 5, b → 6, c → 3, d → \text{undefined} \}

IN\[n11\] = \{a → \text{nac}, b → \text{nac}, c → 3, d → 11 \}

OUT\[n11\] = \{a → \text{nac}, b → \text{nac}, c → 3, d → 11 \}

OUT\[n12\] = \{a → \text{nac}, b → \text{nac}, c → 3, d → 11 \}

Note: in reality, d could be uninitialized at n11 and n12 (see Section 9.4.6 for a good discussion on this issue)
Example: Interprocedural Analysis

- CFG = procedure-level CFGs, plus (call,entry) and (exit,return) edges

```c
void P1() {
    ...
    P2();
    ...
}
```
Valid Paths

Valid path: every (exit, return) matches the corresponding (call, entry)
Design of Interprocedural Analysis

- **Intraprocedural** analysis: considers all CFG paths and computes a static over-approximation of their effects

- **Interprocedural** analysis: should consider all valid CFG paths
  - Option 1: do not distinguish between valid and invalid paths
    - **Calling-context-insensitive** analysis: does not keep track of the calling context of a procedure
  - Option 2: **calling-context-sensitive** analysis
    - Keeps tracks of calling context, and avoids some of the invalid paths