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 $n$
  – $IN[n]$ is the set of definitions that reach the program point immediately before $n$
  – $OUT[n]$ is the set of definitions that reach the program point immediately after $n$
  – Reaching definitions analysis computes $IN[n]$ and $OUT[n]$
\begin{itemize}
\item \textbf{ENTRY} \hspace{1cm} n1
\item i = m - 1 \hspace{1cm} n2
\item j = n \hspace{1cm} n3
\item a = u1 \hspace{1cm} n4
\item i = i + 1 \hspace{1cm} n5
\item j = j - 1 \hspace{1cm} n6
\item \textbf{if (\ldots)} \hspace{1cm} n7
\item a = u2 \hspace{1cm} n8
\item i = u3 \hspace{1cm} n9
\item \textbf{if (\ldots)} \hspace{1cm} n10
\item \textbf{EXIT} \hspace{1cm} n11
\end{itemize}

\begin{align*}
\text{OUT}[n1] &= \{ \} \\
\text{IN}[n2] &= \{ \} \\
\text{OUT}[n2] &= \{ \text{d1} \} \\
\text{IN}[n3] &= \{ \text{d1} \} \\
\text{OUT}[n3] &= \{ \text{d1, d2} \} \\
\text{IN}[n4] &= \{ \text{d1, d2} \} \\
\text{OUT}[n4] &= \{ \text{d1, d2, d3} \} \\
\text{IN}[n5] &= \{ \text{d1, d2, d3, d5, d6, d7} \} \\
\text{OUT}[n5] &= \{ \text{d2, d3, d4, d5, d6} \} \\
\text{IN}[n6] &= \{ \text{d2, d3, d4, d5, d6} \} \\
\text{OUT}[n6] &= \{ \text{d3, d4, d5, d6} \} \\
\text{IN}[n7] &= \{ \text{d3, d4, d5, d6} \} \\
\text{OUT}[n7] &= \{ \text{d3, d4, d5, d6} \} \\
\text{IN}[n8] &= \{ \text{d3, d4, d5, d6} \} \\
\text{OUT}[n8] &= \{ \text{d4, d5, d6} \} \\
\text{IN}[n9] &= \{ \text{d3, d4, d5, d6} \} \\
\text{OUT}[n9] &= \{ \text{d3, d5, d6, d7} \} \\
\text{IN}[n10] &= \{ \text{d3, d5, d6, d7} \} \\
\text{OUT}[n10] &= \{ \text{d3, d5, d6, d7} \} \\
\text{IN}[n11] &= \{ \text{d3, d5, d6, d7} \}
\end{align*}

\textbf{Examples of relationships:}
\begin{itemize}
\item \text{IN}[n2] = \text{OUT}[n1]
\item \text{IN}[n5] = \text{OUT}[n4] \cup \text{OUT}[n10]
\item \text{OUT}[n7] = \text{IN}[n7]
\item \text{OUT}[n9] = (\text{IN}[n9] - \{d1, d4, d7\}) \cup \{d7\}
\end{itemize}
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
  – We will ignore: how do we know that this system has any solutions? how about a unique smallest one?
Iteratively Solving the System of Equations

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

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

While (\text{change})

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

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

– 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
    • $\text{old} = \text{IN}[n]$
    • $\text{IN}[n] = \text{IN}[n] \cup (\text{IN}[m] - \text{KILL}[m]) \cup \text{GEN}[m]$
    • If ($\text{old} \neq \text{IN}[n]$) 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 ($IN[n] – KILL[n]$) ∪ GEN[$n$]
- $KILL[n] = KILL[s_1] ∪ KILL[s_2] ∪ ... ∪ KILL[s_k]$
  - $s_1, s_2, ..., s_k$ are the statements in the basic blocks
- $GEN[n] = GEN[s_k] ∪ (GEN[s_{k-1}] – KILL[s_k]) ∪ (GEN[s_{k-2}] – KILL[s_{k-1}] – KILL[s_k]) ∪ ... ∪ (GEN[s_1] – KILL[s_2] – KILL[s_3] – ... – KILL[s_k])$
  - GEN[$n$] contains any definition in the block that is downward-exposed (i.e., not killed by a subsequent definition in the block)
\[
\begin{align*}
\text{ENTRY} & \quad \text{n1} \\
\text{d1} & \quad i = m-1 \\
\text{d2} & \quad j = n \\
\text{d3} & \quad a = u1 \\
\text{d4} & \quad i = i + 1 \\
\text{d5} & \quad j = j - 1 \\
\text{d6} & \quad a = u2 \\
\text{d7} & \quad i = u3 \\
\text{EXIT} & \quad \text{n6}
\end{align*}
\]

\[
\begin{align*}
\text{KILL}[n2] & = \{ d1, d2, d3, d4, d5, d6, d7 \} \\
\text{GEN}[n2] & = \{ d1, d2, d3 \} \\
\text{KILL}[n3] & = \{ d1, d2, d4, d5, d7 \} \\
\text{GEN}[n3] & = \{ d4, d5 \} \\
\text{KILL}[n4] & = \{ d3, d6 \} \\
\text{GEN}[n4] & = \{ d6 \} \\
\text{KILL}[n5] & = \{ d1, d4, d7 \} \\
\text{GEN}[n5] & = \{ d7 \}
\end{align*}
\]

\[
\begin{align*}
\text{IN}[n2] & = \{ \} \\
\text{OUT}[n2] & = \{ d1, d2, d3 \} \\
\text{IN}[n3] & = \{ d1, d2, d3, d5, d6, d7 \} \\
\text{OUT}[n3] & = \{ d3, d4, d5, d6 \} \\
\text{IN}[n4] & = \{ d3, d4, d5, d6 \} \\
\text{OUT}[n4] & = \{ d4, d5, d6 \} \\
\text{IN}[n5] & = \{ d3, d4, d5, d6 \} \\
\text{OUT}[n5] & = \{ d3, d5, d6, d7 \}
\end{align*}
\]
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 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 \( n \)
\[ \text{ENTRY} \]
\[ i = m - 1 \]
\[ j = n \]
\[ a = u_1 \]
\[ i = i + 1 \]
\[ j = j - 1 \]

\[ \begin{align*}
\text{OUT}[n_1] & = \{ m, n, u_1, u_2, u_3 \} \\
\text{IN}[n_2] & = \{ m, n, u_1, u_2, u_3 \} \\
\text{OUT}[n_2] & = \{ n, u_1, i, u_2, u_3 \} \\
\text{IN}[n_3] & = \{ n, u_1, i, u_2, u_3 \} \\
\text{OUT}[n_3] & = \{ u_1, i, j, u_2, u_3 \} \\
\text{IN}[n_4] & = \{ u_1, i, j, u_2, u_3 \} \\
\text{OUT}[n_4] & = \{ i, j, u_2, u_3 \} \\
\text{IN}[n_5] & = \{ i, j, u_2, u_3 \} \\
\text{OUT}[n_5] & = \{ j, u_2, u_3 \} \\
\text{IN}[n_6] & = \{ j, u_2, u_3 \} \\
\text{OUT}[n_6] & = \{ u_2, u_3, j \} \\
\text{IN}[n_7] & = \{ u_2, u_3, j \} \\
\text{OUT}[n_7] & = \{ u_2, u_3, j \} \\
\text{IN}[n_8] & = \{ u_2, u_3, j \} \\
\text{OUT}[n_8] & = \{ u_3, j, u_2 \} \\
\text{IN}[n_9] & = \{ u_3, j, u_2 \} \\
\text{OUT}[n_9] & = \{ i, j, u_2, u_3 \} \\
\text{IN}[n_10] & = \{ i, j, u_2, u_3 \} \\
\text{OUT}[n_10] & = \{ i, j, u_2, u_3 \} \\
\text{IN}[n_11] & = \{ \} \\
\end{align*} \]

\[ \text{EXIT} \]

Examples of relationships:

\[ \text{OUT}[n_1] = \text{IN}[n_2] \]
\[ \text{OUT}[n_7] = \text{IN}[n_8] \cup \text{IN}[n_9] \]
\[ \text{IN}[n_10] = \text{OUT}[n_10] \]
\[ \text{IN}[n_2] = (\text{OUT}[n_2] - \{i\}) \cup \{m\} \]
Formulation as a System of Equations

• For each CFG node $n$

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

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

— GEN$[n]$ is the set of all variables that are read by $n$
— 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 IN$[n]$ and OUT$[n]$ that satisfy this system are exactly the solution for the Live Variables problem
Iteratively Solving the System of Equations

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

\textit{change} = \text{true}

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

$\text{OUT}[n] = \emptyset$ 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. $\text{IN}[m] = (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m]$
  3. For each predecessor $n$ of $m$
     - $\text{old} = \text{OUT}[n]$
     - $\text{OUT}[n] = \text{OUT}[n] \cup \text{IN}[m]$
     - If ($\text{old} \neq \text{OUT}[n]$) 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 $\text{OUT}[n]$

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

– Ignore successor $m$ if it is EXIT

• Worklist algorithm
  – $\text{OUT}[n] = \emptyset$ 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
    • $old = \text{OUT}[n]$
    • $\text{OUT}[n] = \text{OUT}[n] \cup (\text{OUT}[m] - \text{KILL}[m]) \cup \text{GEN}[m]$
    • If ($old \neq \text{OUT}[n]$) 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 \textbf{forward} data-flow problem
    and Live Variables is a \textbf{backward} data-flow problem
  – Other than that, they are basically the same

• Uses of Live Variables
  – Dead code elimination: e.g., when \( x = y + z \)
  – 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 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 if ($v$) ...

  – Program understanding, restructuring, verification, testing, etc.
Basic Ideas

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

• Special “value” \( \text{nac} \) (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 \( \text{nac} \) 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 undefined

• 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 $c_1 = undefined$, the result is $c_2$
  – Else if $c_2 = undefined$, 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 $nac$: result is $nac$
  - Else: result is $undef$
ENTRY

a = 1

b = 2

c = a+b

if (...) n5

a = 1+c n6

b = 3+c n7

d = a+b n8

a = a+b n11

b = a+c n12

EXIT n13

OUT\[n1\] = \{a \rightarrow \textit{undef}, b \rightarrow \textit{undef}, c \rightarrow \textit{undef}, d \rightarrow \textit{undef} \}
OUT\[n2\] = \{a \rightarrow 1, b \rightarrow \textit{undef}, c \rightarrow \textit{undef}, d \rightarrow \textit{undef} \}
OUT\[n3\] = \{a \rightarrow 1, b \rightarrow 2, c \rightarrow \textit{undef}, d \rightarrow \textit{undef} \}
OUT\[n4\] = \{a \rightarrow 1, b \rightarrow 2, c \rightarrow 3, d \rightarrow \textit{undef} \}
OUT\[n6\] = \{a \rightarrow 4, b \rightarrow 2, c \rightarrow 3, d \rightarrow \textit{undef} \}
OUT\[n7\] = \{a \rightarrow 4, b \rightarrow 7, c \rightarrow 3, d \rightarrow \textit{undef} \}
OUT\[n8\] = \{a \rightarrow 4, b \rightarrow 7, c \rightarrow 3, d \rightarrow 11 \}
OUT\[n9\] = \{a \rightarrow 5, b \rightarrow 2, c \rightarrow 3, d \rightarrow \textit{undef} \}
OUT\[n10\] = \{a \rightarrow 5, b \rightarrow 6, c \rightarrow 3, d \rightarrow \textit{undef} \}
IN\[n11\] = \{a \rightarrow \textit{nac}, b \rightarrow \textit{nac}, c \rightarrow 3, d \rightarrow 11 \}
OUT\[n11\] = \{a \rightarrow \textit{nac}, b \rightarrow \textit{nac}, c \rightarrow 3, d \rightarrow 11 \}
OUT\[n12\] = \{a \rightarrow \textit{nac}, b \rightarrow \textit{nac}, c \rightarrow 3, d \rightarrow 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();
    ...
}
```

CFG for P1

CFG for P2
Valid Paths

Valid path: every (exit, return) matches the corresponding (call, entry)
The blue path is not valid
Design of **Interprocedural Analysis**

- **Intraprocedural** analysis: separately considers the CFG for each procedure; makes conservative assumptions about any calls in the CFG.

- **Interprocedural** analysis: considers all CFGs together; should consider **all valid CFG paths**
  - Option 1: do not distinguish between valid/invalid
    - **Calling-context-insensitive** analysis: does not keep track of the calling context of a procedure
    - **Calling context example**: the CFG call node that made the call (called “call site”)
  - Option 2: **calling-context-sensitive** analysis
    - Keeps tracks of calling context, and avoids some of the invalid paths