CS738: Advanced Compiler Optimizations

Data Flow Analysis

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs738
Department of CSE, IIT Kanpur
Agenda

- Static analysis and compile-time optimizations
Agenda

- Static analysis and compile-time optimizations
- For the next few lectures
Agenda

- Static analysis and compile-time optimizations
- For the next few lectures
- *Intraprocedural* Data Flow Analysis
Agenda

- Static analysis and compile-time optimizations
- For the next few lectures
- *Intraprocedural* Data Flow Analysis
  - Classical Examples
Agenda

- Static analysis and compile-time optimizations
- For the next few lectures
- *Intraprocedural* Data Flow Analysis
  - Classical Examples
  - Components
Assumptions

- Intraprocedural: Restricted to a single function
Assumptions

- Intraprocedural: Restricted to a single function
- Input in 3-address format
Assumptions

- Intraprocedural: Restricted to a single function
- Input in 3-address format
- Unless otherwise specified
3-address Code Format

- Assignments
3-address Code Format

- Assignments
  \[ x = y \text{ op } z \]
Assignments

\[ x = y \text{ op } z \]
\[ x = \text{ op } y \]
3-address Code Format

- Assignments
  - $x = y \text{ op } z$
  - $x = \text{ op } y$
  - $x = y$
3-address Code Format

- Assignments
  - \( x = y \text{ op } z \)
  - \( x = \text{ op } y \)
  - \( x = y \)

- Jump/control transfer
3-address Code Format

- Assignments
  - $x = y \text{ op } z$
  - $x = \text{ op } y$
  - $x = y$

- Jump/control transfer
  - goto L
3-address Code Format

- Assignments
  - $x = y \ op \ z$
  - $x = \ op \ y$
  - $x = y$

- Jump/control transfer
  - goto L
  - if $x \ relop \ y$ goto L
3-address Code Format

- **Assignments**
  - \( x = y \text{ op } z \)
  - \( x = \text{ op } y \)
  - \( x = y \)

- **Jump/control transfer**
  - \( \text{goto L} \)
  - \( \text{if } x \text{ relop } y \text{ goto L} \)

- **Statements can have label(s)**
3-address Code Format

▶ Assignments
  - \( x = y \; \text{op} \; z \)
  - \( x = \text{op} \; y \)
  - \( x = y \)

▶ Jump/control transfer
  - \( \text{goto} \; L \)
  - \( \text{if} \; x \; \text{relop} \; y \; \text{goto} \; L \)

▶ Statements can have label(s)
  - \( L : \ldots \)
3-address Code Format

- **Assignments**
  - \( x = y \text{ op } z \)
  - \( x = \text{ op } y \)
  - \( x = y \)

- **Jump/control transfer**
  - goto L
  - if \( x \text{ relop } y \) goto L

- **Statements can have label(s)**
  - L: . . .

- **Arrays, Pointers and Functions to be added later when needed**
Data Flow Analysis

- Class of techniques to derive information about flow of data
Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
  - whether two identical expressions evaluate to same value
Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
  - whether two identical expressions evaluate to same value
  - used in common subexpression elimination
Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
  - whether two identical expressions evaluate to same value
    - used in common subexpression elimination
  - whether the result of an assignment is used later
Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
  - whether two identical expressions evaluate to same value
    - used in common subexpression elimination
  - whether the result of an assignment is used later
    - used by dead code elimination
Data Flow Abstraction

- Basic Blocks (BB)
Data Flow Abstraction

- Basic Blocks (BB)
  - sequence of 3-address code stmts
Data Flow Abstraction

- Basic Blocks (BB)
  - sequence of 3-address code stmts
  - single entry at the first statement
Data Flow Abstraction

- Basic Blocks (BB)
  - sequence of 3-address code stmts
  - single entry at the first statement
  - single exit at the last statement
Data Flow Abstraction

▸ Basic Blocks (BB)
  ▸ sequence of 3-address code stmts
  ▸ single entry at the first statement
  ▸ single exit at the last statement
  ▸ Typically we use “maximal” basic block (maximal sequence of such instructions)
Identifying Basic Blocks

- **Leader**: The first statement of a basic block
Identifying Basic Blocks

- *Leader*: The first statement of a basic block
  - The first instruction of the program (procedure)
Identifying Basic Blocks

- **Leader**: The first statement of a basic block
  - The first instruction of the program (procedure)
  - Target of a branch (conditional and unconditional goto)
Identifying Basic Blocks

- **Leader**: The first statement of a basic block
  - The first instruction of the program (procedure)
  - Target of a branch (conditional and unconditional goto)
  - Instruction immediately following a branch
Special Basic Blocks

- Two special BBs are added to simplify the analysis
Special Basic Blocks

- Two special BBs are added to simplify the analysis
  - empty (?) blocks!
Special Basic Blocks

- Two special BBs are added to simplify the analysis
  - empty (?) blocks!
- Entry: The first block to be executed for the procedure analyzed
Two special BBs are added to simplify the analysis

- empty (?) blocks!

- *Entry*: The first block to be executed for the procedure analyzed

- *Exit*: The last block to be executed
Data Flow Abstraction

- Control Flow Graph (CFG)
Data Flow Abstraction

- Control Flow Graph (CFG)
- A rooted directed graph $G = (N, E)$
Data Flow Abstraction

- Control Flow Graph (CFG)
- A rooted directed graph $G = (N, E)$
- $N =$ set of BBs
Data Flow Abstraction

- Control Flow Graph (CFG)
- A rooted directed graph $G = (N, E)$
- $N =$ set of BBs
  - including $Entry$, $Exit$
Data Flow Abstraction

- Control Flow Graph (CFG)
- A rooted directed graph $G = (N, E)$
- $N =$ set of BBs
  - including $Entry$, $Exit$
- $E =$ set of edges
Edge $B_1 \rightarrow B_2 \in E$ if control can transfer from $B_1$ to $B_2$
CFG Edges

- Edge $B_1 \rightarrow B_2 \in E$ if control can transfer from $B_1$ to $B_2$
- Fall through
CFG Edges

- Edge $B_1 \rightarrow B_2 \in E$ if control can transfer from $B_1$ to $B_2$
  - Fall through
  - Through jump (goto)
CFG Edges

- Edge $B_1 \rightarrow B_2 \in E$ if control can transfer from $B_1$ to $B_2$
  - Fall through
  - Through jump (goto)
  - Edge from Entry to (all?) real first BB(s)
Edge \( B_1 \rightarrow B_2 \in E \) if control can transfer from \( B_1 \) to \( B_2 \)
- Fall through
- Through jump (goto)
- Edge from \( Entry \) to (all?) real first BB(s)
- Edge to \( Exit \) from all last BBs
CFG Edges

- Edge $B_1 \rightarrow B_2 \in E$ if control can transfer from $B_1$ to $B_2$
  - Fall through
  - Through jump (goto)
  - Edge from $Entry$ to (all?) real first BB(s)
  - Edge to $Exit$ from all last BBs
    - BBs containing return
CFG Edges

- Edge $B_1 \to B_2 \in E$ if control can transfer from $B_1$ to $B_2$
  - Fall through
  - Through jump (goto)
  - Edge from *Entry* to (all?) real first BB(s)
  - Edge to *Exit* from all last BBs
    - BBs containing return
    - Last real BB
Data Flow Abstraction: Control Flow Graph

- Graph representation of paths that program may exercise during execution
Data Flow Abstraction: Control Flow Graph

- Graph representation of paths that program may exercise during execution
- Typically one graph per procedure
Data Flow Abstraction: Control Flow Graph

- Graph representation of paths that program may exercise during execution
- Typically one graph per procedure
- Graphs for separate procedures have to be combined/connected for interprocedural analysis
Data Flow Abstraction: Control Flow Graph

- Graph representation of paths that program may exercise during execution
- Typically one graph per procedure
- Graphs for separate procedures have to be combined/connected for interprocedural analysis
  - Later!
Data Flow Abstraction: Control Flow Graph

- Graph representation of paths that program may exercise during execution
- Typically one graph per procedure
- Graphs for separate procedures have to be combined/connected for interprocedural analysis
  - Later!
  - Single procedure, single flow graph for now.
Data Flow Abstraction: Program Points

- Input state/Output state for Stmt
Data Flow Abstraction: Program Points

- Input state/Output state for Stmt
  - Program point before/after a stmt
Data Flow Abstraction: Program Points

- Input state/Output state for Stmt
  - Program point before/after a stmt
  - Denoted IN[s] and OUT[s]
Data Flow Abstraction: Program Points

- Input state/Output state for Stmt
  - Program point before/after a stmt
  - Denoted IN[s] and OUT[s]
  - Within a basic block:
Data Flow Abstraction: Program Points

- Input state/Output state for Stmt
  - Program point before/after a stmt
  - Denoted IN[s] and OUT[s]
  - Within a basic block:
    - Program point after a stmt is same as the program point before the next stmt
Data Flow Abstraction: Program Points

- Input state/Output state for BBs
Data Flow Abstraction: Program Points

- Input state/Output state for BBs
  - Program point before/after a bb
Data Flow Abstraction: Program Points

- Input state/Output state for BBs
  - Program point before/after a bb
  - Denoted IN[B] and OUT[B]
Data Flow Abstraction: Program Points

- Input state/Output state for BBs
  - Program point before/after a bb
  - Denoted IN[B] and OUT[B]
  - For $B_1$ and $B_2$: 
Data Flow Abstraction: Program Points

- Input state/Output state for BBs
  - Program point before/after a bb
  - Denoted IN[B] and OUT[B]
  - For $B_1$ and $B_2$:
    - if there is an edge from $B_1$ to $B_2$ in CFG, then the program point after the last stmt of $B_1$ may be followed immediately by the program point before the first stmt of $B_2$. 
Data Flow Abstraction: Execution Paths

- An execution path is of the form

\[ p_1, p_2, p_3, \ldots, p_n \]
Data Flow Abstraction: Execution Paths

- An execution path is of the form

\[ p_1, p_2, p_3, \ldots, p_n \]

where \( p_i \rightarrow p_{i+1} \) are adjacent program points in the CFG.
Data Flow Abstraction: Execution Paths

- An execution path is of the form

  \[ p_1, p_2, p_3, \ldots, p_n \]

  where \( p_i \rightarrow p_{i+1} \) are adjacent program points in the CFG.

- Infinite number of possible execution paths in practical programs.
An execution path is of the form

\[ p_1, p_2, p_3, \ldots, p_n \]

where \( p_i \rightarrow p_{i+1} \) are adjacent program points in the CFG.

Infinite number of possible execution paths in practical programs.

Paths having no finite upper bound on the length.
An execution path is of the form

\[ p_1, p_2, p_3, \ldots, p_n \]

where \( p_i \rightarrow p_{i+1} \) are adjacent program points in the CFG.

Infinite number of possible execution paths in practical programs.

Paths having no finite upper bound on the length.

Need to *summarize* the information at a program point with a finite set of facts.
Data Flow Schema

- Data flow values associated with each program point
Data Flow Schema

- Data flow values associated with each program point
  - Summarize all possible states at that point
Data Flow Schema

- Data flow values associated with each program point
  - Summarize all possible states at that point
- Domain: set of all possible data flow values
Data Flow Schema

- Data flow values associated with each program point
  - Summarize all possible states at that point
- Domain: set of all possible data flow values
- Different domains for different analyses/optimizations
Data Flow Problem

- Constraints on data flow values
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
  - Multiple solutions possible
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
  - Multiple solutions possible
  - Trivial solutions, . . . , Exact solutions
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
  - Multiple solutions possible
  - Trivial solutions, . . ., Exact solutions
- We typically compute approximate solution
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
  - Multiple solutions possible
  - Trivial solutions, . . . , Exact solutions
- We typically compute approximate solution
  - Close to the exact solution (as close as possible!)
Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
  - Multiple solutions possible
  - Trivial solutions, ..., Exact solutions
- We typically compute approximate solution
  - Close to the exact solution (as close as possible!)
  - Why not exact solution?
Data Flow Constraints: Transfer Constraints

- Transfer functions
Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
- forward functions: Compute facts after a statement $s$ from the facts available before $s$. 
Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
- forward functions: Compute facts after a statement $s$ from the facts available before $s$.
  - General form:
    $$\text{OUT}[s] = f_s(\text{IN}[s])$$
Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
- forward functions: Compute facts \textit{after} a statement \( s \) from the facts available \textit{before} \( s \).
  - General form:
    \[
    \text{OUT}[s] = f_s(\text{IN}[s])
    \]
- backward functions: Compute facts \textit{before} a statement \( s \) from the facts available \textit{after} \( s \).
Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
- forward functions: Compute facts after a statement s from the facts available before s.
  - General form:
    \[ \text{OUT}[s] = f_s(\text{IN}[s]) \]
- backward functions: Compute facts before a statement s from the facts available after s.
  - General form:
    \[ \text{IN}[s] = f_s(\text{OUT}[s]) \]
Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
  - forward functions: Compute facts after a statement $s$ from the facts available before $s$.
    - General form:
      \[ \text{OUT}[s] = f_s(\text{IN}[s]) \]
  - backward functions: Compute facts before a statement $s$ from the facts available after $s$.
    - General form:
      \[ \text{IN}[s] = f_s(\text{OUT}[s]) \]
- $f_s$ depends on the statement and the analysis
Data Flow Constraints: Control Flow Constraints

- Relationship between the data flow values of two points that are related by program execution semantics
Data Flow Constraints: Control Flow Constraints

- Relationship between the data flow values of two points that are related by program execution semantics.
- For a basic block having $n$ statements:

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \ i = 1, 2, \ldots, n - 1$$
Data Flow Constraints: Control Flow Constraints

- Relationship between the data flow values of two points that are related by program execution semantics
- For a basic block having $n$ statements:
  \[ \text{IN}[s_{i+1}] = \text{OUT}[s_i], \quad i = 1, 2, \ldots, n - 1 \]
- \text{IN}[s_1], \text{OUT}[s_n] to come later
Data Flow Constraints: Notations

- \textsc{PRED} (B): Set of predecessor BBs of block \textit{B} in CFG
Data Flow Constraints: Notations

- \( \text{PRED} (B) \): Set of predecessor BBs of block \( B \) in CFG
- \( \text{SUCC} (B) \): Set of successor BBs of block \( B \) in CFG
Data Flow Constraints: Notations

- PRED (B): Set of predecessor BBs of block B in CFG
- SUCC (B): Set of successor BBs of block B in CFG
- $f \circ g$: Composition of functions $f$ and $g$
Data Flow Constraints: Notations

- PRED (B): Set of predecessor BBs of block $B$ in CFG
- SUCC (B): Set of successor BBs of block $B$ in CFG
- $f \circ g$: Composition of functions $f$ and $g$
- $\oplus$: An abstract operator denoting some way of combining facts present in a set.
Data Flow Constraints: Basic Blocks

- Forward
Data Flow Constraints: Basic Blocks

- **Forward**
  - For $B$ consisting of $s_1, s_2, \ldots, s_n$

  $$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

  $$\text{OUT}[B] = f_B(\text{IN}[B])$$
Data Flow Constraints: Basic Blocks

- **Forward**
  - For $B$ consisting of $s_1, s_2, \ldots, s_n$

  $$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

  $$\text{OUT}[B] = f_B(\text{IN}[B])$$

- **Control flow constraints**

  $$\text{IN}[B] = \bigoplus_{P \in \text{PRED}(B)} \text{OUT}[P]$$
Data Flow Constraints: Basic Blocks

▸ **Forward**
  ▸ For $B$ consisting of $s_1, s_2, \ldots, s_n$

  $$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

  $$\text{OUT}[B] = f_B(\text{IN}[B])$$

▸ **Control flow constraints**

  $$\text{IN}[B] = \bigoplus_{P \in \text{PRED}(B)} \text{OUT}[P]$$

▸ **Backward**

  $$f_B = f_{s_1} \circ f_{s_2} \circ \ldots \circ f_{s_n}$$

  $$\text{IN}[B] = f_B(\text{OUT}[B])$$

  $$\text{OUT}[B] = \bigoplus_{S \in \text{SUCC}(B)} \text{IN}[S]$$
Data Flow Equations

Typical Equation

\[ \text{OUT}[s] = \text{IN}[s] - \text{kill}[s] \cup \text{gen}[s] \]
Data Flow Equations

- Typical Equation

\[ \text{OUT}[s] = \text{IN}[s] - \text{kill}[s] \cup \text{gen}[s] \]

\( \text{gen}(s) \): information generated
Data Flow Equations

Typical Equation

\[ \text{OUT}[s] = \text{IN}[s] - \text{kill}[s] \cup \text{gen}[s] \]

\( \text{gen}(s) \): information generated
\( \text{kill}(s) \): information killed
Data Flow Equations

- Typical Equation

\[ \text{OUT}[s] = \text{IN}[s] - \text{kill}[s] \cup \text{gen}[s] \]

- \text{gen}(s): information generated
- \text{kill}(s): information killed

- Example:
  
  \[
  \begin{align*}
  a &= b \ast c & \text{// generates expression } b \ast c \\
  c &= 5 & \text{// kills expression } b \ast c \\
  d &= b \ast c & \text{// is } b \ast c \text{ redundant here?}
  \end{align*}
  \]
Example Data Flow Analysis

- Reaching Definitions Analysis
- Definition of a variable $x$: $x = \ldots$ something $\ldots$
- Could be more complex (e.g. through pointers, references, implicit)
Reaching Definitions Analysis

- A definition $d$ reaches a point $p$ if
  - there is a path from the point *immediately following* $d$ to $p$
  - $d$ is not “killed” along that path
  - “Kill” means redefinition of the left hand side ($x$ in the earlier example)
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[ \text{IN}(s_1) \]

\[ \text{OUT}(s_1) \]
RD Analysis of a Structured Program

\[
\text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1)
\]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[
\text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1)
\]

\[
\text{GEN}(s_1) =
\]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[ \text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \]

\[ \text{GEN}(s_1) = \{d\} \]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[ \text{IN}(s_1) \]

\[ \text{OUT}(s_1) \]

\[ \text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \]

\[ \text{GEN}(s_1) = \{d\} \]

\[ \text{KILL}(s_1) = \]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[ \text{IN}(s_1) \]

\[ \text{OUT}(s_1) \]

\[ \text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \]

\[ \text{GEN}(s_1) = \{d\} \]

\[ \text{KILL}(s_1) = D_x - \{d\} \text{, where } D_x: \text{set of all definitions of } x \]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[
\begin{align*}
\text{OUT}(s_1) &= \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \\
\text{GEN}(s_1) &= \{d\} \\
\text{KILL}(s_1) &= D_x - \{d\}, \text{where } D_x: \text{set of all definitions of } x \\
\text{KILL}(s_1) &=
\end{align*}
\]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[ \text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \]
\[ \text{GEN}(s_1) = \{d\} \]
\[ \text{KILL}(s_1) = D_x - \{d\}, \text{where } D_x : \text{set of all definitions of } x \]
\[ \text{KILL}(s_1) = D_x \]
RD Analysis of a Structured Program

\[ d : x = y + z \]

\[ \text{OUT}(s_1) = \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \]
\[ \text{GEN}(s_1) = \{d\} \]
\[ \text{KILL}(s_1) = D_x - \{d\}, \text{where } D_x: \text{set of all definitions of } x \]
\[ \text{KILL}(s_1) = D_x \ ? \text{ will also work here but may not work in general} \]
RD Analysis of a Structured Program
RD Analysis of a Structured Program

\[
\text{GEN}(S) = \text{IN}(S) \rightarrow S_1 \rightarrow S_2 \rightarrow \text{OUT}(S)
\]
RD Analysis of a Structured Program

\[ \text{GEN}(S) = \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \]
RD Analysis of a Structured Program

\[ \text{IN}(S) \]

\[ \text{OUT}(S) \]

\[ \text{GEN}(S) = \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \]

\[ \text{KILL}(S) = \]
RD Analysis of a Structured Program

\[
\text{IN}(S) \quad \xrightarrow{S} \quad \text{OUT}(S)
\]

\[
\text{GEN}(S) = \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2)
\]

\[
\text{KILL}(S) = \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2)
\]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{IN}(S) & = \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \\
\text{OUT}(S) & = \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2) \\
\end{align*}
\]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{IN}(S) &= S \\
\text{OUT}(S) &= S \\
\text{GEN}(S) &= \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2) \\
\text{IN}(s_1) &= \text{IN}(S)
\end{align*}
\]
RD Analysis of a Structured Program

IN(S) = \text{IN}(s_1) \cup \text{IN}(s_2)

OUT(S) = \text{OUT}(s_1) \cup \text{OUT}(s_2)

\text{GEN}(S) = \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2)

\text{KILL}(S) = \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2)

\text{IN}(s_1) = \text{IN}(S)

\text{IN}(s_2) = \text{IN}(S)
RD Analysis of a Structured Program

\[ \begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2) \\
\text{IN}(s_1) &= \text{IN}(S) \\
\text{IN}(s_2) &= \text{OUT}(s_1)
\end{align*} \]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2) \\
\text{IN}(s_1) &= \text{IN}(S) \\
\text{IN}(s_2) &= \text{OUT}(s_1) \\
\text{OUT}(S) &=
\end{align*}
\]
RD Analysis of a Structured Program

\[ \text{GEN}(S) = \text{GEN}(s_1) - \text{KILL}(s_2) \cup \text{GEN}(s_2) \]
\[ \text{KILL}(S) = \text{KILL}(s_1) - \text{GEN}(s_2) \cup \text{KILL}(s_2) \]
\[ \text{IN}(s_1) = \text{IN}(S) \]
\[ \text{IN}(s_2) = \text{OUT}(s_1) \]
\[ \text{OUT}(S) = \text{OUT}(s_2) \]
RD Analysis of a Structured Program

IN(S) → S1 → S2 → OUT(S)
RD Analysis of a Structured Program

\[
\text{GEN}(S) = S_1 \rightarrow S_2 \rightarrow S_1
\]
RD Analysis of a Structured Program

\[
\text{GEN}(S) = \text{GEN}(s_1) \cup \text{GEN}(s_2)
\]
RD Analysis of a Structured Program

\[ \text{GEN}(S) = \text{GEN}(s_1) \cup \text{GEN}(s_2) \]

\[ \text{KILL}(S) = \]
RD Analysis of a Structured Program

\[ \text{GEN}(S) = \text{GEN}(s_1) \cup \text{GEN}(s_2) \]
\[ \text{KILL}(S) = \text{KILL}(s_1) \cap \text{KILL}(s_2) \]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) \cap \text{KILL}(s_2) \\
\text{IN}(s_1) &= \\
\end{align*}
\]
**RD Analysis of a Structured Program**

\[
\begin{align*}
\text{IN}(S) &= \text{IN}(s_1) \cup \text{IN}(s_2) \\
\text{OUT}(S) &= \text{OUT}(s_1) \cap \text{OUT}(s_2) \\
\text{GEN}(S) &= \text{GEN}(s_1) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) \cap \text{KILL}(s_2) \\
\text{IN}(s_1) &= \text{IN}(s_2) = \text{IN}(S)
\end{align*}
\]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{IN}(S) &= \text{IN}(s_1) \cup \text{IN}(s_2) \\
\text{OUT}(S) &= \text{OUT}(s_1) \cap \text{OUT}(s_2) \\
\text{GEN}(S) &= \text{GEN}(s_1) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) \cap \text{KILL}(s_2) \\
\text{IN}(s_1) &= \text{IN}(s_2) = \text{IN}(S) \\
\text{OUT}(S) &=
\end{align*}
\]
RD Analysis of a Structured Program

\[ \text{IN}(S) \]

\[ \text{OUT}(S) \]

\[
\begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) \cup \text{GEN}(s_2) \\
\text{KILL}(S) &= \text{KILL}(s_1) \cap \text{KILL}(s_2) \\
\text{IN}(s_1) &= \text{IN}(s_2) = \text{IN}(S) \\
\text{OUT}(S) &= \text{OUT}(s_1) \cup \text{OUT}(s_2)
\end{align*}
\]
RD Analysis of a Structured Program
RD Analysis of a Structured Program

\[ \text{GEN}(S) = \]

\[ \begin{array}{c}
\text{IN}(S) \\
S_1 \\
\text{OUT}(S)
\end{array} \]
RD Analysis of a Structured Program

\[
\text{GEN}(S) = \text{GEN}(s_1)
\]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) \\
\text{KILL}(S) &=
\end{align*}
\]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) \\
\text{KILL}(S) &= \text{KILL}(s_1)
\end{align*}
\]
RD Analysis of a Structured Program

\[ \text{IN}(S) \]

\[ \text{OUT}(S) \]

\[ \text{GEN}(S) = \text{GEN}(s_1) \]

\[ \text{KILL}(S) = \text{KILL}(s_1) \]

\[ \text{OUT}(S) = \]

\[ S \]

\[ s_1 \]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{GEN}(S) & = \text{GEN}(s_1) \\
\text{KILL}(S) & = \text{KILL}(s_1) \\
\text{OUT}(S) & = \text{OUT}(s_1)
\end{align*}
\]
RD Analysis of a Structured Program

\[
\begin{align*}
\text{GEN}(S) &= \text{GEN}(s_1) \\
\text{KILL}(S) &= \text{KILL}(s_1) \\
\text{OUT}(S) &= \text{OUT}(s_1) \\
\text{IN}(s_1) &= 
\end{align*}
\]
RD Analysis of a Structured Program

\[ \text{IN}(S) = \text{IN}(s_1) \]

\[ \text{KILL}(S) = \text{KILL}(s_1) \]

\[ \text{OUT}(S) = \text{OUT}(s_1) \]

\[ \text{IN}(s_1) = \text{IN}(S) \cup \text{GEN}(s_1) \]
RD Analysis is Approximate

Assumption: All paths are feasible.
RD Analysis is Approximate

Assumption: All paths are feasible.

Example:

```java
if (true) s1;
else    s2;
```
RD Analysis is Approximate

Assumption: All paths are feasible.

Example:

```java
if (true) s1;
else s2;
```
RD Analysis is Approximate

Assumption: All paths are feasible.

Example:

```java
if (true) s1;
else s2;
```

<table>
<thead>
<tr>
<th>Fact</th>
<th>Computed</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN(S) =</td>
<td>GEN(s₁) ∪ GEN(s₂) ⊇</td>
<td>GEN(s₁)</td>
</tr>
</tbody>
</table>
RD Analysis is Approximate

- Assumption: All paths are feasible.
- Example:
  ```java
  if (true) s1;
  else s2;
  ```

<table>
<thead>
<tr>
<th>Fact</th>
<th>Computed</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN(S)</td>
<td>GEN(s₁) ∪ GEN(s₂)</td>
<td>⊇ GEN(s₁)</td>
</tr>
<tr>
<td>KILL(S)</td>
<td>KILL(s₁) ∩ KILL(s₂)</td>
<td>⊆ KILL(s₁)</td>
</tr>
</tbody>
</table>
RD Analysis is Approximate

Thus,
RD Analysis is Approximate

Thus,

true $\text{GEN}(S) \subseteq \text{analysis GEN}(S)$
RD Analysis is Approximate

Thus,

\[
\text{true GEN}(S) \subseteq \text{analysis GEN}(S) \\
\text{true KILL}(S) \supseteq \text{analysis KILL}(S)
\]
RD Analysis is Approximate

Thus,
true GEN(S) ⊆ analysis GEN(S)
true KILL(S) ⊇ analysis KILL(S)

More definitions computed to be reaching than actually do!
RD Analysis is Approximate

Thus,

$\text{true \, GEN}(S) \subseteq \text{analysis \, GEN}(S)$

$\text{true \, KILL}(S) \supseteq \text{analysis \, KILL}(S)$

More definitions computed to be reaching than actually do!

Later we shall see that this is **SAFE** approximation
RD Analysis is Approximate

Thus,

\[
\text{true } \text{GEN}(S) \subseteq \text{analysis } \text{GEN}(S)
\]
\[
\text{true } \text{KILL}(S) \supseteq \text{analysis } \text{KILL}(S)
\]

More definitions computed to be reaching than actually do!

Later we shall see that this is **SAFE** approximation

 Prevents optimizations
RD Analysis is Approximate

Thus,
\[ \text{true GEN}(S) \subseteq \text{analysis GEN}(S) \]
\[ \text{true KILL}(S) \supseteq \text{analysis KILL}(S) \]

More definitions computed to be reaching than actually do!
Later we shall see that this is SAFE approximation
  - prevents optimizations
  - but NO wrong optimization
A definition $d$ can reach the start of a block from any of its predecessor

$$
\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)
$$
RD at BB level

- A definition $d$ can reach the start of a block from any of its predecessor
  - if it reaches the end of some predecessor

$$IN(B) = \bigcup_{P \in PRED(B)} OUT(P)$$
A definition $d$ can reach the start of a block from any of its predecessor
  - if it reaches the end of some predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

A definition $d$ reaches the end of a block if

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$
RD at BB level

- A definition $d$ can reach the start of a block from any of its predecessor
  - if it reaches the end of some predecessor
    \[ \text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P) \]

- A definition $d$ reaches the end of a block if
  - either it is generated in the block
    \[ \text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B) \]
RD at BB level

- A definition $d$ can reach the start of a block from any of its predecessor
  - if it reaches the end of some predecessor
    \[ \text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P) \]

- A definition $d$ reaches the end of a block if
  - either it is generated in the block
  - or it reaches block and not killed
    \[ \text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B) \]
Solving RD Constraints

- KILL & GEN known for each BB.
Solving RD Constraints

- KILL & GEN known for each BB.
- A program with $N$ BBs has $2N$ equations with $2N$ unknowns.
Solving RD Constraints

- KILL & GEN known for each BB.
- A program with $N$ BBs has $2N$ equations with $2N$ unknowns.
  - Solution is possible.
Solving RD Constraints

- KILL & GEN known for each BB.
- A program with $N$ BBs has $2N$ equations with $2N$ unknowns.
  - Solution is possible.
  - Iterative approach (on the next slide).
for each block $B$ {
for each block $B$ {
    $\text{OUT}(B) = \emptyset$;
}
for each block $B$ {
    OUT($B$) = $\emptyset$;
}
OUT($Entry$) = $\emptyset$; // note this for later discussion
for each block $B$ {
    OUT($B$) = $\emptyset$;
}
OUT($Entry$) = $\emptyset$; // note this for later discussion
change = true;
while (change) {
    change = false;
for each block $B$ {
    OUT($B$) = $\emptyset$;
}

OUT($Entry$) = $\emptyset$; // note this for later discussion

change = true;
while (change) {
    change = false;
    for each block $B$ other than $Entry$ {

for each block $B$ {
    \text{OUT}(B) = \emptyset;
}

\text{OUT}(\text{Entry}) = \emptyset; \quad \text{// note this for later discussion}

\text{change} = \text{true};

while (change) {
    \text{change} = \text{false};
    for each block $B$ other than \text{Entry} {
        \text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P);
for each block $B$ {
    OUT($B$) = $\emptyset$;
}

OUT($Entry$) = $\emptyset$; // note this for later discussion
change = true;
while (change) {
    change = false;
    for each block $B$ other than $Entry$ {
        IN($B$) = $\bigcup_{P \in \text{PRED}(B)}$ OUT($P$);
        oldOut = OUT($B$);
        OUT($B$) = IN($B$) − KILL($B$) $\cup$ GEN($B$);
    }
}
for each block $B$ {
    \[ \text{OUT}(B) = \emptyset; \]
}
\[ \text{OUT}(Entry) = \emptyset; \] // note this for later discussion
change = true;
while (change) {
    change = false;
    for each block $B$ other than $Entry$ {
        \[ \text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P); \]
        oldOut = \[ \text{OUT}(B); \]
        \[ \text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B); \]
        if (\[ \text{OUT}(B) \neq \text{oldOut} \]) then {
            change = true;
        }
    }
}
Reaching Definitions: Example

ENTRY

B1
- d1: i = m - 1
- d2: j = n
- d3: a = u1

B2
- d4: i = i - 1
- d5: j = j - 1

B3
- d6: a = u2

B4
- d7: i = u3

EXIT
Reaching Definitions: Example

ENTRY

B1
- d1: i = m - 1
- d2: j = n
- d3: a = u1

B2
- d4: i = i - 1
- d5: j = j - 1

B3
- d6: a = u2

B4
- d7: i = u3

EXIT

<table>
<thead>
<tr>
<th>BB</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

ENTRY

B1

d1: i = m - 1
d2: j = n
d3: a = u1

B2

d4: i = i - 1
d5: j = j - 1

B3

d6: a = u2

B4

d7: i = u3

EXIT

<table>
<thead>
<tr>
<th>BB</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{d1, d2, d3}</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

**BB** | **GEN** | **KILL**
--- | --- | ---
B1 | \{d1, d2, d3\} | \{d4, d5, d6, d7\}
B2 |  |  
B3 |  |  
B4 |  |  

**ENTRY**

- **B1**: d1: i = m - 1  
d2: j = n  
d3: a = u1

- **B2**: d4: i = i - 1  
d5: j = j - 1

- **B3**: d6: a = u2

- **B4**: d7: i = u3

**EXIT**
Reaching Definitions: Example

ENTRY

B1
- d1: i = m - 1
- d2: j = n
- d3: a = u1

B2
- d4: i = i - 1
- d5: j = j - 1

B3
- d6: a = u2

B4
- d7: i = u3

EXIT

<table>
<thead>
<tr>
<th>BB</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{d1, d2, d3}</td>
<td>{d4, d5, d6, d7}</td>
</tr>
<tr>
<td>B2</td>
<td>{d4, d5}</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

**BB | GEN | KILL**

| B1 | {d1, d2, d3} | {d4, d5, d6, d7} |
| B2 | {d4, d5} | {d1, d2, d7} |
| B3 | | |
| B4 | | |
Reaching Definitions: Example

**ENTRY**

**B1**
- d1: i = m - 1
- d2: j = n
- d3: a = u1

**B2**
- d4: i = i - 1
- d5: j = j - 1

**B3**
- d6: a = u2

**B4**
- d7: i = u3

**EXIT**

<table>
<thead>
<tr>
<th>BB</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{d1, d2, d3}</td>
<td>{d4, d5, d6, d7}</td>
</tr>
<tr>
<td>B2</td>
<td>{d4, d5}</td>
<td>{d1, d2, d7}</td>
</tr>
<tr>
<td>B3</td>
<td>{d6}</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

<table>
<thead>
<tr>
<th>BB</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{d1, d2, d3}</td>
<td>{d4, d5, d6, d7}</td>
</tr>
<tr>
<td>B2</td>
<td>{d4, d5}</td>
<td>{d1, d2, d7}</td>
</tr>
<tr>
<td>B3</td>
<td>{d6}</td>
<td>{d3}</td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

<table>
<thead>
<tr>
<th>BB</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{d1, d2, d3}</td>
<td>{d4, d5, d6, d7}</td>
</tr>
<tr>
<td>B2</td>
<td>{d4, d5}</td>
<td>{d1, d2, d7}</td>
</tr>
<tr>
<td>B3</td>
<td>{d6}</td>
<td>{d3}</td>
</tr>
<tr>
<td>B4</td>
<td>{d7}</td>
<td></td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

**BB** | **GEN** | **KILL**
--- | --- | ---
B1 | \{d1, d2, d3\} | \{d4, d5, d6, d7\}
B2 | \{d4, d5\} | \{d1, d2, d7\}
B3 | \{d6\} | \{d3\}
B4 | \{d7\} | \{d1, d4\}
Reaching Definitions: Example

<table>
<thead>
<tr>
<th>Pass#</th>
<th>Pt</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>IN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OUT</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

**Example**

- **d1**: $i = m - 1$
- **d2**: $j = n$
- **d3**: $a = u_1$
- **d4**: $i = i - 1$
- **d5**: $j = j - 1$
- **d6**: $a = u_2$
- **d7**: $i = u_3$
Reaching Definitions: Example

<table>
<thead>
<tr>
<th>Pass#</th>
<th>Pt</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>IN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>IN</td>
<td>∅</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d3, d4, d5, d6</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d4, d5, d6</td>
<td>d3, d5, d6, d7</td>
</tr>
</tbody>
</table>

\[ \text{d1: } i = m - 1 \\
\text{d2: } j = n \\
\text{d3: } a = u1 \]

\[ \text{d4: } i = i - 1 \\
\text{d5: } j = j - 1 \]

\[ \text{d6: } a = u2 \]

\[ \text{d7: } i = u3 \]

ENTRY

B1

B2

B3

B4

EXIT
Reaching Definitions: Example

<table>
<thead>
<tr>
<th>Pass#</th>
<th>Pt</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>IN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>IN</td>
<td>∅</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d3, d4, d5</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d4, d5, d6</td>
<td>d3, d5, d6</td>
</tr>
<tr>
<td>2</td>
<td>IN</td>
<td>∅</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d3, d4, d5</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d4, d5, d6</td>
<td>d3, d5, d6</td>
</tr>
</tbody>
</table>
Reaching Definitions: Example

<table>
<thead>
<tr>
<th>Pass#</th>
<th>Pt</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>IN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>IN</td>
<td>∅</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d3, d4, d5, d6</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5</td>
<td>d4, d5, d6</td>
<td>d3, d5, d6, d7</td>
</tr>
<tr>
<td>2</td>
<td>IN</td>
<td>∅</td>
<td>d1, d2, d3, d5, d6, d7</td>
<td>d3, d4, d5, d6, d7</td>
<td>d3, d4, d5, d6</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5, d6</td>
<td>d4, d5, d6</td>
<td>d3, d5, d6, d7</td>
</tr>
<tr>
<td>3</td>
<td>IN</td>
<td>∅</td>
<td>d1, d2, d3, d5, d6, d7</td>
<td>d3, d4, d5, d6, d7</td>
<td>d3, d4, d5, d6</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>d1, d2, d3</td>
<td>d3, d4, d5, d6</td>
<td>d4, d5, d6</td>
<td>d3, d5, d6, d7</td>
</tr>
</tbody>
</table>
Reaching Definitions: Bitvectors

a bit for each definition:

\[ d1 \quad d2 \quad d3 \quad d4 \quad d5 \quad d6 \quad d7 \]
a bit for each definition:
\[
\begin{array}{ccccccc}
d1 & d2 & d3 & d4 & d5 & d6 & d7 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Pass#</th>
<th>Pt</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>IN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td>000000</td>
<td>000000</td>
<td>000000</td>
<td>000000</td>
</tr>
<tr>
<td>1</td>
<td>IN</td>
<td>000000</td>
<td>111000</td>
<td>001110</td>
<td>001110</td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td>111000</td>
<td>001110</td>
<td>000111</td>
<td>001011</td>
</tr>
<tr>
<td>2</td>
<td>IN</td>
<td>000000</td>
<td>111011</td>
<td>001110</td>
<td>001110</td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td>111000</td>
<td>001110</td>
<td>000111</td>
<td>001011</td>
</tr>
<tr>
<td>3</td>
<td>IN</td>
<td>000000</td>
<td>111011</td>
<td>001110</td>
<td>001110</td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td>111000</td>
<td>001110</td>
<td>000111</td>
<td>001011</td>
</tr>
</tbody>
</table>
Reaching Definitions: Bitvectors

- Set-theoretic definitions:

\[
\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)
\]

\[
\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)
\]
Reaching Definitions: Bitvectors

- Set-theoretic definitions:

\[
\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)
\]

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

- Bitvector definitions:

\[
\text{IN}(B) = \bigvee_{P \in \text{PRED}(B)} \text{OUT}(P)
\]

\[
\text{OUT}(B) = \text{IN}(B) \land \neg \text{KILL}(B) \lor \text{GEN}(B)
\]
Reaching Definitions: Bitvectors

- Set-theoretic definitions:

\[
\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)
\]

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

- Bitvector definitions:

\[
\text{IN}(B) = \bigvee_{P \in \text{PRED}(B)} \text{OUT}(P)
\]

\[
\text{OUT}(B) = \text{IN}(B) \land \neg \text{KILL}(B) \lor \text{GEN}(B)
\]

- Bitwise $\lor$, $\land$, $\neg$ operators
Reaching Definitions: Application

Constant Folding

while changes occur {

Reaching Definitions: Application

Constant Folding

while changes occur {
    forall the stmts S of the program {

Reaching Definitions: Application

Constant Folding

while changes occur {
    forall the stmts $S$ of the program {
        foreach operand $B$ of $S$ {
            ...
Constant Folding

while changes occur {
  forall the stmts S of the program {
    foreach operand B of S {
      if there is a unique definition of B
        that reaches S and is a constant C {
}
Reaching Definitions: Application

Constant Folding

while changes occur {
  forall the stmts S of the program {
    foreach operand B of S {
      if there is a unique definition of B that reaches S and is a constant C {
        replace B by C in S;
      }
    }
  }
}
Reaching Definitions: Application

Constant Folding

while changes occur {
  forall the stmts S of the program {
    foreach operand B of S {
      if there is a unique definition of B that reaches S and is a constant C {
        replace B by C in S;
        if all operands of S are constant {
          ...
        }
      }
    }
  }
}
Constant Folding

while changes occur {
    forall the stmts S of the program {
        foreach operand B of S {
            if there is a unique definition of B that reaches S and is a constant C {
                replace B by C in S;
                if all operands of S are constant {
                    replace rhs by eval(rhs);
                }
        }
    }
}
Constant Folding

while changes occur {
  forall the stmts S of the program {
    foreach operand B of S {
      if there is a unique definition of B that reaches S and is a constant C {
        replace B by C in S;
        if all operands of S are constant {
          replace rhs by eval(rhs);
          mark definition as constant;
        }
      }
    }
  }
}
Reaching Definitions: Application

- Recall the approximation in reaching definition analysis
Reaching Definitions: Application

- Recall the approximation in reaching definition analysis
  $\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$
Reaching Definitions: Application

- Recall the approximation in reaching definition analysis
  
  \[ \text{true GEN}(S) \subseteq \text{analysis GEN}(S) \]
  
  \[ \text{true KILL}(S) \supseteq \text{analysis KILL}(S) \]
Recall the approximation in reaching definition analysis:

- True GEN(S) ⊆ Analysis GEN(S)
- True KILL(S) ⊇ Analysis KILL(S)

- Can it cause the application to infer
Reaching Definitions: Application

- Recall the approximation in reaching definition analysis:
  \[ \text{true GEN}(S) \subseteq \text{analysis GEN}(S) \]
  \[ \text{true KILL}(S) \supseteq \text{analysis KILL}(S) \]

- Can it cause the application to infer:
  - an expression as a constant when it has different values for different executions?
Recall the approximation in reaching definition analysis

\[ \text{true } \text{GEN}(S) \subseteq \text{analysis } \text{GEN}(S) \]
\[ \text{true } \text{KILL}(S) \supseteq \text{analysis } \text{KILL}(S) \]

Can it cause the application to infer

- an expression as a constant when it is has different values for different executions?
- an expression as not a constant when it is a constant for all executions?
Recall the approximation in reaching definition analysis
true GEN(S) ⊆ analysis GEN(S)
true KILL(S) ⊇ analysis KILL(S)

Can it cause the application to infer
- an expression as a constant when it is has different values for different executions?
- an expression as not a constant when it is a constant for all executions?

Safety? Profitability?
Reaching Definitions: Summary

$\text{Gen}(B) = \left\{ d_x \mid d_x \text{ in } B \text{ defines variable } x \text{ and is not followed by another definition of } x \text{ in } B \right\}$
Reaching Definitions: Summary

- **GEN**($B$) = \( \{ d_x \mid d_x \text{ in } B \text{ defines variable } x \text{ and is not followed by another definition of } x \text{ in } B \} \)

- **KILL**($B$) = \( \{ d_x \mid B \text{ contains some definition of } x \} \)
Reaching Definitions: Summary

- \( \text{GEN}(B) = \left\{ d_x \mid d_x \text{ in } B \text{ defines variable } x \text{ and is not followed by another definition of } x \text{ in } B \right\} \)
- \( \text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \} \)
- \( \text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P) \)
Reaching Definitions: Summary

- $\text{GEN}(B) = \left\{ d_x \mid d_x \text{ in } B \text{ defines variable } x \text{ and is not followed by another definition of } x \text{ in } B \right\}$
- $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$
- $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$
- $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$
Reaching Definitions: Summary

- GEN($B$) = \{ $d_x$ | $d_x$ in $B$ defines variable $x$ and is not followed by another definition of $x$ in $B$ \}

- KILL($B$) = \{ $d_x$ | $B$ contains some definition of $x$ \}

- IN($B$) = $\bigcup_{P \in \text{PRED}(B)}$ OUT($P$)

- OUT($B$) = IN($B$) − KILL($B$) $\cup$ GEN($B$)

- meet ( $\land$ ) operator: The operator to combine information coming along different predecessors is $\bigcup$
Reaching Definitions: Summary

- $\text{GEN}(B) = \left\{ d_x \mid d_x \text{ in } B \text{ defines variable } x \text{ and is not followed by another definition of } x \text{ in } B \right\}$
- $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$
- $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$
- $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$
- meet ($\land$) operator: The operator to combine information coming along different predecessors is $\bigcup$
- What about the Entry block?
Entry block has to be initialized specially:

\[
\begin{align*}
\text{OUT}(\text{Entry}) & = \text{EntryInfo} \\
\text{EntryInfo} & = \emptyset
\end{align*}
\]
Entry block has to be initialized specially:

\[
\text{OUT}(\text{Entry}) = \text{EntryInfo}
\]
\[
\text{EntryInfo} = \emptyset
\]

A better entry info could be:

\[
\text{EntryInfo} = \{ x = \text{undefined} \mid x \text{ is a variable} \} 
\]
Reaching Definitions: Summary

- Entry block has to be initialized specially:

\[
\text{OUT}(\text{Entry}) = \text{EntryInfo} \\
\text{EntryInfo} = \emptyset
\]

- A better entry info could be:

\[
\text{EntryInfo} = \{ x = \text{undefined} \mid x \text{ is a variable} \}
\]

- Why?