Static Program Analysis Part III
Data Flow Analysis

- Data-flow analysis provides information for compiling and SE tasks by computing the flow of different types of data to points in the program
- For structured programs, data-flow analysis can be performed on an AST
- In general, intra-procedural (global) data-flow analysis is performed on the Control Flow Graph
- Exact solutions to most problems are undecidable
  - May depend on input
  - May depend on outcome of a conditional statement
  - May depend on termination of loop
- We compute approximations of the exact solution
• **Data-flow testing**
  – suppose that a statement assigns a value but the use of that value is never executed under test

```plaintext
a = c + 10
```

“a” not used on this path

```plaintext
d = a + y
```

– need **definition-use pairs (du-pairs)**: associations between definitions and uses of the same variable or memory location
Data Flow Analysis for Debugging

• **Debugging**
  – suppose that \( a \) has the incorrect value in the statement

\[
a = c + y
\]

  – need **data dependence information**: statements that can affect the incorrect value at this point
Data Flow Problems – Reaching & Uses

- Compute the flow of data to points in the program - e.g.,
  - Where does the assignment to I in statement 1 reach?
  - Where does the expression computed in statement 2 reach?
  - Which uses of variable J are reachable from the end of B1?
  - Is the value of variable I live after statement 3?

- Interesting points before and after basic blocks or statements

B1

1. I := 2
2. J := I + 1

B2

3. I := 1

B3

4. J := J + 1

B4

5. J := J - 4
Data Flow Problems – Reaching Definitions

- A *definition* of a variable or memory location is a point or statement where that variable gets a value - e.g., input statement, assignment statement.

- X *reaches* a point P if there exists a control-flow path in the CFG from the definition to P with no other definitions of X on the path (called a *definition-clear path*)

- Such a path may exist in the graph but may not be executable (i.e., there may be no input to the program that will cause it to be executed); such a path is *infeasible*.

1. $I := 2$
2. $J := I + 1$
3. $I := 1$
4. $J := J + 1$
5. $J := J - 4$
A use of a variable or memory location is a point or statement where that variable is referenced but not changed - e.g., used in a computation, used in a conditional, output.

Use of X is reachable from a point P if there exists a control-flow path in the CFG from the P to the use with no definitions of X on the path.

Reachable uses also called upwards exposed uses.
Reachable Uses Example

- **Definitions:**
  - I: 1, 3
  - J: 2, 4, 5

- **Uses:**
  - I: 2, 4
  - J: 4, 5

- **Reachable Uses:**
  - I from 1: 2
  - I from 3: 4
  - J from 2: 4
  - J from 4: 4, 5
  - J from 5:
DU-Chains, UD-chains, Webs

- A definition-use chain or DU-chain for a definition $D$ of variable $V$ connects the $D$ to all uses of $V$ that it can reach.
- A use-definition chain or UD-chain for a use $U$ of variable $V$ connects $U$ to all definitions of $V$ that reach it.
- A web for a variable is the maximal union of intersecting DU-chains.
Data-Dependence

- A *data-dependence graph* has one node for every basic block and one edge representing the flow of data between the two nodes.
- X is *data dependent* on Y iff there exists a variable v such that:
  - Y has a definition of v and
  - X has a use of v and
  - There exists a control path from Y to X along which v is not redefined.
- Different types of data dependence edges can be defined:
  - Flow: def to use (most common)
  - Anti: use to def
  - Out: def to def
Data (flow) Dependence Graph

\[
\begin{align*}
X &= 1 \\
Z &> 2 \\
Y &= X + 1
\end{align*}
\]

\[
\begin{align*}
X &= 2 \\
Z &= X - 3 \\
X &= 4
\end{align*}
\]

\[
\begin{align*}
Z &= X + 7
\end{align*}
\]

Control flow graph
Control Dependence

• A statement S1 is *control dependent* on a statement S2 if the outcome of S2 determines whether S1 is reached in the CFG

• We define control dependence for language constructs

• Control dependencies can be derived for arbitrary control flow using the concept of post dominator of *conditional* instructions
Definitions

if Y then B1 else B2;
• X is control dependent on Y iff X is in B1 or B2

while Y do B;
• X is control dependent on Y iff X is in B
Program-Dependence Graph

- A *program dependence graph* (PDG) for a program $P$ is the combination of the control-dependence graph for $P$ and the data-dependence graph for $P$
- Can be used for
  - Redundant code analysis
  - I/O relation analysis
  - Program slicing
Compute a PDG

1. **read** (n)  
2. \( i := 1 \)  
3. \( \text{sum} := 0 \)  
4. \( \text{product} := 1 \)  
5. **while** \( i \leq n \) **do**  
6. \( \text{sum} := \text{sum} + i \)  
7. \( \text{product} := \text{product} \times i \)  
8. \( i := i + 1 \)  
9. **write** (sum)  
10. **write** (product)

Identify control dependencies via CFG and conditionals

Identify data dependencies via definition/uses
Computing a PDG

1. **read** (n)
2. i := 1
3. sum := 0
4. product := 1
5. **while** i <= n **do**
   6. sum := sum + i
   7. product := product * i
   8. i := i + 1
9. **write** (sum)
10. **write** (product)

6,7,8 are control dependent on 5

DU-Chains:
(1,5)
(2,5), (2,6), (2,7), (2,8),
(8,5), (8,6), (8,7), (8,8)
(3,6), (3,9), (6,6), (6,6),
(6,9)
(4,7), (4,10), (7,7), (7,10)